prajwal_problemset_3a.R

Aaryan Agarwal

2023-11-20

```
# PROBLEM SET 3A
# This assignment is done in a group of three
# Prajwal Kaushal, Sumukha Sharma, Aaryan Agarwal


#Part 1: R Questions

###############################################################################
#Question 1: Data Import and tidying
###############################################################################

###############################################################################
# TIBBLES ANND DATA FRAMES

#1. We can tell if an objest is a tibble in 2 ways:
# -> By printing:  When you print a tibble, it shows the first 10 rows and
# all columns that fit on the screen, along with the type of each column.
# This is different from regular data frames, which may print many more rows
# and potentially not fit well on the screen.
# -> Class check: Using the class() function in R.
# For a tibble, it will return "tbl_df", "tbl", and "data.frame",
# indicating it's a tibble and a data frame.
# For a regular data frame like mtcars, it will just return "data.frame".

#2.
df <- data.frame(abc = 1, xyz = "a")
df$x
```

```
## [1] "a"
```

```
df[, "xyz"]
```

```
## [1] "a"
```

```
df[, c("abc", "xyz")]
```

```
##   abc xyz
## 1   1   a
```

```
# df$x: Accessing a column.
# data.frame: Returns the column x. If x doesn't exist, it returns NULL.
# tibble: Same behavior as a data.frame.
# df[, "xyz"]: Extracting a single column.
# data.frame: Returns a vector.
# tibble: Returns a tibble with one column.
# This is a key difference and can be more consistent for programming
# since the output type is predictable.
# df[, c("abc", "xyz")]: Extracting multiple columns.
# data.frame: Returns a data.frame with the specified columns.
# tibble: Similar behavior, but with tibble specific formatting and printing.

###############################################################################
# DATA IMPORT

#1.
# we cann use read_delim() from the readr package in R.
# This function allows you to specify any delimiter, including "|".
# Example: read_delim(file, delim = "|")

#2.
# read_fwf() is used for reading fixed width files.
# The important arguments are:
# file: The path to the file or a connection.
# col_positions: This is crucial as it defines the start and end points of each
# column. we can use fwf_positions(), fwf_widths(), or fwf_empty() to specify this.
# col_types: To specify the type of each column.
# It helps in controlling how columns are read and preventing unnecessary type conversions.
# Other common arguments like col_names, na, skip, etc., can also be important
# depending on the specific needs of the data import.

#3.
# To read text like "x,y\n1,'a,b'" using read_delim(), we need to specify the
# delimiter and the quote character.
# Since the data is using a comma as a delimiter and a single quote (') for quoting,
# the command would be:
# read_delim("x,y\n1,'a,b'", delim = ",", quote = "'")
# Here, delim = "," specifies that the fields are separated by commas,
# and quote = "'" tells the function to recognize single quotes as the quoting character,

###############################################################################
# PARSING VECTORS

#1. If we set both decimal_mark and grouping_mark to the same character in R,
# it will result in an error because the parser won't be able to differentiate between
# the decimal point and the thousands separator.
# This would make parsing numbers ambiguous.
# When we set decimal_mark to ",", the default value of grouping_mark typically changes to "."
# in many international ares where a comma is used as a decimal separator.
# Conversely, if we set grouping_mark to ".", the default decimal_mark often becomes ","

#2.
# Europe: UTF-8 is widely used due to its ability to handle a vast range of characters
```

```r
# and its compatibility with various languages and systems.
# Other encodings like ISO-8859-1 is also used
# Asia: Some common encodings include Big5, GB 2312, GBK, and HZ.
# These encodings are particularly used for Chinese characters.
# Other Asian languages also have specific encodings,
# but the widespread adoption of Unicode and UTF-8 has significantly reduced the
# dependency on these region-specific encodings.

################################################################################
# SPREADING AND GATHERING

#1.
library(tibble)
library(tidyr)
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
```

```r
stocks <- tibble(
  year = c(2015, 2015, 2016, 2016),
  half = c( 1, 2, 1, 2),
  return = c(1.88, 0.59, 0.92, 0.17)
)
stocks %>%
  spread(year, return) %>%
  gather("year", "return", `2015`:`2016`)
```

```
## # A tibble: 4 x 3
##    half year  return
##   <dbl> <chr>  <dbl>
## 1     1 2015    1.88
## 2     2 2015    0.59
## 3     1 2016    0.92
## 4     2 2016    0.17
```

```r
# In the given example, the spread() function transforms the stocks tibble by
# spreading the year column into multiple columns, each named after a specific year
# and containing values from the return column.
# However, when you reverse the process with gather(),
# the year becomes a character type instead of numeric because column
# names are always characters.
# This change in data type is one reason why gather() and spread() are not
```

```r
# perfectly symmetrical.

#2.
# The convert argument attempts to automatically convert the result columns to
# the appropriate data type. By default, when spreading or gathering, the columns
# are character type.
# When convert is set to TRUE, it tries to infer and convert the data types
# based on the data content.

#3.
# The code fails because gather() expects column names or indices as the second
# and third arguments, but '1999' and '2000' are interpreted as numbers,
# not as column names.

#4.
# The spreading fails because there are duplicate combinations of name and key.
# For eg. "Phillip Woods" has two entries for "age".
# To fix this, we can add a new column that creates a unique identifier for each row
# before spreading.

#5.
preg <- tribble(
  ~pregnant, ~male, ~female,
  "yes", NA, 10,
  "no", 20, 12
)

# Transforming the data into a tidy format
tidy_preg <- preg %>%
  gather(key = "gender", value = "count", -pregnant)

tidy_preg
```

```
## # A tibble: 4 x 3
##   pregnant gender count
##   <chr>    <chr>  <dbl>
## 1 yes      male      NA
## 2 no       male      20
## 3 yes      female    10
## 4 no       female    12
```

```r
# The variables are "pregnant", "gender", and "count".
# After gathering, you would have a tidy format where each row is an observation
# with a "pregnant" status, "gender" (male or female), and the corresponding count.

################################################################################
# SEPARATING AND UNITING
#1.

# extra and fill are arguments in the separate() function that control how
# additional pieces of the split string are handled.

tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
```

```
  separate(x, c("one", "two", "three"))
```

```
## Warning: Expected 3 pieces. Additional pieces discarded in 1 rows [2].
```

```
## # A tibble: 3 x 3
##   one   two   three
##   <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     f
## 3 h     i     j
```

```
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"))
```

```
## Warning: Expected 3 pieces. Missing pieces filled with `NA` in 1 rows [2].
```

```
## # A tibble: 3 x 3
##   one   two   three
##   <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     <NA>
## 3 f     g     i
```

```
# First dataset experiment
tibble(x = c("a,b,c", "d,e,f,g", "h,i,j")) %>%
  separate(x, c("one", "two", "three"), extra = "merge", fill = "right")
```

```
## # A tibble: 3 x 3
##   one   two   three
##   <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     f,g
## 3 h     i     j
```

```
# For the first dataset, extra = "merge" will merge any extra pieces into
# the last column, and fill = "right" will fill missing values from the right with NA.

# Second dataset experiment
tibble(x = c("a,b,c", "d,e", "f,g,i")) %>%
  separate(x, c("one", "two", "three"), extra = "drop", fill = "right")
```

```
## # A tibble: 3 x 3
##   one   two   three
##   <chr> <chr> <chr>
## 1 a     b     c
## 2 d     e     <NA>
## 3 f     g     i
```

```
# For the second dataset, extra = "drop" will drop any extra pieces,
# and fill = "right" again fills missing values from the right with NA.

#2.
# The remove argument determines whether the original columns that are being
# united or separated should be removed from the resulting data frame.
# When set to TRUE (default), the original columns are removed.
# When set to FALSE, the original columns are retained alongside the new columns.

###############################################################################
# MISSING VALUES

#1.

# The fill argument in spread() is used to replace NA values that appear in
# the spread data. When spreading a key-value pair across a wider format,
# any missing combinations will result in NA.
# The fill argument allows to specify a value that should replace these NAs.
# In complete(), the fill argument also replaces NA values,
# but it does so in a different context.
# complete() is used to expand a dataset to include all combinations
# of specified keys, filling in NA where data does not exist.
# The fill argument in complete() lets us specify values to replace these NAs
# across the newly created rows.

#2.

# The direction argument in the fill() function specifies the direction in
# which to fill missing values (NA).
# The options are:
# down: Fills values downwards (from top to bottom).
# up: Fills values upwards (from bottom to top).
# downup: First fills downwards, then upwards.
# This is useful when you want to fill NAs with the
# nearest non-NA value either above or below.
# updown: First fills upwards, then downwards.

###############################################################################
#Question 2: Relational Data and Data Types
###############################################################################

###############################################################################
# RELATIONAL DATA

# 1.
# Variables Needed: We need the geographical coordinates (latitude and longitude)
# of both the origin and destination airports.
# Tables Needed: You would combine data from the flights table (which includes
# origin and destination airport codes) with the airports table
# (which provides the coordinates of each airport).

# 2.
# The relationship is likely based on the location.
```

```r
# The weather data would correspond to the airports based on the airport's
# geographical location.

# 3.
# If the weather table contained records for all airports in the US, it would
# define an additional relationship with the destination airports in the flights table.
# This means there would be two relationships for weather: one with the origin airport
# and another with the destination airport in the flights table.

# 4.
# we can create a data frame with dates and an indicator of whether the day is special
# (e.g., a holiday).
# The primary key would be the date.
# This table could be connected to the flights table through the date,
# allowing analysis of flight patterns on these special days

##############################################################################
#  KEYS

#1.

library(nycflights13)

flights_with_key <- flights %>%
  mutate(surrogate_key = row_number())

# In this code the mutate() function is used to add a new column called
# surrogate_key to the flights table.
# The row_number() function generates a sequence of numbers from 1 to
# the number of rows in the table, ensuring each row has a unique identifier.


##############################################################################
# MUTATING JOINS

#1.

library(ggplot2)
```

```
## Warning: package 'ggplot2' was built under R version 4.2.2
```

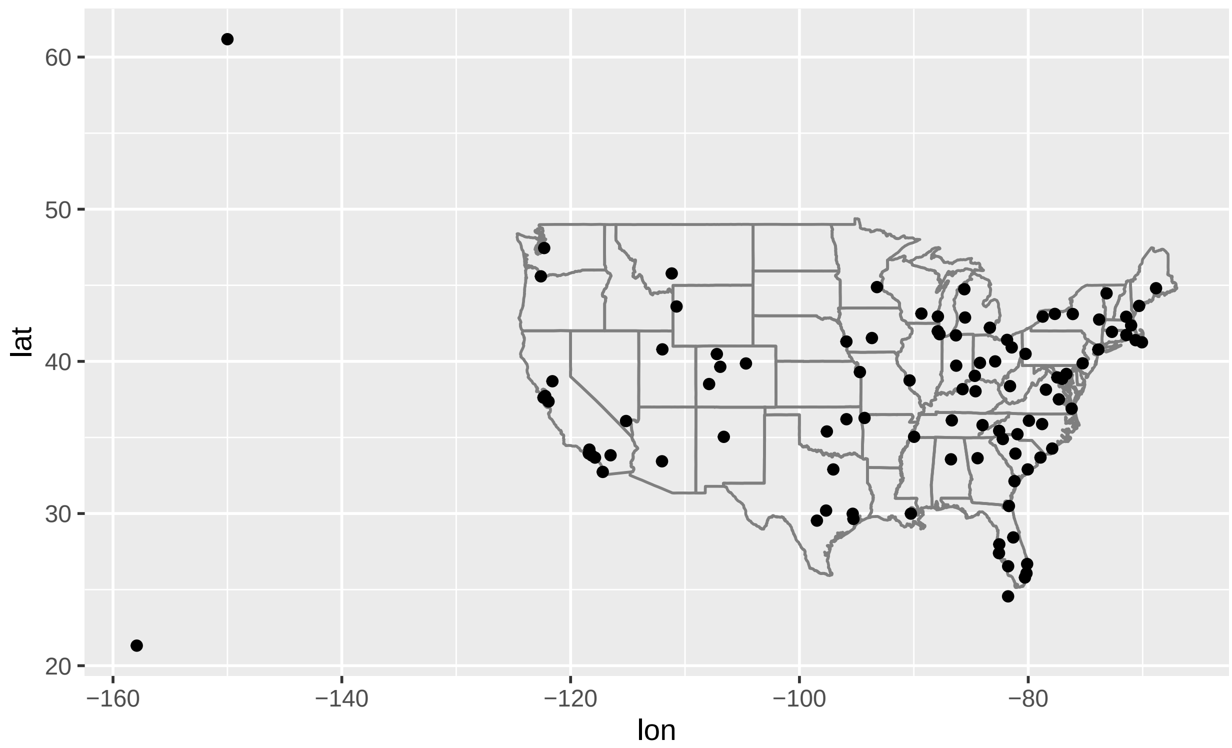
```r
# install.packages("maps")
library(maps)
```

```
## Warning: package 'maps' was built under R version 4.2.3
```

```r
library(nycflights13)

airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  ggplot(aes(lon, lat)) +
  borders("state") +
```

```
  geom_point() +
  coord_quickmap()
```

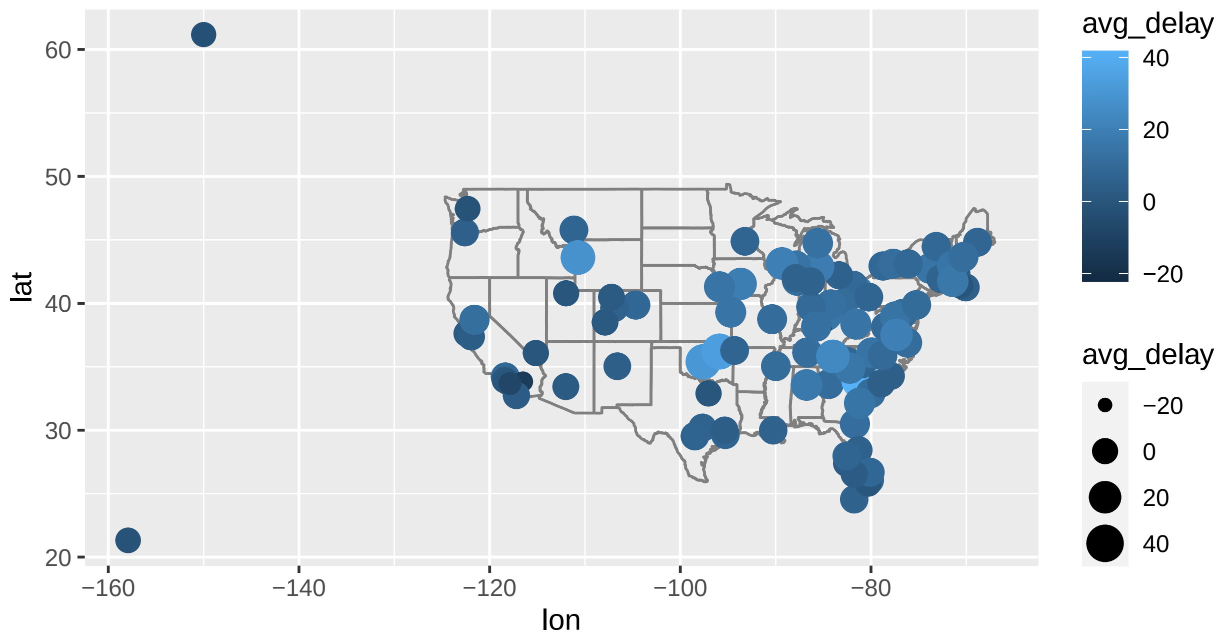



```
# Calculate average delay by destination
avg_delay_by_dest <- flights %>%
  group_by(dest) %>%
  summarize(avg_delay = mean(arr_delay, na.rm = TRUE))

# Join with airports data
delay_airports <- airports %>%
  semi_join(flights, c("faa" = "dest")) %>%
  inner_join(avg_delay_by_dest, by = c("faa" = "dest"))

# Plot the map
ggplot(delay_airports, aes(x = lon, y = lat, size = avg_delay, color = avg_delay)) +
  borders("state") +
  geom_point() +
  coord_quickmap()
```

```
## Warning: Removed 1 rows containing missing values (`geom_point()`).
```




```
# 2.

flights_with_loc <- flights %>%
  left_join(airports, c("origin" = "faa")) %>%
  rename(origin_lat = lat, origin_lon = lon) %>%
  left_join(airports, c("dest" = "faa")) %>%
  rename(dest_lat = lat, dest_lon = lon)

# 3.

# Joining flights and planes tables
flights_planes_joined <- flights %>%
  left_join(planes, by = "tailnum")

# Calculating the age of the planes using the 'year.y' column
flights_planes_joined <- flights_planes_joined %>%
  mutate(plane_age = 2013 - year.y) %>%  # Using 'year.y' for plane manufacture year
  filter(!is.na(plane_age))  # Removing rows where plane age could not be calculated

# Analyzing the relationship between plane age and delays
correlation_result <- cor(flights_planes_joined$plane_age, flights_planes_joined$arr_delay, use = "compl

correlation_result
```

```
## [1] -0.01767153
```

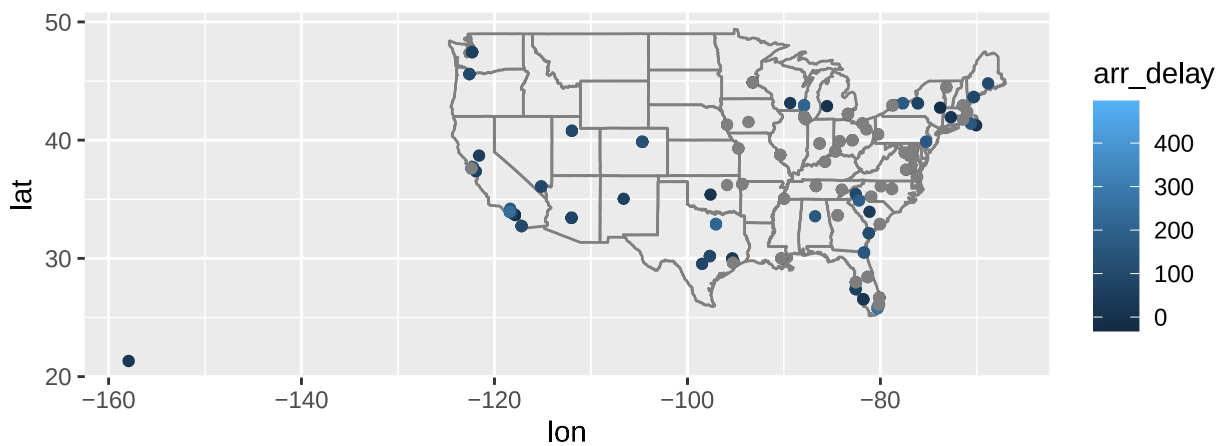
```r
# A correlation coefficient of approximately -0.01767 suggests a very weak negative
# relationship between the age of the plane and its delays

#4.

# Filter flights on June 13, 2013
june_13_flights <- flights %>%
  filter(month == 6, day == 13, year == 2013) %>%
  left_join(airports, c("dest" = "faa"))

# Plotting
ggplot(june_13_flights, aes(x = lon, y = lat, color = arr_delay)) +
  borders("state") +
  geom_point() +
  coord_quickmap()
```

```
## Warning: Removed 21 rows containing missing values (`geom_point()`).
```




```r
################################################################################
# FILTERING JOINS

#1.

# Counting flights per plane
plane_flight_count <- flights %>%
```

```
  group_by(tailnum) %>%
  summarize(flight_count = n()) %>%
  filter(flight_count >= 100)


plane_flight_count
```

```
## # A tibble: 1,218 x 2
##    tailnum flight_count
##    <chr>          <int>
##  1 N0EGMQ           371
##  2 N10156           153
##  3 N10575           289
##  4 N11106           129
##  5 N11107           148
##  6 N11109           148
##  7 N11113           138
##  8 N11119           148
##  9 N11121           154
## 10 N11127           124
## # ... with 1,208 more rows
```

```
# Filtering flights
flights_with_frequent_planes <- flights %>%
  semi_join(plane_flight_count, by = "tailnum")

flights_with_frequent_planes
```

```
## # A tibble: 230,902 x 19
##     year month   day dep_time sched_de~1 dep_d~2 arr_t~3 sched~4 arr_d~5 carrier
##    <int> <int> <int>    <int>      <int>   <dbl>   <int>   <int>   <dbl> <chr>
##  1  2013     1     1      517        515       2     830     819      11 UA
##  2  2013     1     1      533        529       4     850     830      20 UA
##  3  2013     1     1      544        545      -1    1004    1022     -18 B6
##  4  2013     1     1      554        558      -4     740     728      12 UA
##  5  2013     1     1      555        600      -5     913     854      19 B6
##  6  2013     1     1      557        600      -3     709     723     -14 EV
##  7  2013     1     1      557        600      -3     838     846      -8 B6
##  8  2013     1     1      558        600      -2     849     851      -2 B6
##  9  2013     1     1      558        600      -2     853     856      -3 B6
## 10  2013     1     1      558        600      -2     923     937     -14 UA
## # ... with 230,892 more rows, 9 more variables: flight <int>, tailnum <chr>,
## #   origin <chr>, dest <chr>, air_time <dbl>, distance <dbl>, hour <dbl>,
## #   minute <dbl>, time_hour <dttm>, and abbreviated variable names
## #   1: sched_dep_time, 2: dep_delay, 3: arr_time, 4: sched_arr_time,
## #   5: arr_delay
```

```
#2.

# Finding top 48 hours with worst delays
top_delays <- flights %>%
  group_by(year, month, day, hour) %>%
```

```r
  summarize(total_delay = sum(dep_delay, na.rm = TRUE)) %>%
  arrange(desc(total_delay)) %>%
  slice_head(n = 48)
```

```
## `summarise()` has grouped output by 'year', 'month', 'day'. You can override
## using the `.groups` argument.
```

```r
top_delays
```

```
## # A tibble: 6,936 x 5
## # Groups:   year, month, day [365]
##     year month   day  hour total_delay
##    <int> <int> <int> <dbl>       <dbl>
##  1  2013     1     1    17        1908
##  2  2013     1     1    18        1456
##  3  2013     1     1    13        1100
##  4  2013     1     1    16        1044
##  5  2013     1     1    14         828
##  6  2013     1     1    19         722
##  7  2013     1     1    20         602
##  8  2013     1     1    15         513
##  9  2013     1     1    12         322
## 10  2013     1     1     9         299
## # ... with 6,926 more rows
```

```r
# Joining with weather data
weather_delays <- top_delays %>%
  left_join(weather, by = c("year", "month", "day", "hour"))

weather_delays
```

```
## # A tibble: 20,720 x 16
## # Groups:   year, month, day [365]
##     year month   day  hour total_delay origin  temp  dewp humid wind_dir wind_~1
##    <int> <int> <int> <dbl>       <dbl> <chr>  <dbl> <dbl> <dbl>    <dbl>   <dbl>
##  1  2013     1     1    17        1908 EWR     36.0  19.0  49.8      330    11.5
##  2  2013     1     1    17        1908 JFK     37.0  17.1  43.8      330    16.1
##  3  2013     1     1    17        1908 LGA     36.0  17.1  45.8      300    18.4
##  4  2013     1     1    18        1456 EWR     34.0  15.1  45.4      310    12.7
##  5  2013     1     1    18        1456 JFK     35.1  14    41.5      310    15.0
##  6  2013     1     1    18        1456 LGA     34.0  16.0  47.2      320    15.0
##  7  2013     1     1    13        1100 EWR     39.2  28.4  69.7      330    16.1
##  8  2013     1     1    13        1100 JFK     37.9  26.6  64.7      340    15.0
##  9  2013     1     1    13        1100 LGA     37.9  25.0  59.2      310    16.1
## 10  2013     1     1    16        1044 EWR     37.0  19.9  49.6      300    13.8
## # ... with 20,710 more rows, 5 more variables: wind_gust <dbl>, precip <dbl>,
## #   pressure <dbl>, visib <dbl>, time_hour <dttm>, and abbreviated variable
## #   name 1: wind_speed
```

```r
#3.
```

```r
# Checking if each plane is associated with a single airline
plane_airline_relation <- flights %>%
  group_by(tailnum) %>%
  summarize(airlines = n_distinct(carrier)) %>%
  filter(!is.na(tailnum))

# Checking for planes associated with more than one airline
multi_airline_planes <- plane_airline_relation %>%
  filter(airlines > 1)

multi_airline_planes
```

```
## # A tibble: 17 x 2
##    tailnum airlines
##    <chr>      <int>
##  1 N146PQ         2
##  2 N153PQ         2
##  3 N176PQ         2
##  4 N181PQ         2
##  5 N197PQ         2
##  6 N200PQ         2
##  7 N228PQ         2
##  8 N232PQ         2
##  9 N933AT         2
## 10 N935AT         2
## 11 N977AT         2
## 12 N978AT         2
## 13 N979AT         2
## 14 N981AT         2
## 15 N989AT         2
## 16 N990AT         2
## 17 N994AT         2
```

```r
# There are 17 planes in the nycflights13 dataset that are associated with more than one airline,
# as each of these planes has an airlines count of 2.
# This finding rejects the hypothesis that each plane is exclusively flown by a single airline.
# In this dataset, at least, some planes are operated by multiple airlines.

################################################################################
# STRINGS

#1.
# paste() concatenates strings with a separator between them. By default, this separator is a space.
# paste0() is a variation of paste() that uses an empty string as the separator,
# effectively concatenating strings without any space.
# In the stringr package, the equivalent function is str_c().
# Regarding NA handling: paste() turns NA into "NA" (a string), while paste0() does the same.
# In contrast, str_c() will return NA if any of the inputs is NA, unless na.rm = TRUE is specified.

#2.
# The sep argument specifies the string to use between each element when concatenating.
# The collapse argument is used when combining multiple strings into a single string.
# It specifies the separator to use between the combined strings.
```

```
#3.
# str_wrap() wraps a string into formatted lines of a specified width. This is useful for
# creating text with a specific width for display purposes, such as in console output or
# when formatting text for reports.

#4.
# str_trim() trims whitespace from the start and end of a string.
# The opposite function could be considered to be adding spaces or padding to a string, which can be do
# functions like str_pad() in stringr. However, this is not a direct opposite as str_pad() requires spe
# of the desired string length and padding character.

##########################################################################################################

#Part 2 - Project

##########################################################################################################
#An interesting data set we came across was the General Social Survey (GSS)
#dataset. It is a high-quality survey which gathers data on American society
#and opinions, and it conducted since 1972. Our research delves into understanding the impact of socio-
#factors on educational attainment in urban and rural settings.
#The primary question revolves around identifying the key determinants that influence
#educational outcomes, particularly focusing on the influence of household income, parental education,
#and geographical location. The data set can be accessed in
#R using the library infer. The dataset present in R is a sample of the
#original dataset of 500 entries from the original with a span of years
#1973-2018.It includes demographic markers and some economic variables.
#It contains of 11 variables namely year (year the respondent was surveyed),
#age (age of the respondent at the time of the survey), sex (gender of the
#respondent which is self-identified by them), college
#(whether the respondent has a valid college degree or no),
#partyid (respondents political party affiliation),
#hompop (number of people in the respondents house),
#hours (number of hours the respondent works while he was being surveyed),
#income (total family income of the respondent), class
#(subjective socioeconomic class identification), finrela
#(opinion of family income) and weight (survey weight). The data set consists
#of just 500 rows of data.
#We can use this dataset to generate the average number of people living
#in each household in a certain year. We can chart out the slope of the '
#increase or the decrease in the number of people in each household.
#We can determine how much an average worker works each week and
#the average salary they get for each hour. We can group the previous
#result based on the class of the individual. We can determine which political
#party is likely to succeed in that area during a specific year. The literacy
#rate of the area can be determined on whether a person has achieved a degree
#or not. Many such inferences can be made through this dataset by various
#statistical methods. We can group the dataset based upon the years by
#splitting the dataset and can determine many inferences according to the year.
#Same can be done by splitting the dataset by class or political party
#preferences.
#Its good data because we can infer many different conditions as given above
#and it gives us a lot of potential. The original dataset is available on the
#gss website and should be easily accessible. A shorter format is available
```

```r
#in the infer library in R if for some reason we are not able to process the
#data.

library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.2 --

## v readr   2.1.3     v stringr 1.5.0
## v purrr   0.3.5     v forcats 0.5.2

## Warning: package 'stringr' was built under R version 4.2.2

## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## x purrr::map()    masks maps::map()
```

```r
library(infer)
```

```
## Warning: package 'infer' was built under R version 4.2.2
```

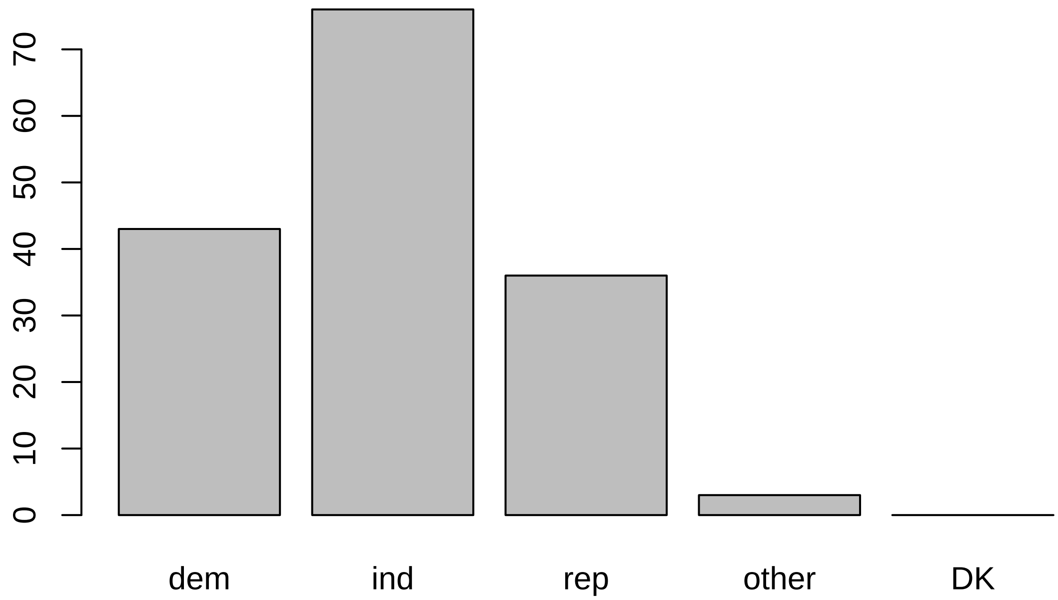
```r
library(ggplot2)
data<-gss

summary(data)
```

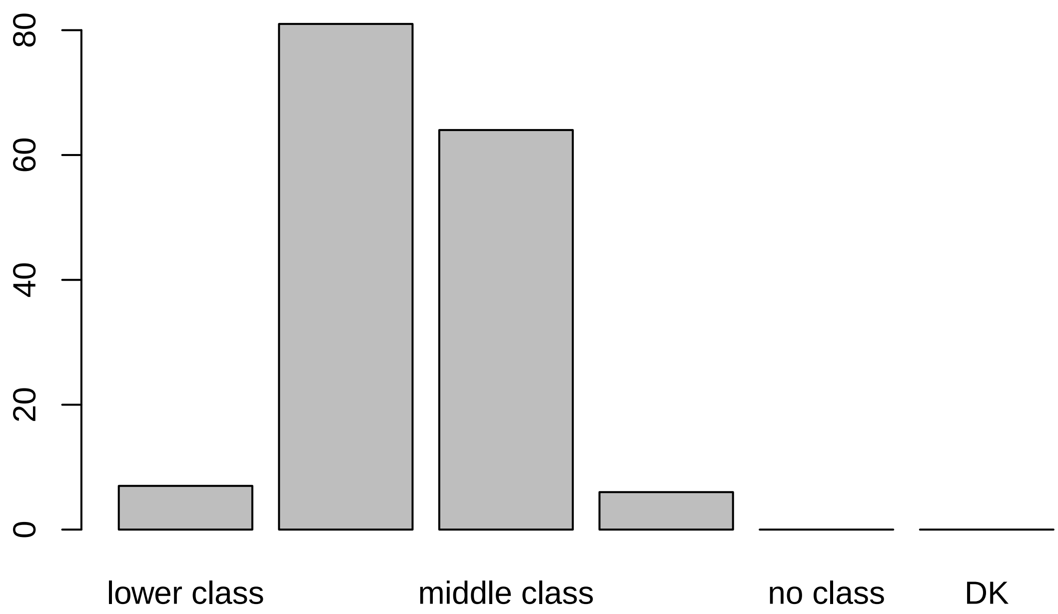
```
##       year           age            sex            college      partyid
##  Min.   :1973   Min.   :18.00   male  :263   no degree:326   dem  :177
##  1st Qu.:1985   1st Qu.:29.00   female:237   degree   :174   ind  :192
##  Median :1996   Median :38.00                                rep  :123
##  Mean   :1995   Mean   :40.27                                other:  8
##  3rd Qu.:2006   3rd Qu.:50.00                                DK   :  0
##  Max.   :2018   Max.   :87.00
##
##      hompop           hours                      income               class
##  Min.   : 1.000   Min.   : 3.00   $25000 or more:303   lower class   : 20
##  1st Qu.: 2.000   1st Qu.:36.75   $20000 - 24999: 60   working class:251
##  Median : 3.000   Median :40.00   $10000 - 14999: 49   middle class :209
##  Mean   : 2.858   Mean   :41.38   $15000 - 19999: 42   upper class   : 20
##  3rd Qu.: 4.000   3rd Qu.:48.00   $8000 to 9999 : 10   no class      :  0
##  Max.   :11.000   Max.   :89.00   $5000 to 5999 :  7   DK            :  0
##                                   (Other)       : 29
##               finrela        weight
##  far below average: 25   Min.   :0.4119
##  below average    :115   1st Qu.:0.8550
##  average          :239   Median :1.0127
##  above average    :106   Mean   :1.0541
##  far above average: 10   3rd Qu.:1.0985
##  DK               :  5   Max.   :5.2439
##
```

```
#Data collected after the year 2000
newdata1<-filter(data,year>2000)
plot(newdata1['sex'])
```


```
plot(newdata1['partyid'])
```

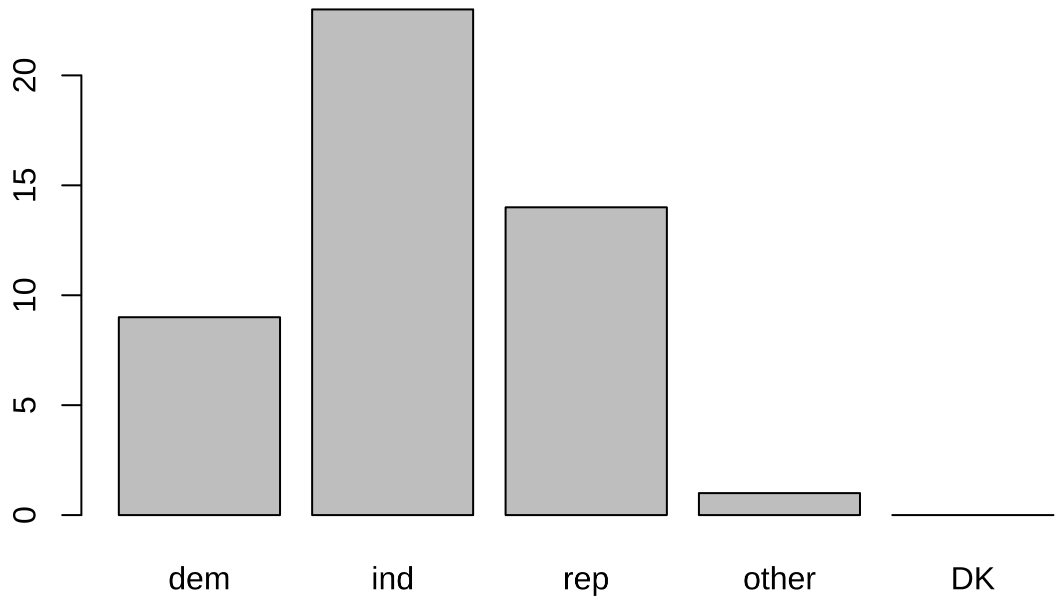



```
plot(newdata1['class'])
```

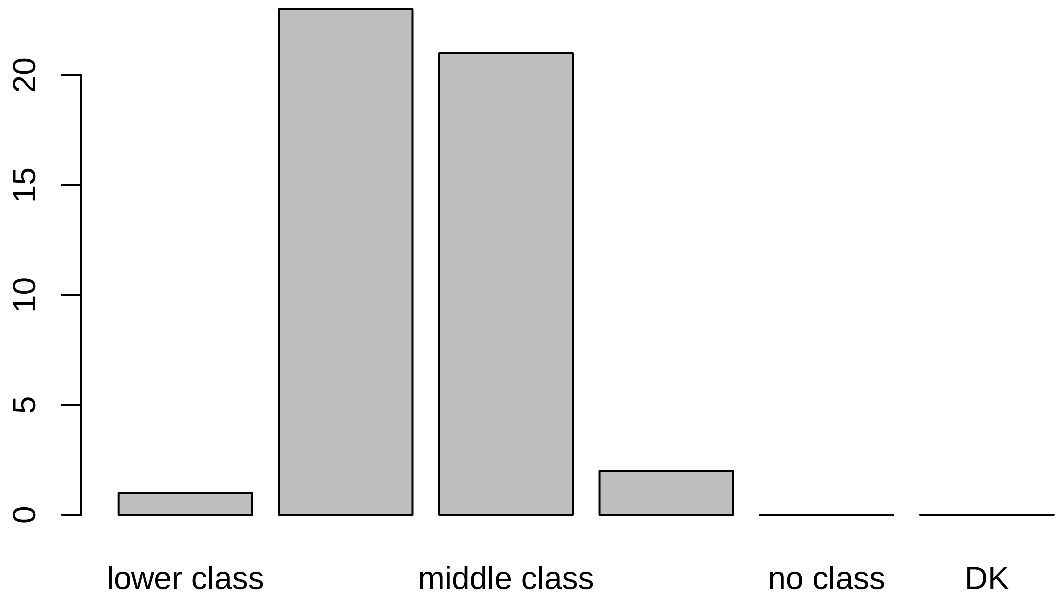



```
#Data collected after the year 2000 and the survey weight is greater than 1
newdata2<-filter(data,year>2000 & weight>1)
plot(newdata2['sex'])
```

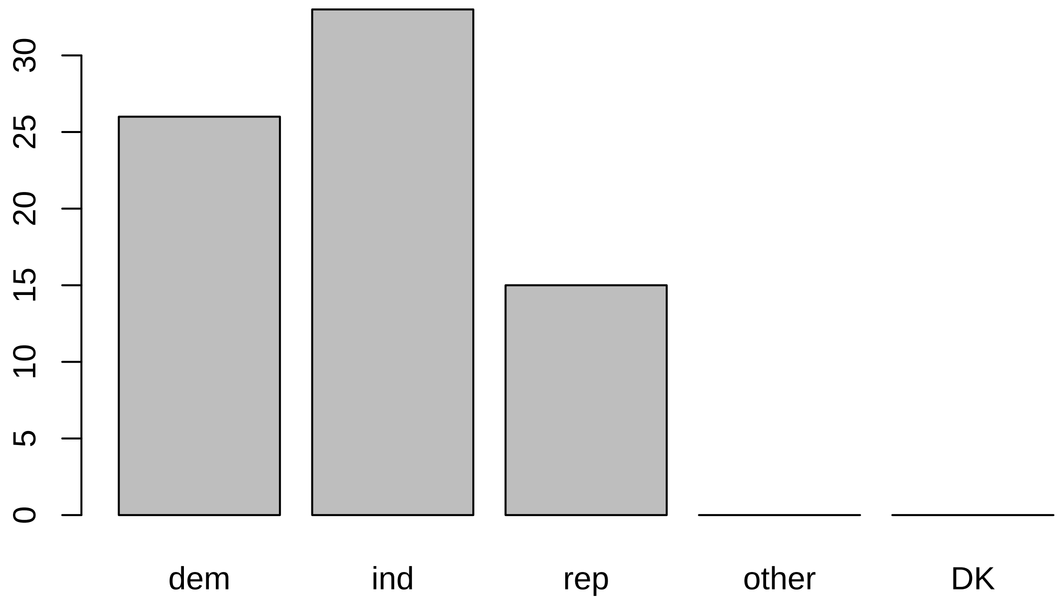

```
plot(newdata2['partyid'])
```



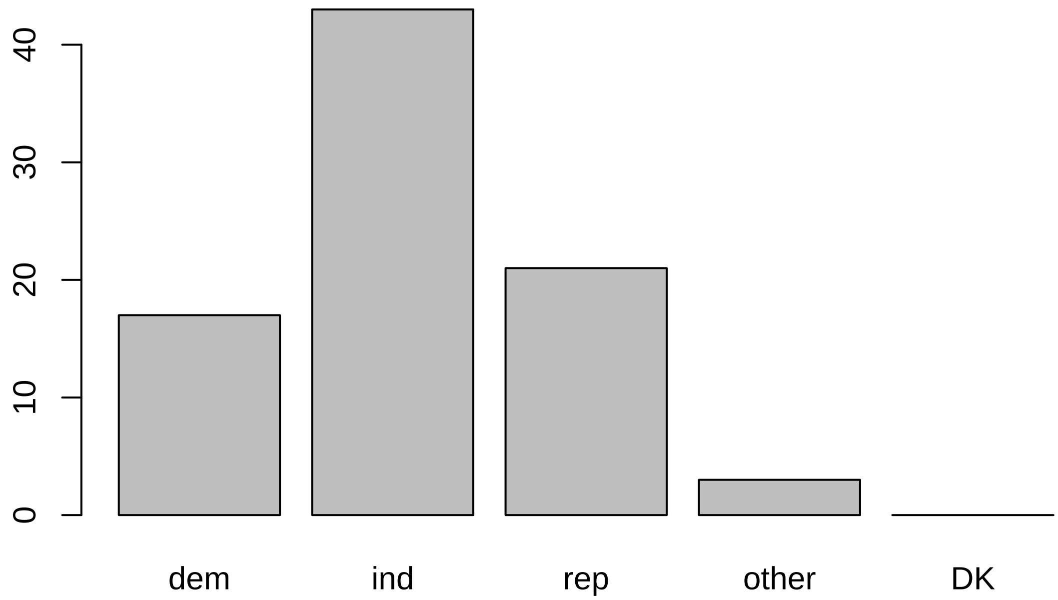

```
plot(newdata2['class'])
```



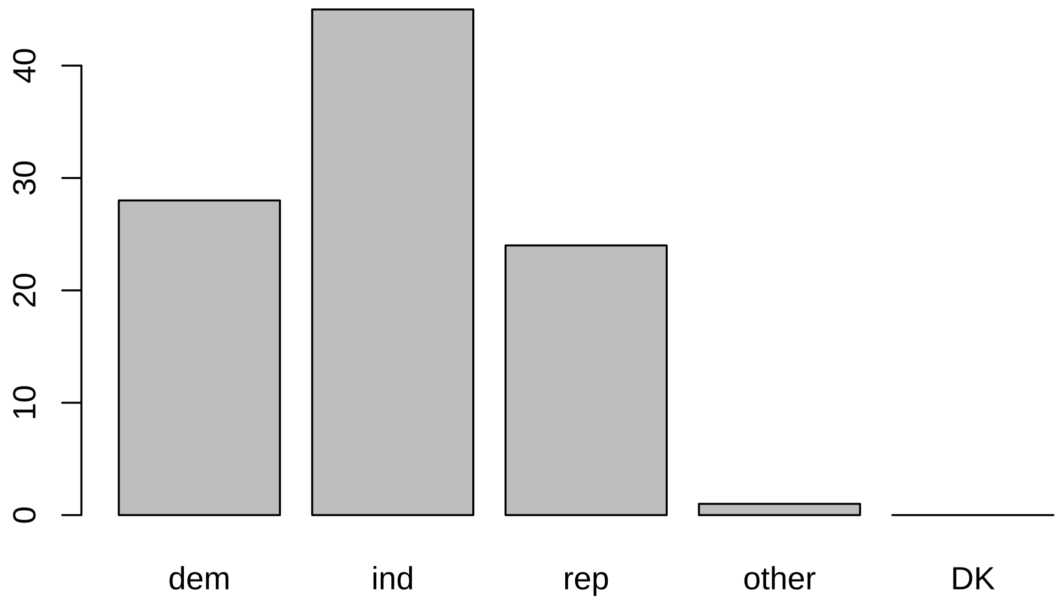

```
#We can see that the females have a higher survey weight than the men
#Data collected from men and women respectively
newdata3<-filter(data,year>2000 & sex=='female')
plot(newdata3['partyid'])
```



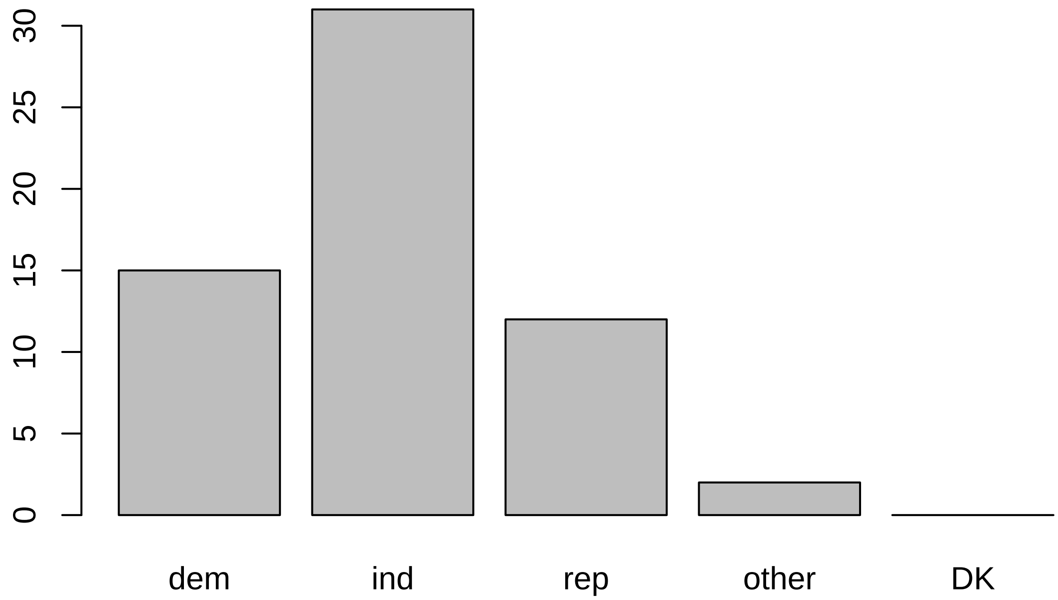

```
newdata4<-filter(data,year>2000 & sex=='male')
plot(newdata4['partyid'])
```




```
#We can see that the females tend to vote for the democratic party
#less than the males
#Data collected from people above and below the age of 35 respectively
newdata5<-filter(data,year>2000 & age>35)
plot(newdata5['partyid'])
```




```
newdata6<-filter(data,year>2000 & age<=35)
plot(newdata6['partyid'])
```




```
#We can see that the people below the age of 35 have less confidence
#in the democratic party than the people above the age of 35
```