Project 1 – R Practice Summary Report

ALY6000

Prepared By: Muhammad Umer

Presented To: Prof. John Wilder

Date: 2024/09/22

Introduction and Key Findings

Overview:

The course "Introduction to Analytics" marks my first step into the world of R programming and its practical applications in data analysis. As someone new to R, I initially found it challenging to determine how to start. However, the structured resources provided in the course, along with supportive references and tutorials, allowed me to gradually familiarize myself with the basic syntax, functions, and principles of R. Through consistent practice, I have begun to understand how R can be used to manipulate data, perform statistical operations, and create visualizations, all of which are essential skills for analyzing and interpreting data effectively.

Key Findings:

1. **Vector Operations**: Throughout the project, I worked extensively with vector creation and manipulation, which highlighted R's ability to efficiently handle sequences and repetitive data structures. It was fascinating to see how flexible vectors can be for organizing and analyzing data.
2. **Statistical Functions**: I used a variety of R's built-in functions such as sum(), mean(), median(), max(), and min() to quickly perform statistical analyses on vectors. These functions provided a simple way to summarize data and extract meaningful insights.
3. **Data Manipulation**: Using the tidyverse library, I learned how to read, filter, arrange, and modify data from a CSV file (ds_salaries.csv).
4. **Visualization**: I got the opportunity to dive into data visualization using ggplot2. By creating a bar chart that compared job titles and salaries, I saw how powerful visual representation can be in simplifying complex datasets.
5. **Random Number Generation**: The project also introduced me to random number generation and distribution through functions like runif() and rnorm(). This was a great way to explore R's capabilities for simulating data and generating insights from random distributions.

Through this module, what I loved most was the practical experience of working with real-world data. By analyzing a dataset of data science salaries, I was able to convert currencies and make comparisons across job titles, which gave me a real sense of how R can be applied in professional contexts.

Output Screenshots

1. Computed snapshot of Mathematical functions:

75 %% 10 = **5** (remainder when 75 is divided by 10)

```
> 123 * 453
[1] 55719
> 5^2 * 40
[1] 1000
> TRUE & FALSE
[1] FALSE
> TRUE | FALSE
[1] TRUE
> 75 %% 10
[1] 5
> 75 / 10
[1] 7.5
```

2. First vector with c function with random values:

C function combines values into vectors

```
>
> first_vector <- c(17,12,-33,5)
> print(first_vector)
[1]  17  12 -33   5
>
```

3. Counting by fives vector:

```
> #counting by fives, this can also be achieved using seq(5,35,5)
> counting_by_fives <- c(5, 10, 15, 20, 25, 30, 35)
> print(counting_by_fives)
[1]  5 10 15 20 25 30 35
>
```

4. Vector using **seq** function containing even number between 10 and 30 inclusive:

```
> #Even numbers using seq function
> second_vector <- seq(10, 30, by = 2) #generates sequences of numbers
> print(second_vector)
 [1] 10 12 14 16 18 20 22 24 26 28 30
>
```

5. Counting by fives vector using **seq:**

```
> #Counting by fives with seq
> counting_by_fives_with_seq <- seq(5,35, by = 5)
> print(counting_by_fives_with_seq)
[1]  5 10 15 20 25 30 35
>
```

6. Vector using **rep function:**

rep(first_vector,10)

rep: repeats values a specified number of times

```
> third_vector <- rep(first_vector,10) #repeats values a specified number of times
> print(third_vector)
 [1]  17  12 -33   5  17  12 -33   5  17  12 -33   5  17  12 -33   5  17  12 -33   5  17
[22]  12 -33   5  17  12 -33   5  17  12 -33   5  17  12 -33   5  17  12 -33   5
>
```

7. vector containing the number zero, 20 times:

rep(0,20)

```
> #vector containing the number zero, 20 times
> rep_vector <- rep(0,20)
> print(rep_vector)
 [1] 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
>
```

8. Vector using the range operator containing numbers from 10 to 1:

10:1

OR

Seq(10,1, by = -1)

```
> #Vector using range operator
> fourth_vector <- 10:1 #other way: seq(10, 1, by = -1)
> print(fourth_vector)
 [1] 10  9  8  7  6  5  4  3  2  1
>
```

9. Vector using the range operator that contains the numbers from 5 to 15:

```
> #vector using range with numbers from 5 to 15
> counting_vector <- 5:15
> print(counting_vector)
 [1]  5  6  7  8  9 10 11 12 13 14 15
>
```

Range function generates a sequence of numbers within a specified interval as shown in above snapshot.

10. Vector with the values (96, 100, 85, 92, 81, 72) stored in variable grades:

```
> grades <- c(96,100,85,92,81,72)
> print(grades)
[1]  96 100  85  92  81  72
```

11. Adding number into vector grades:

grades + 3

```
> #Adding number to each index in vector
> bonus_points_added <- grades + 3
> print(bonus_points_added)
[1]  99 103  88  95  84  75
```

12. Vector with the values 1 – 100(not typed):

```
> #Seq of numbers using range function
> one_to_one_hundred <- 1:100 #other way: seq(1,100,1)
> print(one_to_one_hundred)
  [1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20  21
 [22]  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42
 [43]  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63
 [64]  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84
 [85]  85  86  87  88  89  90  91  92  93  94  95  96  97  98  99 100
>
```

Two ways to do that one is using range (the colon) function and other using seq function.
I used range function.

13. Reverse Numbers : Vector with values from 100 to -100 by 3s:

```
> #Reverse number by -3
> reverse_numbers <- seq(100,-100, by = -3)
> print(reverse_numbers)
 [1] 100  97  94  91  88  85  82  79  76  73  70  67  64  61  58  55  52  49  46  43  40
[22]  37  34  31  28  25  22  19  16  13  10   7   4   1  -2  -5  -8 -11 -14 -17 -20 -23
[43] -26 -29 -32 -35 -38 -41 -44 -47 -50 -53 -56 -59 -62 -65 -68 -71 -74 -77 -80 -83 -86
[64] -89 -92 -95 -98
>
```

14.

```
> # Adding 20 to each element of second_vector
> second_vector + 20
 [1] 30 32 34 36 38 40 42 44 46 48 50
>
> # Multiplying each element of second_vector by 20
> second_vector * 20
 [1] 200 240 280 320 360 400 440 480 520 560 600
>
> # Checking which elements of second_vector are greater than or equal to 20
> second_vector >= 20
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> # Checking which elements of second_vector are not equal to 20
> second_vector != 20 # != means "not equal"
 [1]  TRUE  TRUE  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
>
```

15. Compute the sum using built in Sum Function:

```
> total <- sum(one_to_one_hundred) # Sum all the index in vector
> print(total)
[1] 5050
>
```

Sum function: Sum all the index in vector

16. Compute average using mean function:

```
> #mean: calculates the average of given numbers
> average_value <- mean(one_to_one_hundred)
> print(average_value)
[1] 50.5
>
```

17. Median of given numbers:

```
> median_value <- median(one_to_one_hundred) # returns the middle value of data
> print(median_value)
[1] 50.5
>
```

18. Maximum Value in Data?
19. Minimum Value in Data?

```
> max_value <- max(one_to_one_hundred) #returns the largest value in data
> print(max_value)
[1] 100
>
>
> min_value <- min(one_to_one_hundred) #returns the smallest value in data
> print(min_value)
[1] 1
```

20. Extract the first value from second vector using brackets:

```
> first_value <- second_vector[1] #directly accessing the index using bracket
> print(first_value)
[1] 10
```

Note: In R index starts from 0 and we can access any index using brackets.

21. Extract the first, second and third values second vector using brackets:

```
> first_three_values <- second_vector[1:3] #other way: [c(1,2,3)] or seq
> print(first_three_values)
[1] 10 12 14
>
```

22. Extract the 1st, 5th, 10th, and 11th elements of second vector

```
> vector_from_brackets <- second_vector[c(1,5,10,11)]
> print(vector_from_brackets)
[1] 10 18 28 30
>
```

I found it tricky at first but c function made it look easy.

23. Extract elements using bracket by passing c(FALSE, TRUE, FALSE, TRUE).

```
> #The TRUE values in the vector indicate which elements to keep, while the FALSE values indi
cate which elements to discard. In this case, only the elements corresponding to TRUE will be
included in vector_from_boolean_brackets
> vector_from_boolean_brackets <- first_vector[c(FALSE, TRUE, FALSE, TRUE)]
> print(vector_from_boolean_brackets)
[1] 12  5
>
```

24.

```
> second_vector >= 20 # Checking which elements of second_vector are greater than or equal to
20
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
>
```

Second_vector >= 20 checks which element of second vector are greater than or equal to 20.
It returns false where condition does not satisfy and true where it satisfies

25.

```
> ages_vector <- seq(from = 10, to = 30, by = 2) # Generates vector of even numbers
> print(ages_vector)
 [1] 10 12 14 16 18 20 22 24 26 28 30
>
```

It generates a vector of even numbers from 10 to 30, incrementing by 2, and stores it in the variable `ages_vector`

26.

```
> ages_vector[ages_vector >= 20] # returns the elements of ages_vector that are greater than
or equal to 20
[1] 20 22 24 26 28 30
>
```

Ages_vector[ages_vector >= 20] extracts and return the elements of ages_vector that are greater than or equal to 20, resulting in a new vector containing only those values where condition is true.

27.

```
> lowest_grades_removed <- grades[grades >= 85]
> print(lowest_grades_removed)
[1]  96 100  85  92
>
```

Code in above snapshot extracts and retains only the elements of `grades` that are greater than or equal to 85, resulting in a new vector stored in `lowest_grades_removed`.

28. Remove 3rd and 4th element from grades:

```
> middle_grades_removed <- grades[-c(3,4)]
> print(middle_grades_removed)
[1]  96 100  81  72
>
```

This is how specifying the negative indices we can remove elements from the vector.

29. Remove 5th and 10th element from second_vector:

```
> fifth_vector <- second_vector[-c(5,10)]
> print(fifth_vector)
[1] 10 12 14 16 20 22 24 26 30
>
```

30.

```
> set.seed(5) # Sets the random number generator's seed for reproducibility
> random_vector <- runif(n=10, min = 0, max = 1000) # Generates a vector of 10 random numbers
uniformly distributed between 0 and 1000
> print(random_vector)
 [1] 200.2145 685.2186 916.8758 284.3995 104.6501 701.0575 527.9600 807.9352 956.5001
[10] 110.4530
>
```

The set.seed(5) function initializes the random number generator, ensuring that the sequence of random numbers generated can be reproduced in future runs.
The runif(n = 10, min = 0, max = 1000) function generates a vector containing 10 random numbers uniformly distributed between 0 and 1000, storing the result in the variable random_vector.

31. Compute the total of random_vector using sum function:

```
> sum_vector <- sum(random_vector)
> print(sum_vector)
[1] 5295.264
>
```

32. Use the cumsum function to compute the cumulative sum of random_vector:

```
> cumsum_vector <- cumsum(random_vector)
> print(cumsum_vector)
 [1]  200.2145  885.4330 1802.3088 2086.7083 2191.3584 2892.4159 3420.3759 4228.3111
 [9] 5184.8112 5295.2642
>
```

cumsum function calculates the cumulative sum of elements. Cumulative sum starts with the first element and adds each subsequent number to the total.

33. Mean of random_vector:

```
> #Mean of random vector
> mean_vector <- mean(random_vector)
> print(mean_vector)
[1] 529.5264
>
```

34. Find Standard Deviation of random vector:

```
> # Calculate the standard deviation of random_vector
> sd_vector <- sd(random_vector)
> print(sd_vector)
[1] 331.3606
>
```

Standard deviation (SD) is a measure of how spread out the numbers in a dataset are.

Steps to Calculate Standard Deviation:

1. **Find the mean (average)** of the data.
2. **Subtract the mean** from each number to get the deviation of each number.
3. **Square the deviations** to remove negative values.
4. **Find the average** of these squared deviations (this is called variance).
5. **Take the square root** of the variance to get the standard deviation.

35. Use the round function to round the values of random_vector:

```
> # Round the elements of random_vector
> round_vector <- round(random_vector)
> print(round_vector)
 [1] 200 685 917 284 105 701 528 808 957 110
>
```

Round the number to its nearest whole number.

36. Sort the values:

```
> # Sort the elements of random_vector in ascending order
> sort_vector <- sort(random_vector)
> print(sort_vector)
 [1] 104.6501 110.4530 200.2145 284.3995 527.9600 685.2186 701.0575 807.9352 916.8758
[10] 956.5001
> 
```

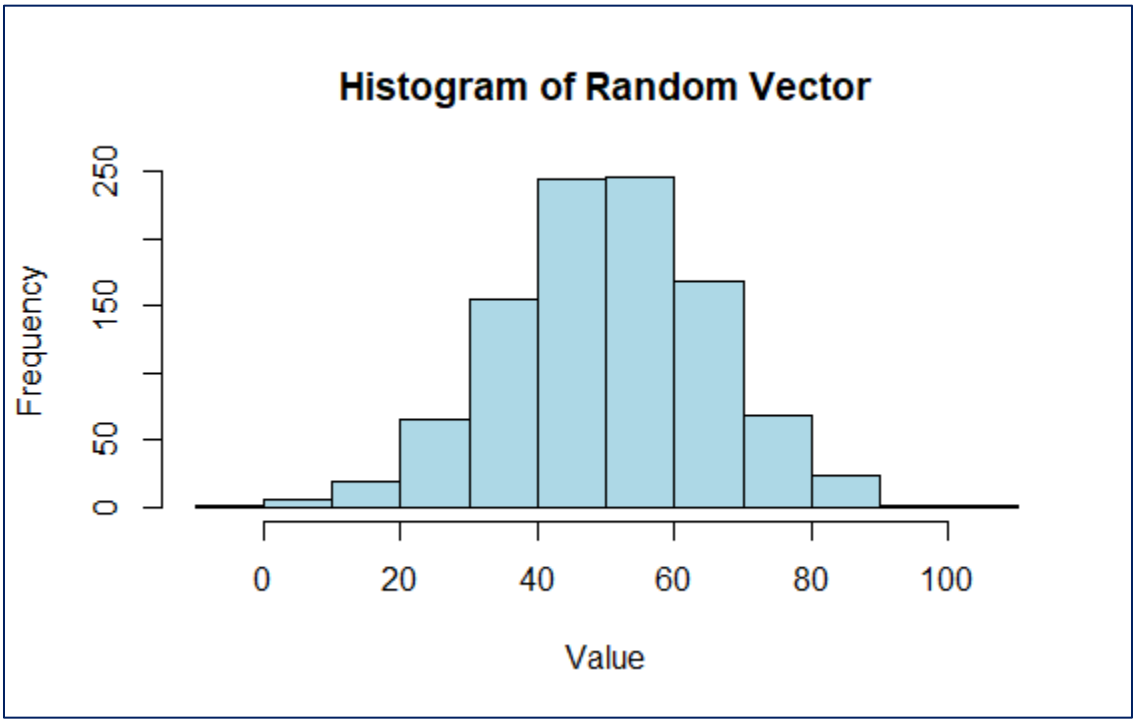
37.

```
> set.seed(5) # # Set a random seed for reproducibility
> # Generate a vector of 1000 random numbers from a normal distribution
> # with a mean of 50 and a standard deviation of 15
> random_vector <- rnorm(n=1000, mean = 50, sd = 15)
> print(random_vector)
   [1]  37.387168  70.765390  31.167622  51.052141  75.671613  40.956380  42.917504
   [8]  40.469430  45.713395  52.071623  68.414455  37.973308  33.794111  47.636985
  [15]  33.923599  47.915208  41.040304  17.240499  53.612259  46.109669  63.507679
  [22]  64.128041  72.019429  60.601416  62.285134  45.597772  71.278836  72.481607
  [29]  40.143769  37.208068  54.738726  66.645413  83.231909  68.256555  72.188327
  [36]  64.273607  34.857010  19.992909  23.567212  47.860878  73.250906  37.963652
  [43]  48.881316  78.435019  43.151466  58.433350  36.694872  43.096331  39.135073
  [50]  48.961833  71.948728  52.815891  65.330343  41.122478  48.316990  36.125704
  [57]  61.299572  48.310864  49.038636  53.499129  32.951258  62.822456  41.324444
  [64]  57.445423  38.599131  44.879206  18.465063  45.474466  30.914248  45.805008
  [71]  46.938540  46.615787  55.205427  50.485518  56.202969  47.669773  64.602281
  [78]  51.816352  52.837605  41.556724  57.476242  23.865463  64.632936  49.638757
  [85]  60.135267  39.345356  85.808490  42.898520  48.863412  42.172399  63.890707
  [92]  34.063832  58.355508  63.510959  64.849185  55.754121  44.801243  41.897161
  [99]  47.261666  49.110505  20.069195  67.029669  60.136918  53.127249  49.132315
 [106]  63.407171  46.567019  20.515210  38.697343  69.202274  35.706426  74.335691
 [113]  89.002130  52.094728  29.739205  61.983965  26.675062  56.955801  50.786443
 [120]  46.969523  67.562846  63.272673  30.231671  25.351236  65.888756  54.351254
 [127]  43.999498  68.646437  29.503842  28.378800  70.228236  20.322075  31.385741
 [134]  48.439413  60.994594  56.835194  54.321193  33.894636  59.731138  54.487434
 [141]  38.060075  49.559699  82.703536  64.361277  45.424270  43.723950  51.499311
 [148]  46.552856  28.771777  44.111017  64.191328  61.276563  42.239347  62.125040
 [155]  40.781972  68.573884  44.928573  67.945495  43.350224  52.791723  10.679828
 [162]  83.693819  51.401475  74.409201  42.336237  40.109287  49.397148  48.219590
 [169]  49.705147  42.714823  28.397787  52.156533  31.481200  23.712482  49.467556
 [176]  54.980524  73.584324  33.957941  63.744298  41.075107  82.724700  39.743401
 [183]  61.250888  64.615740  31.032898  45.838679  47.159020  44.239626  61.108820
 [190]  32.474924  60.013080  55.493554  42.275855  56.758524  47.184194  70.086041
 [197]  62.243288  51.233026  40.237059  60.896135  48.294827  45.573488  64.837527
```

Above snapshot continues ……

Above Code sets a seed for random number generation to ensure that the results can be reproduced. It then creates a vector called random_vector that contains 1000 random numbers drawn from a normal distribution with a specified mean of 50 and a standard deviation of 15. This means most of the generated numbers will cluster around 50, with some variability due to the standard deviation.

38. Use the hist function and provide it with random_vector:



From the above histogram, we can analyze that:

- The data is centered around 50.
- The distribution is symmetrical.
- Data follows a normal distribution pattern.

This can help in making predictions about future data points, as we know most values will likely fall near the mean of 50, and extreme deviations are rare.

Read csv :

```
pacman::p_load(tidyverse)  # Load the tidyverse libraries
# Read the CSV file into a data frame
first_dataframe <- read_csv("ds_salaries.csv")
print(first_dataframe)
```

Output in first_dataframe:

```
first_dataframe   607 obs. of 12 variables
    $ ...1              : num [1:607] 0 1 2 3 4 5 6 7 8 9 ...
    $ work_year         : num [1:607] 2020 2020 2020 2020 2020 20…
    $ experience_level  : chr [1:607] "MI" "SE" "SE" "MI" ...
    $ employment_type   : chr [1:607] "FT" "FT" "FT" "FT" ...
    $ job_title         : chr [1:607] "Data Scientist" "Machine L…
    $ salary            : num [1:607] 70000 260000 85000 20000 15…
    $ salary_currency   : chr [1:607] "EUR" "USD" "GBP" "USD" ...
```

Learned how to read csv using read_csv() function which is provided by tidyverse library. It made it easy to look store data into a variable and to make decisions.

head(first_dataframe):

```
> # Displays the first six rows of first_dataframe
> head(first_dataframe)
# A tibble: 6 × 12
   ...1 work_year experience_level employment_type job_title          salary salary_currency
  <dbl>     <dbl> <chr>            <chr>           <chr>               <dbl> <chr>
1     0      2020 MI               FT              Data Scientist      70000 EUR
2     1      2020 SE               FT              Machine Learnin…   260000 USD
3     2      2020 SE               FT              Big Data Engine…    85000 GBP
4     3      2020 MI               FT              Product Data An…    20000 USD
5     4      2020 SE               FT              Machine Learnin…   150000 USD
6     5      2020 EN               FT              Data Analyst        72000 USD
# i 5 more variables: salary_in_usd <dbl>, employee_residence <chr>, remote_ratio <dbl>,
#   company_location <chr>, company_size <chr>
```

Display first six rows of data to provide a quick overview of its content.

head(first_dataframe, n = 7):

```
> # Shows the first seven rows of first_dataframe
> head(first_dataframe, n = 7)
# A tibble: 7 × 12
   ...1 work_year experience_level employment_type job_title        salary salary_currency
  <dbl>     <dbl> <chr>            <chr>           <chr>             <dbl> <chr>
1     0      2020 MI               FT              Data Scientist    70000 EUR
2     1      2020 SE               FT              Machine Learnin… 260000 USD
3     2      2020 SE               FT              Big Data Engine…  85000 GBP
4     3      2020 MI               FT              Product Data An…  20000 USD
5     4      2020 SE               FT              Machine Learnin… 150000 USD
6     5      2020 EN               FT              Data Analyst      72000 USD
7     6      2020 SE               FT              Lead Data Scien… 190000 USD
# i 5 more variables: salary_in_usd <dbl>, employee_residence <chr>, remote_ratio <dbl>,
#   company_location <chr>, company_size <chr>
```

Shows the first seven rows of `first_dataframe`, allowing for a more extended preview.

names(first_dataframe):

```
> # Returns the column names
> names(first_dataframe)
 [1] "...1"              "work_year"         "experience_level"  "employment_type"
 [5] "job_title"         "salary"            "salary_currency"   "salary_in_usd"
 [9] "employee_residence" "remote_ratio"     "company_location"  "company_size"
>
```

Returns the column names of `first_dataframe`, showing the structure of the dataframe.

smaller_dataframe <- select(first_dataframe, job_title, salary_in_usd):

```
> # Returns only the job_title and salary_in_usd columns from first_dataframe
> smaller_dataframe <- select(first_dataframe, job_title, salary_in_usd)
> # Returns only the job_title and salary_in_usd columns from first_dataframe
> smaller_dataframe <- select(first_dataframe, job_title, salary_in_usd)
> smaller_dataframe
# A tibble: 607 x 2
   job_title                  salary_in_usd
   <chr>                              <dbl>
 1 Data Scientist                     79833
 2 Machine Learning Scientist        260000
 3 Big Data Engineer                 109024
 4 Product Data Analyst               20000
 5 Machine Learning Engineer         150000
 6 Data Analyst                       72000
 7 Lead Data Scientist               190000
 8 Data Scientist                     35735
 9 Business Data Analyst             135000
10 Lead Data Engineer                125000
# i 597 more rows
# i Use `print(n = ...)` to see more rows
> |
```

better_smaller_dataframe <- arrange(smaller_dataframe, desc(salary_in_usd)):

```
> #Sorts smaller_dataframe in descending order based on the salary_in_usd column and stores t
he result in better_smaller_dataframe.
> better_smaller_dataframe <- arrange(smaller_dataframe, desc(salary_in_usd))
> better_smaller_dataframe
# A tibble: 607 x 2
   job_title                          salary_in_usd
   <chr>                                      <dbl>
 1 Principal Data Engineer                   600000
 2 Research Scientist                        450000
 3 Financial Data Analyst                    450000
 4 Applied Machine Learning Scientist        423000
 5 Principal Data Scientist                  416000
 6 Data Scientist                            412000
 7 Data Analytics Lead                       405000
 8 Applied Data Scientist                    380000
 9 Director of Data Science                  325000
10 Data Engineer                             324000
# i 597 more rows
# i Use `print(n = ...)` to see more rows
```

better_smaller_dataframe <- filter(smaller_dataframe, salary_in_usd > 80000):

```
> # Filters smaller_dataframe to show only rows where salary_in_usd is greater than 80,000 an
d stores the result in better_smaller_dataframe.
> better_smaller_dataframe <- filter(smaller_dataframe, salary_in_usd >
+                                      80000)
> better_smaller_dataframe
# A tibble: 384 x 2
   job_title                  salary_in_usd
   <chr>                              <dbl>
 1 Machine Learning Scientist        260000
 2 Big Data Engineer                 109024
 3 Machine Learning Engineer         150000
 4 Lead Data Scientist               190000
 5 Business Data Analyst             135000
 6 Lead Data Engineer                125000
 7 Lead Data Analyst                  87000
 8 Data Analyst                       85000
 9 Big Data Engineer                 114047
10 BI Data Analyst                    98000
# i 374 more rows
# i Use `print(n = ...)` to see more rows
>
```

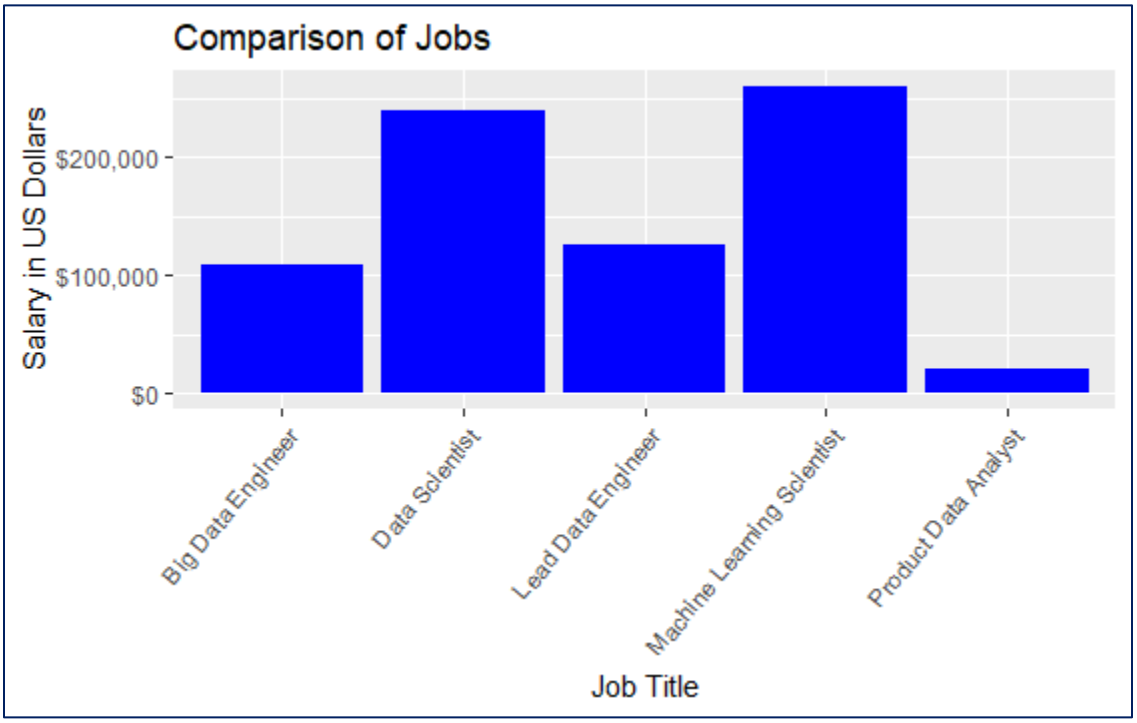
better_smaller_dataframe <- mutate(smaller_dataframe, salary_in_euros = salary_in_usd * .94):

```
> #Adds a new column salary_in_euros to smaller_dataframe, converting the salary_in_usd value
s to euros by multiplying by 0.94.
> better_smaller_dataframe <-
+   mutate(smaller_dataframe, salary_in_euros = salary_in_usd * .94)
> better_smaller_dataframe
# A tibble: 607 x 3
   job_title                  salary_in_usd salary_in_euros
   <chr>                              <dbl>           <dbl>
 1 Data Scientist                     79833           75043.
 2 Machine Learning Scientist        260000          244400
 3 Big Data Engineer                 109024          102483.
 4 Product Data Analyst               20000           18800
 5 Machine Learning Engineer         150000          141000
 6 Data Analyst                       72000           67680
 7 Lead Data Scientist               190000          178600
 8 Data Scientist                     35735           33591.
 9 Business Data Analyst             135000          126900
10 Lead Data Engineer                125000          117500
# i 597 more rows
# i Use `print(n = ...)` to see more rows
>
```

better_smaller_dataframe <- slice(smaller_dataframe, 1, 1, 2, 3, 4, 10, 1):

```
> #Selects specific rows (1, 1, 2, 3, 4, 10, and 1) from smaller_dataframe, allowing for dupl
ication of row indices, and stores the result in better_smaller_dataframe.
> better_smaller_dataframe <- slice(smaller_dataframe, 1, 1, 2, 3, 4, 10,
+                                   1)
> better_smaller_dataframe
# A tibble: 7 x 2
  job_title                  salary_in_usd
  <chr>                              <dbl>
1 Data Scientist                     79833
2 Data Scientist                     79833
3 Machine Learning Scientist        260000
4 Big Data Engineer                 109024
5 Product Data Analyst               20000
6 Lead Data Engineer                125000
7 Data Scientist                     79833
> |
```

ggplot(better_smaller_dataframe) + geom_col(mapping = aes(x = job_title, y = salary_in_usd), fill = "blue") + xlab("Job Title") + ylab("Salary in US Dollars") + labs(title = "Comparison of Jobs ") + scale_y_continuous(labels = scales::dollar) + theme(axis.text.x = element_text(angle = 50, hjust = 1)):



Key concepts learned while using these code snippets involve basic data manipulation and visualization in R, which are essential for data analysis. Functions like head() and names() help preview and understand the structure of a dataset. select(), arrange(), filter(), and mutate() demonstrate powerful

ways to filter, sort, and modify data frames for targeted analysis. Finally, ggplot() is used to create visual representations of data, helping to communicate insights more effectively. These operations are useful for cleaning, exploring, and summarizing data in various real-world analytics scenarios.

Conclusion:

Throughout this project, I have learned how to effectively use various functions and attributes to manipulate data and achieve the desired results. While it took some time to grasp the basics as a beginner, this exercise has been a valuable learning experience, deepening my understanding of data manipulation and analysis.

Bibliography

Bluman, A. G. (2018). Elementary statistics: a step by step approach. McGraw-Hill Education.

W3 School for R Tutorials, https://www.w3schools.com/R/