(USDI): Merge, Join and Preparing Data for Plots

Abiola Oyebanjo

July 4-6 , 2024

Contents

---

1 Import Libraries Dataframes

```
library(dplyr)
library(tidyverse)
library(reshape2) # for melt
```

2 Set Working Directory

3 Joining Dataframes

3.1 Joining dataframes

Joining dataframes is a common operation when working with relational data. It allows us to combine information from different sources based on common variables.

3.1.1 Creating Sample Dataframes

Let's start by creating two simple dataframes:

```
df1 <- data.frame(
  ID = 1:5,
  Name = c("Alice", "Bob", "Charlie", "David", "Eve")
)

df2 <- data.frame(
  ID = 2:6,
  Score = c(85, 92, 78, 95, 88)
)

print(df1)
```

```
##   ID    Name
## 1  1   Alice
## 2  2     Bob
## 3  3 Charlie
## 4  4   David
## 5  5     Eve
```

```
print(df2)
```

```
##   ID Score
## 1  2    85
## 2  3    92
## 3  4    78
## 4  5    95
## 5  6    88
```

Here, we've created two dataframes:

`df1` contains student IDs and names `df2` contains student IDs and scores

Notice that the IDs don't perfectly align between the two dataframes. This is intentional to demonstrate different join behaviors.

3.2 Using merge() from Base R

The `merge()` function in base R is versatile and can perform various types of joins.

3.3 Left Join

```
left_merge <- merge(df1, df2, by = "ID", all.x = TRUE)
print(left_merge)
```

```
##   ID    Name Score
## 1  1   Alice    NA
## 2  2     Bob    85
## 3  3 Charlie    92
## 4  4   David    78
## 5  5     Eve    95
```

Explanation:

`by = "ID"` specifies that we're joining on the "ID" column

`all.x = TRUE` means we keep all rows from the left dataframe (df1), even if there's no match in df2

This results in NA values for scores where there's no match in df2

3.4 Right Join

```
##   ID    Name Score
## 1  2     Bob    85
## 2  3 Charlie    92
## 3  4   David    78
## 4  5     Eve    95
## 5  6    <NA>    88
```

Explanation:

`all.y` = TRUE means we keep all rows from the right dataframe (df2), even if there's no match in df1 This results in NA values for names where there's no match in df1

3.5 Inner Join

```
##   ID    Name Score
## 1  2     Bob    85
## 2  3 Charlie    92
## 3  4   David    78
## 4  5     Eve    95
```

Explanation:

Without `all.x` or `all.y`, merge() performs an inner join by default This keeps only the rows where there's a match in both dataframes

3.6 Full Outer Join

```
##   ID    Name Score
## 1  1   Alice    NA
## 2  2     Bob    85
## 3  3 Charlie    92
## 4  4   David    78
## 5  5     Eve    95
```

```
## 6  6    <NA>    88
```

Explanation:

`all = TRUE` keeps all rows from both dataframes, filling in NA where there's no match This is useful when you want to see all data from both sources, regardless of matches

4 Using dplyr for Joins

The dplyr package provides more intuitive join functions that are often preferred in modern R programming.

4.1 Left Join

```
left_join_dplyr <- left_join(df1, df2, by = "ID")
print(left_join_dplyr)
```

```
##   ID    Name Score
## 1  1   Alice    NA
## 2  2     Bob    85
## 3  3 Charlie    92
## 4  4   David    78
## 5  5     Eve    95
```

Explanation:

`left_join()` keeps all rows from df1 and adds matching data from df2 It's equivalent to the left merge we did earlier, but with a more readable syntax

4.2 Right Join

```
left_join_dplyr <- left_join(df1, df2, by = "ID")
print(left_join_dplyr)
```

```
##   ID    Name Score
## 1  1   Alice    NA
## 2  2     Bob    85
## 3  3 Charlie    92
## 4  4   David    78
## 5  5     Eve    95
```

Explanation:

`right_join()` keeps all rows from df2 and adds matching data from df1 It's equivalent to the right merge we did earlier

4.3 Inner Join

```
inner_join_dplyr <- inner_join(df1, df2, by = "ID")
print(inner_join_dplyr)
```

```
##   ID    Name Score
## 1  2     Bob    85
## 2  3 Charlie    92
## 3  4   David    78
## 4  5     Eve    95
```

Explanation:

`inner_join()` keeps only rows with matches in both dataframes It's equivalent to the inner merge we did earlier

4.4 Full Join

```
full_join_dplyr <- full_join(df1, df2, by = "ID")
print(full_join_dplyr)
```

```
##   ID    Name Score
## 1  1   Alice    NA
## 2  2     Bob    85
## 3  3 Charlie    92
## 4  4   David    78
## 5  5     Eve    95
## 6  6    <NA>    88
```

Explanation:

`full_join()` keeps all rows from both dataframes, filling in NA where there's no match It's equivalent to the full outer merge we did earlier

5 Reshaping Data (for Visualization and Panel Data Analysis)

Reshaping data involves changing the structure of a dataset without changing the information it contains. The two main forms are "wide" and "long" formats. This is very helpful for visualization purposes and Panel (Longitudinal) data analysis. Longitudinal data involves repeated observations of the same variables over time for the same subjects. This type of data allows for the analysis of changes over time and the study of temporal dynamics within the data.

5.1 Wide to Long Format

In wide format, each subject's responses are in a single row. In long format, each row is a single subject-variable combination.

Let's create a wide format dataframe:

```
wide_df <- data.frame(
  ID = 1:2,
  Math = c(35, 32, 48, 44),
  English = c(92, 88, 95, 89),
  Science = c(49, 85, 40,55),
  Year =c(2023, 2023, 2024,2024)
)

print(wide_df)
```

```
##   ID Math English Science Year
## 1  1   35      92      49 2023
## 2  2   32      88      85 2023
## 3  1   48      95      40 2024
## 4  2   44      89      55 2024
```

5.1.1 Using melt() from reshape2

```
##    ID Subject Score
```

```
## 1   1    Math    35
## 2   2    Math    32
## 3   1    Math    48
## 4   2    Math    44
## 5   1 English    92
## 6   2 English    88
## 7   1 English    95
## 8   2 English    89
## 9   1 Science    49
## 10  2 Science    85
## 11  1 Science    40
## 12  2 Science    55
## 13  1    Year  2023
## 14  2    Year  2023
## 15  1    Year  2024
## 16  2    Year  2024
```

Explanation:

`melt()` is similar to gather() but from an older package `id.vars` specifies which columns to keep as is `variable.name` and `value.name` specify names for the new columns

5.1.2 Reshape the data to long format, excluding the Year column

```
## # A tibble: 12 x 4
##       ID  Year Subject Score
##    <int> <dbl> <chr>   <dbl>
##  1     1  2023 Math       35
##  2     1  2023 English    92
##  3     1  2023 Science    49
##  4     2  2023 Math       32
##  5     2  2023 English    88
##  6     2  2023 Science    85
##  7     1  2024 Math       48
##  8     1  2024 English    95
##  9     1  2024 Science    40
## 10     2  2024 Math       44
## 11     2  2024 English    89
## 12     2  2024 Science    55
```

5.1.3 Using gather() from tidyr

```
long_df_gather <- wide_df %>%
  gather(key = "Subject", value = "Score", -c(ID,Year))

print(long_df_gather)
```

```
##    ID Year Subject Score
## 1   1 2023    Math    35
## 2   2 2023    Math    32
## 3   1 2024    Math    48
## 4   2 2024    Math    44
## 5   1 2023 English    92
## 6   2 2023 English    88
## 7   1 2024 English    95
```

```
## 8   2 2024 English    89
## 9   1 2023 Science    49
## 10  2 2023 Science    85
## 11  1 2024 Science    40
## 12  2 2024 Science    55
```

Explanation:

`gather()` takes all columns except ID and creates two new columns:

`Subject`: contains the original column names (Math, English, Science)

`Score`: contains the values from those columns

5.2 Using pivot_longer from tidyverse (preferred)

For this course, we will be using `pivot_longer` from the `tidyverse` package frequently. The reason is that, like other similar functions discussed above, `pivot_longer` allows us to transform data from a wide format to a long format, which is essential for many types of data analysis and visualization tasks. More on in the next section:

6 Preparing Data for Visualization

`pivot_longer`: This function is particularly useful because it converts multiple columns into key-value pairs, creating a single column for the variable names and another for the values. This is crucial when we need a single column for the x or y axis in our plots. In wide format data, having multiple columns for what we want to plot can complicate the visualization process.

`summarize`: We use this function to calculate summary statistics such as average percentages for each subject across all rows.

`starts_with`: This function helps us select columns that start with a specific string, which is useful when dealing with data that has multiple similarly named columns, like percentages in this example.

```r
# Transform from wide to long using pivot_longer
long_df_pivot <- pivot_longer(wide_df,
            cols = -c(ID, Year), # Columns to pivot (excluding ID and Year)
            names_to = "Subject", # New column for variable names
            values_to = "Score")%>%  # New column for values
  mutate(ID = as.factor(ID))  # Convert ID to factor


# Print the resulting long data frame
print(long_df_pivot)
```

```
## # A tibble: 12 x 4
##    ID     Year Subject Score
##    <fct> <dbl> <chr>   <dbl>
##  1 1      2023 Math       35
##  2 1      2023 English    92
##  3 1      2023 Science    49
##  4 2      2023 Math       32
##  5 2      2023 English    88
##  6 2      2023 Science    85
##  7 1      2024 Math       48
##  8 1      2024 English    95
##  9 1      2024 Science    40
```

```
## 10 2      2024 Math       44
## 11 2      2024 English    89
## 12 2      2024 Science    55
```

6.0.1 Preparing data for plotting

Long format and summarizing the data are important steps to plotting. Let plots the average scores for each course using ggplot by following the steps below:

6.1 Step 1: Use mutate to calculate percentages in a new vector

```
data_percent <- wide_df %>%
  mutate(total = Math + English + Science) %>%
  mutate(percent_Math = Math / total * 100,
         percent_English = English / total * 100,
         percent_Science = Science / total * 100)
print(data_percent)
```

```
##   ID Math English Science Year total percent_Math percent_English
## 1  1   35      92      49 2023   176     19.88636        52.27273
## 2  2   32      88      85 2023   205     15.60976        42.92683
## 3  1   48      95      40 2024   183     26.22951        51.91257
## 4  2   44      89      55 2024   188     23.40426        47.34043
##   percent_Science
## 1        27.84091
## 2        41.46341
## 3        21.85792
## 4        29.25532
```

6.2 Step 2: Calculate average/mean percentages per course

```
## Calculate average percentages
avg_percent <- data_percent %>%
  summarise(avg_Math = mean(percent_Math),
            avg_English = mean(percent_English),
            avg_Science = mean(percent_Science))

avg_percent
```

avg_Math	avg_English	avg_Science
21.28247	48.61314	30.10439

6.3 Step 3: use starts_with (in pivot_longer) to pivot data to long

```
data_long <- avg_percent %>%
  pivot_longer(cols = starts_with("avg_"),
               names_to = "Subject",
               values_to = "Percentage")
print(data_long)
```

```
## # A tibble: 3 x 2
##   Subject     Percentage
```

```
##   <chr>              <dbl>
## 1 avg_Math            21.3
## 2 avg_English         48.6
## 3 avg_Science         30.1
```

6.4 Step 4: Rename columns

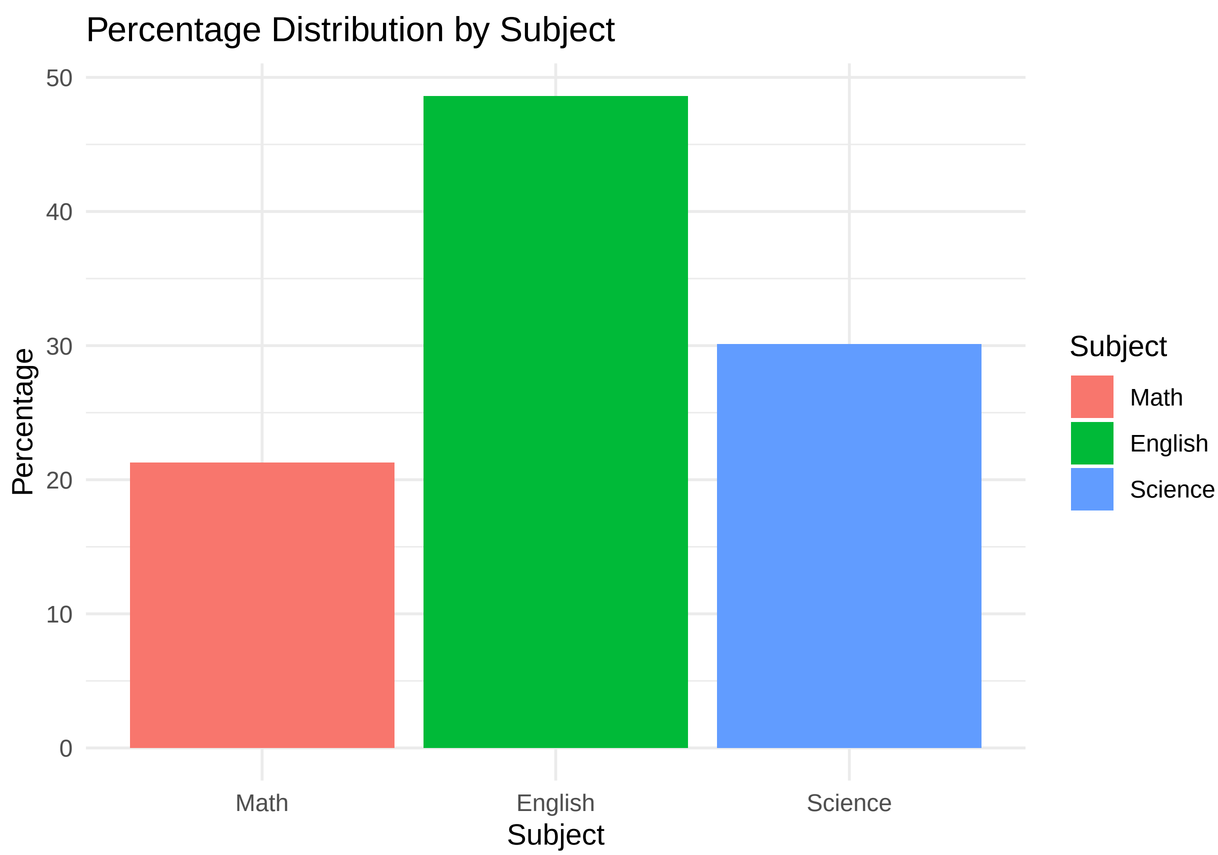
```
avg_percent <- data_long %>%
  mutate(Subject = factor(Subject, levels = c("avg_Math", "avg_English", "avg_Science"),
                          labels = c("Math", "English", "Science")))

print(avg_percent)
```

```
## # A tibble: 3 x 2
##   Subject Percentage
##   <fct>        <dbl>
## 1 Math          21.3
## 2 English       48.6
## 3 Science       30.1
```

6.5 Step 5: Plot the data

```
# Plotting using ggplot2
ggplot(avg_percent, aes(x = Subject, y = Percentage, fill = Subject)) +
  geom_bar(stat = "identity") +
  labs(title = "Percentage Distribution by Subject",
       x = "Subject", y = "Percentage") +
  theme_minimal()
```

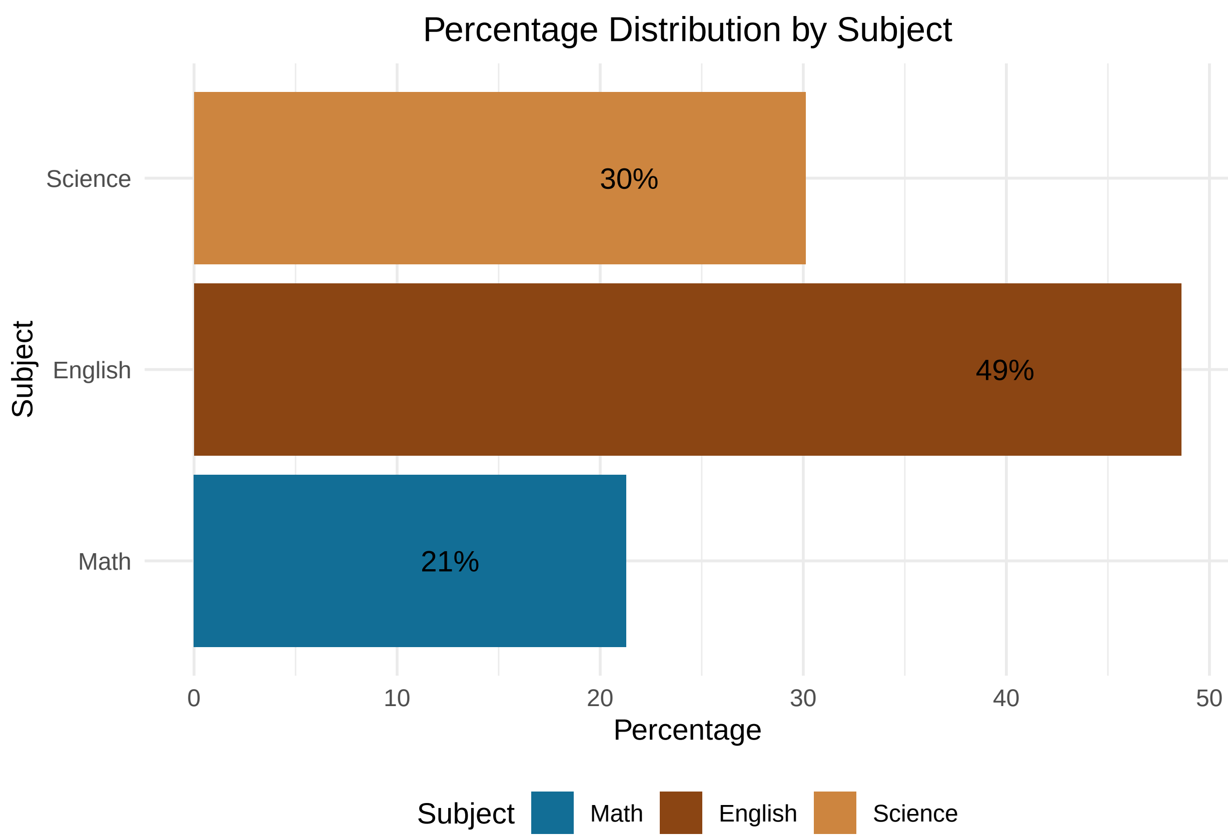


6.6 Step 6: Plotting with customizations

```
# Define custom color palette
custom_colors <- c("#126e96", "#8B4513", "#CD853F")

# Plotting with customizations
plot <- ggplot(avg_percent, aes(x = Subject, y = Percentage, fill = Subject)) +
  geom_bar(stat = "identity") +
  scale_fill_manual(values = custom_colors) +
  labs(title = "Percentage Distribution by Subject",
       x = "Subject", y = "Percentage") +
  theme_minimal() +
  theme(plot.title = element_text(hjust = 0.5), legend.position = "bottom") +
  ylab("Percentage") +
  geom_text(aes(label = paste0(round(Percentage), "%")), hjust = 3.5)+
  coord_flip()

# Print the plot
print(plot)
```



6.7 Step 7: Plotting with customizations with geom_point and faceting

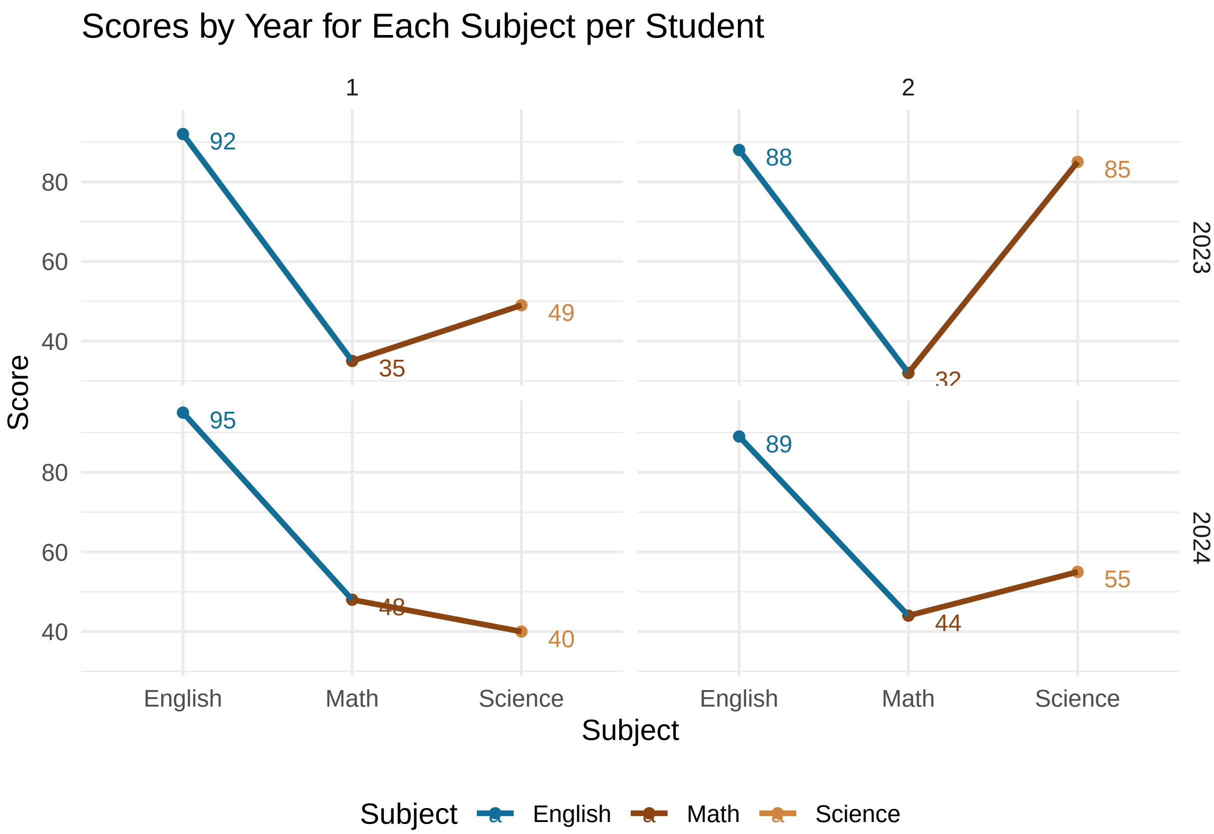
6.7.1 Scores by Year for Each Subject per Student

```
# Ensure the Year are treated as factors
long_df_pivot$Year<- as.factor(long_df_pivot$Year)

# Plotting scores by Year for Each Subject with points, lines, and y-axis labels
plot_points <- ggplot(long_df_pivot, aes(x = Subject, y = Score, color = Subject)) +
  geom_point(linewidth = 3) +
  geom_line(aes(group = ID), linewidth = 1) +  # Add a line connecting points for each ID
  geom_text(aes(label = Score), vjust=0.9, hjust = -1.0, size = 3) +  # Add y-axis labels
  scale_color_manual(values = custom_colors) +
  labs(title = "Scores by Year for Each Subject per Student",
       x = "Subject", y = "Score") +
  theme_minimal() +
  theme(plot.title = element_text(vjust = 1.5), legend.position = "bottom") +
  facet_grid(Year ~ ID, scales = "free_x")  # Facet by Year
```

```
## Warning in geom_point(linewidth = 3): Ignoring unknown parameters: `linewidth`
```

```
# Print the plot
print(plot_points)
```



7 Creating plot for the entire database

7.1 Using the wide database

Creating plots using the wide format data can be beneficial when you want to plot multiple data points rather than summary statistics (mean, mode etc).

In a wide format, each column typically represents a different variable or measurement, making it easier to plot relationships between these variables directly.

This approach is often more straightforward for scatter plots and other visualizations that involve pairwise comparisons or multiple variables.

7.2 Using the facet_grid

8 Explanation:

Using the long format data can also allow for more flexibility in visualizing data by group. The long format typically involves having one column for the variable names and another for the values, which makes it easier to apply faceting.

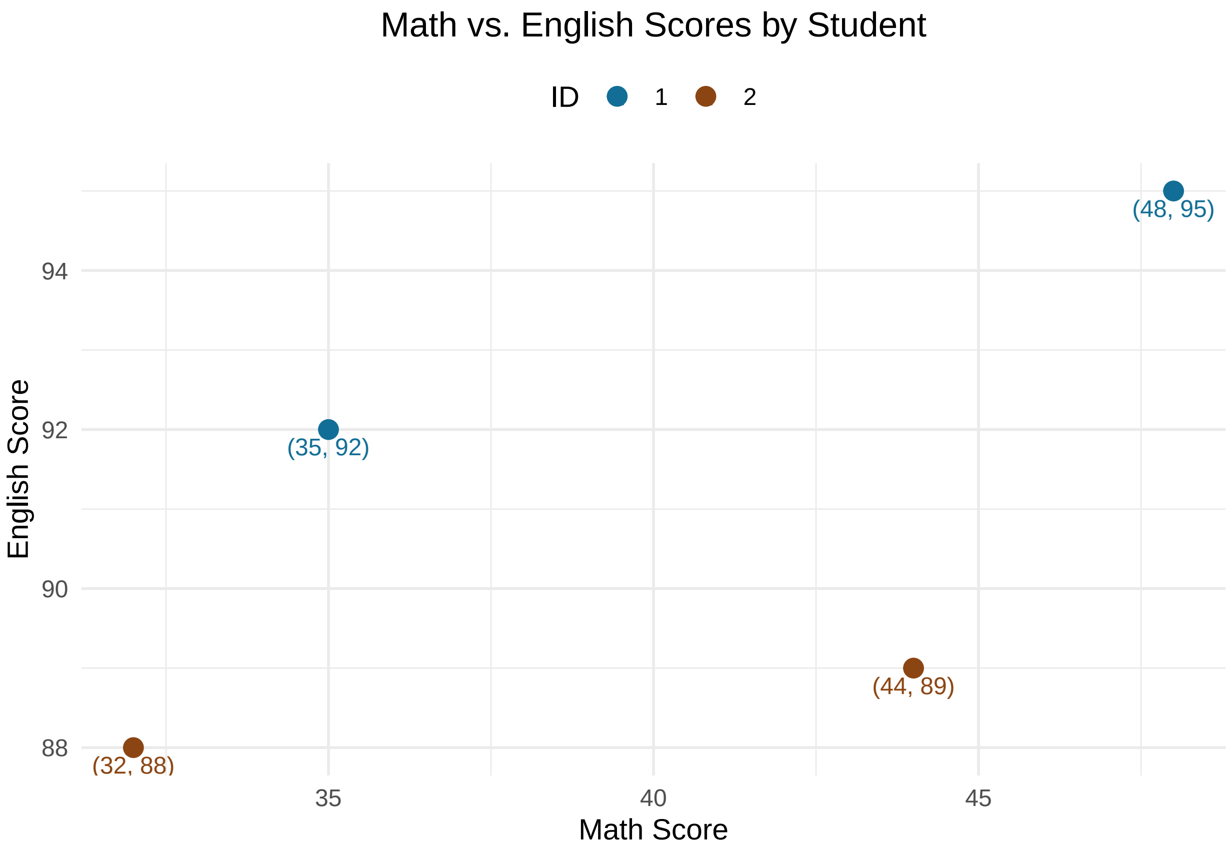
Faceting, such as with `facet_grid` or `facet_wrap`, allows you to create multiple plots based on the levels of one or more grouping variables. This is particularly useful for comparing subgroups within your data, as each subgroup gets its own individual plot.

```
# Ensure the IDs are treated as factors
wide_df$ID <- as.factor(wide_df$ID)

# Plotting scores by student with geom_point and geom_text
```

```r
plot_points1 <- ggplot(wide_df, aes(x = Math, y = English, color = ID)) +
  geom_point(size = 3) +
  geom_text(aes(label = paste0("(", Math, ", ", English, ")")), vjust = 1.5, size = 3) +
  scale_color_manual(values = custom_colors) +
  labs(title = "Math vs. English Scores by Student",
       x = "Math Score", y = "English Score") +
  theme_minimal() +
  theme(plot.title = element_text(hjust = 0.5), legend.position = "top")

# Print the plot
print(plot_points1)
```



9 Long to Wide Format

Now let's convert our long format data back to wide format.

9.1 Using spread() from tidyr

```
##   ID Year English Math Science
## 1  1 2023      92   35      49
## 2  1 2024      95   48      40
## 3  2 2023      88   32      85
## 4  2 2024      89   44      55
```

Explanation:

`spread()` is the opposite of `gather()` It takes the `Subject` column and spreads it out into separate columns

The values in these new columns come from the `Score` column