Lecture 4, MTH 400, Fall 2024

Comparing Data, Faceting/Grouping

In R, there are several methods to filter data based on specific conditions or criteria. Here are some commonly used approaches:

1. Base R Subsetting: In base R, you can use subsetting techniques to filter data based on logical conditions. The subset operator [ ] or the subset() function can be used for this purpose. For example, to filter a data frame df for rows where a specific variable x meets a condition, you can use:

```
filtered_data <- df[df$x > 5, ]
```

2. dplyr Package: The dplyr package, part of the tidyverse, provides a set of functions for efficient data manipulation. It offers the filter() function to filter rows based on specific conditions. Here's an example using dplyr:

   ```
   library(dplyr)
   filtered_data <- filter(df, x > 5)
   ```

3. data.table Package: The data.table package provides fast and memory-efficient data manipulation capabilities. It offers the [ ] operator and the subset() function similar to base R. For example:

   ```
   library(data.table)
   setDT(df)  # Convert data.frame to data.table
   filtered_data <- df[x > 5]
   ```

4. sqldf Package: The sqldf package allows you to write SQL queries to filter data frames. It can be useful if you are familiar with SQL syntax. Here's an example:

   ```
   library(sqldf)
   filtered_data <- sqldf("SELECT * FROM df WHERE x > 5")
   ```

5. filter() Function from the stats package: The stats package in R provides the filter() function, which can be used to extract subsets of data based on logical conditions. Here's an example:

   ```
   filtered_data <- filter(df, x > 5)
   ```

These are just a few examples of how you can filter data in R. Depending on your specific needs and preferences, there may be other packages or methods that can be applied. It's recommended to refer to the documentation and examples provided by each package for more detailed usage instructions.

In R, you can use pipelines to create a sequence of data transformation steps, making your code more readable, concise, and organized. Pipelines allow you to chain multiple operations together, passing the output of one operation as the input to the next. Here's how you can use pipelines in R for data cleaning and processing:

1. dplyr Package: The dplyr package provides a convenient way to create data pipelines using the %>% (pipe) operator. This operator takes the output of the previous step and passes it as the first

argument to the next step. You can use functions like mutate(), filter(), select(), and arrange() to perform various data manipulation tasks. Here's an example:

```
library(dplyr)

cleaned_data <- raw_data %>%
  mutate(new_variable = old_variable * 2) %>%
  filter(condition) %>%
  select(variable1, variable2) %>%
  arrange(variable1)
```

2. magrittr Package: The magrittr package provides the %>% operator, similar to dplyr. It allows you to create pipelines for data processing. Here's an example:

```
library(magrittr)

cleaned_data <- raw_data %>%
  mutate(new_variable = old_variable * 2) %>%
  filter(condition) %>%
  select(variable1, variable2) %>%
  arrange(variable1)
```

3. data.table Package: The data.table package also supports pipelines using the %>% operator. You can perform data manipulation operations using functions like [, :=], [, .()], and [, ..()]. Here's an example:

```
library(data.table)
library(magrittr)

setDT(raw_data)

cleaned_data <- raw_data %>%
  .[, new_variable := old_variable * 2] %>%
  .[condition] %>%
  .[, .(variable1, variable2)] %>%
  .[order(variable1)]
```

4. pipeR Package: The pipeR package provides a pipe operator %>>% that can be used to create data pipelines. It allows you to chain operations together. Here's an example:

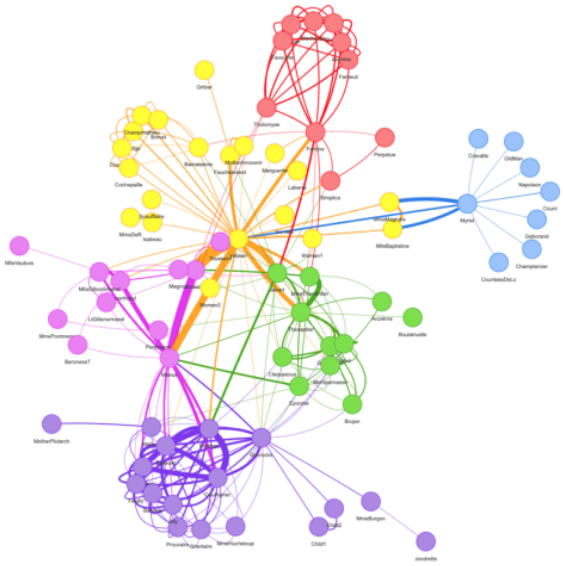
```
library(pipeR)

cleaned_data <- raw_data %>>%
  mutate(new_variable = old_variable * 2) %>>%
  filter(condition) %>>%
  select(variable1, variable2) %>>%
  arrange(variable1)
```

These are just a few examples of how you can use pipelines in R for data cleaning and processing. The choice of package and operator depends on your personal preference and the specific packages you are using. Pipelines provide a concise and readable way to perform a series of data transformations, making your code more efficient and easier to understand.

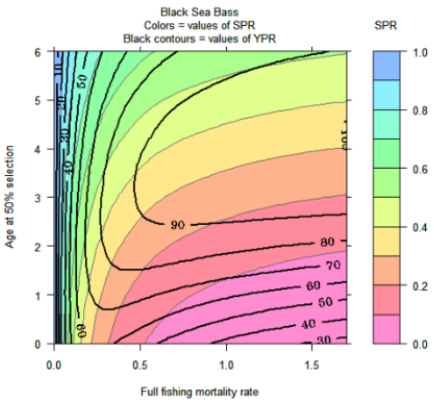
Advanced data types refer to data structures or formats that go beyond the basic primitive types (such as integers, floating-point numbers, strings, and booleans) commonly used in programming languages. These advanced data types are designed to represent more complex and specialized data structures and are often used in specific domains or applications. Here are some examples of advanced data types:

1. Arrays: Arrays are a collection of elements of the same type, arranged in a contiguous block of memory. They allow for efficient storage and retrieval of multiple values. Arrays can be one-dimensional (vectors), two-dimensional (matrices), or multi-dimensional.
2. Lists: Lists are a collection of elements that can be of different types. Unlike arrays, lists are not constrained to a fixed size, and elements can be added or removed dynamically. Lists provide flexibility in managing and manipulating heterogeneous data.
3. Data Frames: Data frames are two-dimensional tabular data structures where columns can contain different types of data. They are commonly used to store and manipulate structured data, such as spreadsheets or database tables. Data frames are widely used in statistical analysis and data manipulation in languages like R and Python (with Pandas).
4. Time Series: Time series data represents observations or measurements taken at regular intervals over time. It is commonly used in fields such as finance, economics, weather forecasting, and signal processing. Time series data types often include timestamps or time indices associated with each data point.
5. Geospatial Data Types: Geospatial data types represent geographic features and their attributes. They are used to store and analyze spatial data, such as points, lines, polygons, and raster images. Geospatial data types include geometries, topologies, and coordinate reference systems, and are commonly used in Geographic Information Systems (GIS).
6. Networks and Graphs: Network data types represent relationships or connections between entities. They are used to model and analyze complex systems, such as social networks, transportation networks, or computer networks. Network data types include nodes (vertices) and edges (links) that define the structure and connectivity of the graph.



7. Text and Text Mining Data Types: Text data types represent textual information, such as documents, articles, or social media posts. Text data types can include plain text, tokenized text, or structured representations like bag-of-words or term frequency-inverse document frequency (TF-IDF) matrices. They are commonly used in natural language processing (NLP) and text mining tasks.
8. Multimedia Data Types: Multimedia data types represent various forms of multimedia content, including images, audio, video, and other multimedia formats. They are used in fields such as computer vision, image processing, speech recognition, and multimedia analytics.

These are just a few examples of advanced data types. The choice of data type depends on the specific needs of the application and the characteristics of the data being processed. Advanced data types allow for more expressive representations and enable efficient handling and analysis of complex and specialized data structures.



Faceting in ggplot2 is a powerful technique to create multiple plots based on one or more categorical variables, allowing you to compare subsets of your data easily. Faceting divides the data into subsets and creates a separate plot for each subset, arranging them into a grid. There are two main functions used for faceting in ggplot2: facet_wrap() and facet_grid().

facet_wrap():
facet_wrap() arranges the plots in a wrapping manner, typically in a single direction (either horizontally or vertically) but can be adjusted to fit into a grid. It is useful when you have a single faceting variable.

facet_wrap(~ variable, nrow = NULL, ncol = NULL, scales = "fixed")

- variable: The variable to facet by.
- nrow: Number of rows.
- ncol: Number of columns.
- scales: Determines if the scales are fixed ("fixed") or free ("free", "free_x", "free_y").

Example:

```
library(ggplot2)
# Example dataset
data(mpg)
# Basic ggplot
p <- ggplot(mpg, aes(x = displ, y = hwy)) +
```

```
  geom_point()
# Faceting by 'class'
p + facet_wrap(~ class)
```

facet_grid()

facet_grid() arranges the plots into a grid of rows and columns, defined by one or two faceting variables. It is particularly useful when you want to compare data across two variables.

Syntax:

```
facet_grid(rows ~ cols, scales = "fixed")
```

- rows: The variable to facet by rows.
- cols: The variable to facet by columns.
- scales: Determines if the scales are fixed ("fixed") or free ("free", "free_x", "free_y").

Example:

```
library(ggplot2)
# Example dataset
data(mpg)
# Basic ggplot
p <- ggplot(mpg, aes(x = displ, y = hwy)) +
  geom_point()
# Faceting by 'drv' (rows) and 'cyl' (columns)
p + facet_grid(drv ~ cyl)
```

Customizing Facets

You can further customize facets using additional parameters:

- labeller: Customizes the facet labels.
- switch: Moves the facet labels to the opposite side.
- as.table: Controls the order of the facets (by default, facets are arranged like a table).

Customizing Example:

```
# Custom labeller
p + facet_wrap(~ class, labeller = label_both)
```

Practical Considerations

- Scales: By default, facets share the same scales. Use scales = "free" to allow each facet to have its own scale.
- Layout: The layout of facets can be controlled by specifying nrow and ncol in facet_wrap(), or by arranging the variables in facet_grid().
- Theme Adjustments: Facets can be further customized using theme elements like strip.text, strip.background, etc.

Example with Custom Theme:

```
p + facet_wrap(~ class) +
  theme(strip.text = element_text(size = 12, face = "bold"),
        strip.background = element_rect(fill = "lightblue"))
```

Faceting is a powerful way to compare different subsets of your data in ggplot2. By using facet_wrap() for single variables and facet_grid() for combinations of variables, you can create comprehensive and visually appealing multi-plot layouts. Remember to leverage customization options to enhance the readability and aesthetics of your faceted plots.

Plotting data with groups in R involves creating visualizations that distinguish between different categories or groups within the data. Here are some common types of grouped plots using ggplot2:

1. Grouped Scatter Plot

A scatter plot that distinguishes groups using color or shape.

Example:

```
library(ggplot2)

# Example dataset
data(mpg)

# Scatter plot with groups by manufacturer
ggplot(mpg, aes(x = displ, y = hwy, color = manufacturer)) +
  geom_point() +
  theme_minimal() +
  labs(title = "Scatter Plot with Groups",
       x = "Displacement",
       y = "Highway Miles per Gallon")
```

2. Grouped Line Plot

A line plot that shows trends for different groups.

Example:

```
# Example dataset
data(economics_long)

# Line plot with groups by variable
ggplot(economics_long, aes(x = date, y = value01, color = variable)) +
  geom_line() +
  theme_minimal() +
  labs(title = "Line Plot with Groups",
       x = "Date",
       y = "Normalized Value")
```

3. Grouped Bar Plot

A bar plot that shows the distribution of different groups.

Example:

```
# Example dataset
data(diamonds)

# Grouped bar plot by cut
```

```
ggplot(diamonds, aes(x = cut, fill = clarity)) +
  geom_bar(position = "dodge") +
  theme_minimal() +
  labs(title = "Grouped Bar Plot",
       x = "Cut",
       y = "Count")
```

4. Grouped Box Plot
A box plot that shows the distribution of a continuous variable for different groups.

Example:

```
# Grouped box plot by cut
ggplot(diamonds, aes(x = cut, y = price, fill = cut)) +
  geom_boxplot() +
  theme_minimal() +
  labs(title = "Grouped Box Plot",
       x = "Cut",
       y = "Price")
```

5. Grouped Density Plot
A density plot that shows the distribution of a continuous variable for different groups.

Example:

```
# Grouped density plot by cut
ggplot(diamonds, aes(x = price, fill = cut)) +
  geom_density(alpha = 0.5) +
  theme_minimal() +
  labs(title = "Grouped Density Plot",
       x = "Price",
       y = "Density")
```

6. Grouped Violin Plot
A violin plot that combines a box plot and a density plot to show the distribution of a continuous variable for different groups.

Example:

```
# Grouped violin plot by cut
ggplot(diamonds, aes(x = cut, y = price, fill = cut)) +
  geom_violin() +
  theme_minimal() +
  labs(title = "Grouped Violin Plot",
       x = "Cut",
       y = "Price")
```

7. Grouped Facet Plot
Using faceting to create a grid of plots for different groups.

Example:

```
# Scatter plot with facets by cut
ggplot(diamonds, aes(x = carat, y = price)) +
  geom_point() +
  facet_wrap(~ cut) +
  theme_minimal() +
  labs(title = "Facet Plot by Cut",
       x = "Carat",
       y = "Price")
```

Grouping data in plots helps to visualize comparisons and differences between categories within the data. The ggplot2 package provides various functions to create grouped plots, such as using color, fill, facet_wrap(), and facet_grid(). These examples showcase how to effectively use these techniques to gain insights from grouped data.

To create network graphs of relationships in R, you can use the igraph package. igraph provides a wide range of functions and capabilities for analyzing and visualizing networks. Here's an overview of the steps involved:

1. Install and load the igraph package:

```
install.packages("igraph")
library(igraph)
```

2. Prepare your data:
   - Network data is typically represented as an edge list or an adjacency matrix. The edge list contains pairs of nodes that are connected, and the adjacency matrix represents the connections between nodes.
   - If you have an edge list, you can create a graph object using the graph_from_edgelist() function. If you have an adjacency matrix, you can use the graph_from_adjacency_matrix() function.
   - Optionally, you can also assign attributes to nodes or edges using the V() and E() functions.
3. Customize the graph appearance:
   - igraph provides various options to customize the appearance of the graph. You can modify attributes such as node color, size, shape, label, and edge width.
   - Use functions like set_vertex_attr(), set_edge_attr(), and V()$attribute_name to modify node and edge attributes.
4. Visualize the network graph:
   - Use the plot() function to visualize the network graph. By default, it will create a basic plot, but you can customize it further.
   - igraph provides different layout algorithms to position the nodes, such as Fruchterman-Reingold (layout_with_fr()), Kamada-Kawai (layout_with_kk()), or circular (layout_as_circle()).
   - You can also use functions like vertex.color, vertex.size, vertex.label, and edge.width to control the visual attributes of nodes and edges.

Here's a simplified example that demonstrates the basic process:

```
# Install and load the igraph package
install.packages("igraph")
```

```
library(igraph)

# Create an edge list
edges <- data.frame(
  from = c("A", "A", "B", "C", "D"),
  to = c("B", "C", "D", "E", "E")
)

# Create a graph object
graph <- graph_from_data_frame(edges)

# Customize the graph appearance
V(graph)$color <- "lightblue"  # Node color
V(graph)$size <- 25  # Node size
E(graph)$width <- 2  # Edge width

# Visualize the network graph
plot(graph, layout = layout_with_fr)
```

This is a basic example, but igraph provides many advanced features for network analysis, such as community detection, centrality measures, and path finding. You can explore the igraph documentation for more details on these advanced functionalities and customization options.

Resources:

1. https://flowingdata.com/2012/05/15/how-to-visualize-and-compare-distributions/
2. https://r4ds.had.co.nz/data-visualisation.html
3. https://www.r-bloggers.com/2019/06/interactive-network-visualization-with-r/
4. https://builtin.com/data-science/grouping-r
5. https://dplyr.tidyverse.org/reference/group_by.hhttps://builtin.com/data-science/grouping-rml
6. https://www.sfu.ca/~mjbrydon/tutorials/BAinR/filter.html
7. https://cran.r-project.org/web/packages/crunch/vignettes/filters.html
8. https://www.listendata.com/2023/08/how-to-filter-dataframe-in-r-with.html
9. https://blog.revolutionanalytics.com/2009/01/r-graph-gallery.html