Introduction to Data Science

Topic : Roles in Data Science Project(Unit-1)

What is data science and how it is different from others?

- **Data Science**: Encompasses the entire process of data analysis, including data collection, cleaning, analysis, visualization, and applying machine learning.
- **Data Science** is a multidisciplinary field that combines various techniques, tools, and methodologies to extract insights and knowledge from data.
- Data science integrates elements from statistics, computer science, and domain expertise to turn data into actionable insights.

Tools and Technologies

Data science often involves using various tools and technologies such as:

- **Programming Languages:** Python, R
- **Data Analysis Tools:** Pandas, NumPy
- **Machine Learning Libraries:** Scikit-learn, TensorFlow, Keras
- **Data Visualization Tools:** Matplotlib, Seaborn, Tableau
- **Big Data Technologies:** Hadoop, Spark

Roles in Data Science Project:

Problem Statement: The bank is losing money due to bad loans and wants to reduce losses by better identifying risky loans

1. Sponsor Role

Definition:

The sponsor is the person responsible for the overall success of the

project. In this case, it's ideally the bank's head of Consumer Lending. They **set the goals, and evaluate the project's success.**

Key Points:

1. **Role**:
    - The sponsor represents the **business interests** and decides if the project is a **success or failur**e.
    - They ensure the project aligns with the bank's strategic objectives.
2. **Responsibilities**:
    - **Setting Goals**: Define clear, measurable goals for the project. ○ **Decision Making**: Make key decisions about the project's direction and priorities.
    - **Evaluation**: Assess the final outcomes to determine if the project meets the defined goals.
3. **Involvement:**
    - **Regular Updates**: The sponsor needs to be regularly updated on the project's progress.
    - **Understanding Terms**: Information should be communicated in terms that are easy for them to understand.
    - **Feedback**: Provide feedback based on the progress and any intermediate successes or failures.
4. **Example of a Goal in loan application project:**
    - **Quantitative Goal**: "Identify 90% of accounts that will go into default((When a borrower fails to make payments as agreed in the loan contract) at least two months before the first missed payment((The prediction should be made at least two months before the borrower actually misses their first payment) with a false positive rate of no more than 25%."(When the model incorrectly predicts that an account will default, but it does not.Out of every 100 accounts predicted to default, no more

than 25 should be incorrect predictions)
    - This goal is clear and measurable, making it easier to assess whether the project is successful.

5. **Communication**:
    - **Clear and Regular**: Regular meetings and updates are crucial to keep the sponsor informed.
    - **Progress Report**s: Share plans, progress, and results in a way that the sponsor can easily understand.
6. **Central Focus**:
    - Getting sponsor sign-off is the central organizing goal of the project. The project is successful if the sponsor is satisfied with the outcomes.

In Summary:

The sponsor is the key decision-maker who ensures the project aligns with the bank's goals. They set clear, measurable objectives and evaluate the project's success based on these criteria. Keeping the sponsor informed and involved throughout the project is critical to its success. If the sponsor, such as the head of Consumer Lending, is happy with the project's outcome, then the project is considered a success.

2. Client Role

Definition:

The client is the person who **represents the end users of the model**—in this case, the loan officers who will use the tool to identify risky loans. They make sure the model fits into the daily workflow of the loan officers and meets their needs.

Key Points:

1. **Role**:

- The client ensures that the model is practical and useful for loan officers.
- They understand the business processes(knows the steps involved in processing a loan application, such as credit checks, income verification, and risk assessment. And criteria used to evaluate loan applications and the timelines for processing.) and act as the domain expert.(They know which data points (e.g., credit score, loan amount, repayment history) are crucial for assessing loan risk.)

2. **Responsibilities:**
   - **Hands-On Involvement:** Unlike the sponsor who oversees the project's success, the client is more involved in the day-to-day details.
   - **Interface**: They bridge the gap between technical model building and real-world application.
   - **Feedback:** They provide insights into how the model can be integrated into current workflows and ensure it meets the practical needs of loan officers.
3. **Characteristics**:
   - **Business Expertise**: They know the business processes but may not have deep mathematical or statistical knowledge.
   - **Regular Interaction**: Regular meetings with the data science team to ensure alignment with user needs.
   - **Representative of End Users:** They advocate for the loan officers who will use the model.

Example:

In the loan application project:

1. **Who Could Be the Client?**
   - A senior loan officer or a representative of the loan officers.
2. **Role in the Project:**

- Initial Setup: Explains how loan officers currently assess loan applications.
- Feedback Loop: Provides feedback on the model's interface, usability, and results.
- Testing and Validation: Works with the data science team to test the model in real-world scenarios and suggests improvements.

3. **Communication:**
   - Clear and Focused Meetings: Regular, focused meetings where the data science team presents progress in understandable terms.
4. **Outcome:**
   - Ensures the model fits smoothly into the loan officers' daily routines and effectively helps them identify risky loans.

In Summary:

The client ensures that the model is practical, usable, and meets the needs of the loan officers. By keeping the client informed and involved, the data science team ensures the model is not only technically sound but also effective in real-world applications. If the end users (loan officers) find the model helpful and easy to use, the project is considered a success in the long run.

3. Data Scientist Role

Definition:

The data scientist is the person **who designs and executes the technical aspects of the projec**t. They are responsible for making sure the project succeeds by using their expertise in data, statistics, and machine learning.

Key Points:

1. **Responsibilities:**

- **Project Strategy**: Decide the overall approach and plan for the project.
- **Client Communication**: Keep the client (e.g., loan officers) informed and involved throughout the project.
- **Data and Tools**: Choose the right data sources and tools for analysis and modeling.

2. **Technical Tasks**:
   - **Data Analysis**: Examine the data to understand its characteristics and quality.
   - **Statistical Tests**: Perform tests to analyze data trends and patterns.
   - **Machine Learning Models:** Apply machine learning techniques to build predictive models.
   - **Evaluation:** Assess the model's performance to ensure it meets the project's goals.

3. Example in Loan Application Project:

Steps Involved:

1. **Project Strategy**:
   - **Plan:** The data scientist outlines the steps needed to build a tool that predicts risky loans.
   - **Approach**: They decide to use historical loan data to train a predictive model.

2. Client Communication:

- **Meetings**: Regularly update the loan officers (clients) on the progress and gather their feedback.
- **Usability:** Ensure the tool will be easy for loan officers to use in their daily work.

3. **Data and Tools**:
   - **Data Sources**: Select relevant data, such as borrower credit scores, payment history, and economic indicators.

- **Tools**: Choose appropriate software and machine learning libraries (e.g., Python, scikit-learn).

4. **Technical Execution:**
   - **Data Analysis**: Clean and prepare the data, identifying key features that might indicate a loan default.
   - **Statistical Tests**: Check for trends and correlations in the data.
   - **Machine Learning Models:** Train a model to predict loan defaults based on the data.
   - **Evaluation**: Test the model's accuracy and adjust as needed to meet the goal (e.g., identifying 90% of defaults with a false positive rate of no more than 25%).
5. **Outcome:**
   - **Model Deployment**: The final predictive model is integrated into the loan officers' workflow, helping them identify risky loans early and reduce losses.

In Summary:

The data scientist is the technical expert who designs the project plan, analyzes data, applies machine learning models, and evaluates results. They ensure the project stays on track and meets the defined goals, keeping the client informed and involved throughout the process. In the loan application project, the data scientist's work helps the bank develop a tool to predict and manage risky loans more effectively.

4. Data Architect Role

Definition:

The data architect is the person responsible for managing and organizing all the data used in the project. They ensure that the data is stored securely and is accessible when needed. This role is often filled by someone who specializes in managing databases.

Key Points:

1. **Responsibilities**:
    - **Data Management**: Ensure that all the data required for the project is properly stored and organized.
    - **Data Storage**: Design and maintain the databases where the data is stored.
    - **Consultatio**n: Provide quick advice and support on data-related issues.
2. Role in the Loan Application Project:

Steps Involved:

1. **Data Collection:**
    - The data architect ensures that all relevant data, such as borrower information, loan details, and payment histories, is collected and stored in a central location.
2. **Data Storage**:
    - **Databases:** They set up and manage databases where this data is stored securely and efficiently.
    - **Data Warehouses:** They may use data warehouses to store large amounts of historical data for analysis.
3. **Data Access**:
    - **Accessibilit**y: They make sure that the data scientists and other team members can easily access the data they need for analysis and modeling.
    - **Security:** They ensure that the data is protected and only accessible to authorized personnel.
4. **Consultation**:
    - **Support:** The data architect provides support to the data science team, answering questions about data storage and helping resolve any data-related issues.
    - **Quick Consultations**: Since they often manage data for multiple projects, their involvement may be limited to quick

consultations rather than ongoing participation.

Example in Loan Application Project:

1. **Data Collection**:
    - The data architect collects data on loan applications, borrower credit scores, repayment histories, and other relevant information from various sources.
2. **Data Storage:**
    - Database Setup: They set up a database to store this information securely.
    - Data Warehouse: They may use a data warehouse to store large volumes of historical data for easy access and analysis by the data science team.
3. **Data Acces**s:
    - Access Provision: They ensure that the data scientists working on the project can access the stored data quickly and efficiently.
    - Security Measures: They implement security measures to protect sensitive borrower information.
4. **Consultation:**
    - Data Queries: When the data science team needs specific data or faces issues accessing certain data, the data architect provides quick and effective solutions.

In Summary:

The data architect ensures that all the data needed for the loan application project is properly stored, organized, and accessible. They manage the databases and data warehouses, providing support and consultation to the data science team as needed. In the loan application project, the data architect's role is crucial for ensuring that the data is available and secure, enabling the data scientists to build accurate and effective predictive models.

5.Operations Role

- The operations role is critical both in acquiring data and delivering the final results. The person filling this role usually has operational responsibilities outside of the data science group.
- This person will likely have constraints on response time, programming language, or data size that you need to respect in deployment
- The operations role is crucial for both acquiring the necessary data and ensuring that the final predictive model is implemented smoothly within the bank's existing systems.
- This person helps manage technical constraints and ensures the model is practical and usable for loan officers.
- In the loan application project, the operations role ensures that the predictive model is effectively integrated into the bank's workflow, helping to identify risky loans and reduce losses.

Introduction to Data Science

Topic : Stages in Datascience(Unit-1)

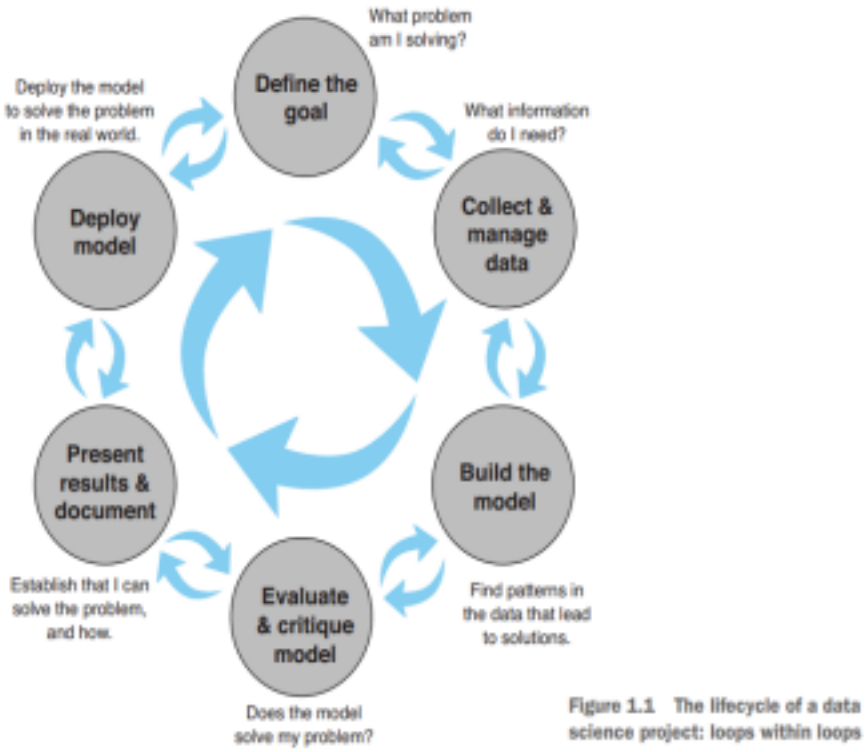


Figure 1.1 The lifecycle of a data science project: loops within loops

- The ideal data science environment is one that encourages feedback and iteration between the data scientist and all other stakeholders.

  This is reflected in the lifecycle of a data science project.

1. Defining the goal

The first task in a data science project is to define a measurable and quantifiable goal.

At this stage, learn all that you can about the context of your project:

- Why do the sponsors want the project in the first place?
- What do they lack, and what do they need?

- What are they doing to solve the problem now, and why isn't that good enough?
- What resources will you need: what kind of data and how much staff?
- Will you have domain experts to collaborate with, and what are the computational resources?
- How do the project sponsors plan to deploy your results? ● What are the constraints that have to be met for successful deployment?

Defining Project Goals in the Loan Application Example

- **Ultimate Business Goal:**
  - Reduce the bank's losses due to bad loans.
- **Sponsor's Vision:**
  - Create a tool to help loan officers accurately score loan applicants.
  - Ensure loan officers retain final discretion on loan approvals.
- **Defining the Precise Goal:**
  - **Specific and Measurable Goal:**
    - Not: "We want to get better at finding bad loans."
    - Instead: "We want to reduce our rate of loan charge-offs by at least 10%, using a model that predicts which loan applicants are likely to default."
  - **Concrete Goals:**
    - Lead to clear stopping conditions and acceptance criteria.
    - Prevent the project from becoming unbounded or never-ending.
- **Avoiding Vague Goals:**
  - Vague goals can lead to an unbounded project with no clear end or satisfactory outcome.
- **Need for Exploratory Projects:**
  - Some projects may be more exploratory:

■ Examples: "Is there something in the data that correlates to higher defaults?" or "Which types of loans should we consider reducing?"

2. Data collection and management

This step encompasses identifying the data you need, exploring it, and conditioning it to be suitable for analysis.

This stage is often the most time-consuming step in the process. It's also one of the most important:

- What data is available to me?
- Will it help me solve the problem?
- Is it enough?
- Is the data quality good enough?

Imagine that, for your loan application problem, you've collected a sample of representative loans from the last decade. Some of the loans have defaulted; most of them (about 70%) have not. You've collected a variety of attributes about each loan application, as listed in table 1.2.

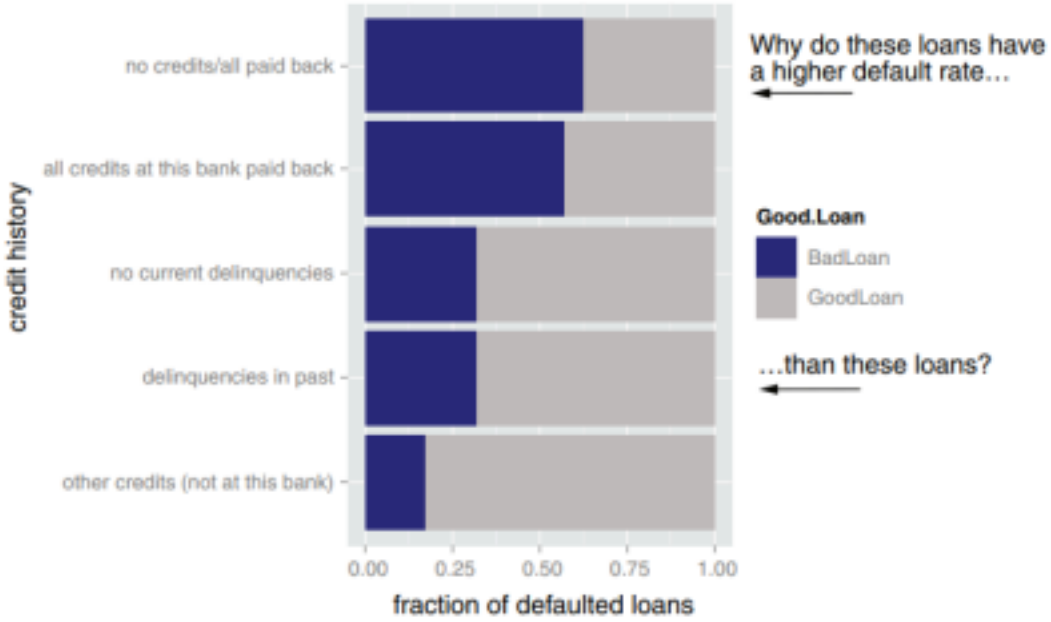
Table 1.2 Loan data attributes

`Status_of_existing_checking_account` *(at time of application)*
`Duration_in_month` *(loan length)*
`Credit_history`
`Purpose` *(car loan, student loan, and so on)*
`Credit_amount` *(loan amount)*
`Savings_Account_or_bonds` *(balance/amount)*
`Present_employment_since`
`Installment_rate_in_percentage_of_disposable_income`
`Personal_status_and_sex`
`Cosigners`
`Present_residence_since`
`Collateral` *(car, property, and so on)*
`Age_in_years`
`Other_installment_plans` *(other loans/lines of credit—the type)*
`Housing` *(own, rent, and so on)*
`Number_of_existing_credits_at_this_bank`
`Job` *(employment type)*
`Number_of_dependents`
`Telephone` *(do they have one)*
`Loan_status` *(dependent variable)*

- In your data, Loan_status takes on two possible values:

GoodLoan and BadLoan. For the purposes of this discussion, assume that a GoodLoan was paid off, and a BadLoan defaulted

- This is the stage where you initially explore and visualize your data.
- You'll also clean the data: repair data errors and transform variables, as needed.
- In the process of exploring and cleaning the data, you may discover that it isn't suitable for your problem, or that you need other types of information as well.
- You may discover things in the data that raise issues more important than the one you originally planned to address. For example, the data in figure 1.2 seems counterintuitive.



Visual Representation of Bias (Figure 1.2):

- **Credit History vs. Fraction of Defaulted Loans:**
  - No credits/all paid back: Higher default rate.
  - All credits at this bank paid back: Higher default rate.
  - No current delinquencies: Moderate default rate.
  - Delinquencies in past: Lower default rate.
  - Other credits (not at this bank): Lower default rate.

This indicates a need to reassess the variables used in the model and the

sample of data being analyzed.

Bias in the Data:

- Realization that the sample is biased.
- The data only includes loans that were actually accepted.
- An unbiased sample should include both accepted and rejected loan applications.
- Fewer risky-looking loans than safe-looking ones are present in the data due to stricter vetting for risky loans.

Implications of the Bias:

- If the model is used downstream of the current application approval process, credit history may no longer be a useful variable. ● Even seemingly safe loan applications should be more carefully scrutinized.

Adjusting Project Goals:

- Discoveries may lead to changes or refinements in project goals. ● Consider focusing on seemingly safe loan applications for more careful scrutiny.

Iterative Process:

- Common to cycle between data exploration, goal definition, and modeling stages.
- Continue discovering new insights in the data and refining the approach accordingly.

3. Modeling Stage

- **Objective:** Extract useful insights from data to achieve the project's goals using statistics and machine learning.
- **Interplay with Data Cleaning:**
    - Many modeling procedures require specific assumptions about data distribution and relationships.

- There may be back-and-forth between modeling and data cleaning to find the best data representation.

- **Common Data Science Modeling Tasks:**
  1. **Classifying:** Determine if something belongs to one category or another.

     Example: Spam Email Detection

     - **Task:** Decide if an email belongs to the "spam" category or the "non-spam" category.
     - **Application:** Email service providers use classification algorithms to filter out spam emails from users' inboxes.

  2. Scoring: Predicting or estimating a numeric value, such as a price or probability

     Example: Credit Scoring

     - **Task:** Predict a numeric value that represents the creditworthiness of an individual, such as a credit score.
     - **Application:** Banks and financial institutions use credit scoring models to determine the likelihood that a loan applicant will repay a loan.

  3. Ranking : —Learning to order items by preferences

     Example: Search Engine Results

     - **Task:** Learn to order web pages by their relevance to a given search query.
     - **Application:** Search engines like Google rank web pages so that the most relevant results appear at the top of the search results.

  4. Clustering : Grouping items into most-similar groups

Example: Customer Segmentation

- **Task:** Group customers into clusters based on similar purchasing behavior.
- **Application:** Retail companies use clustering to create targeted marketing campaigns for different customer segments.

5. **Finding Relations : Finding correlations or potential causes of effects seen in the data**

Example: Market Basket Analysis

- **Task:** Find correlations between products that are frequently bought together.
- **Application:** Retailers use market basket analysis to optimize product placement and cross-selling strategies in stores.

6. **Characterizing: Very general plotting and report generation from data**

Example: Sales Report Generation

- **Task:** Generate general plots and reports to describe the sales performance over a period.
- **Application:** Businesses use characterization to create sales reports that summarize total sales, trends, and other key performance indicators for management review.

Loan Application Problem:

- **Type:** Classification problem.
- **Goal:** Identify loan applicants who are likely to default.
- **Common Approaches:** Logistic regression, tree-based methods

- **User Requirements:**
  - Loan officers and other end-users need to understand the reasoning behind the model's classification.
  - They want an indication of the model's confidence in its

decisions (e.g., whether an applicant is highly likely to default or only somewhat likely).

Key Points:

- **Model Selection:** Choose appropriate methods based on the type of problem (e.g., classification).
- **User Interaction:** Ensure the model's decisions are interpretable and provide confidence levels for practical use.
- **Iterative Process:** Continue refining the model and data as needed to meet these goals.

```
library('rpart')
load('GCData.RData')
model <- rpart(Good.Loan ~
  Duration.in.month +
  Installment.rate.in.percentage.of.disposable.income +
  Credit.amount +
  Other.installment.plans,
  data=d,
  control=rpart.control(maxdepth=4),  method="class")
```

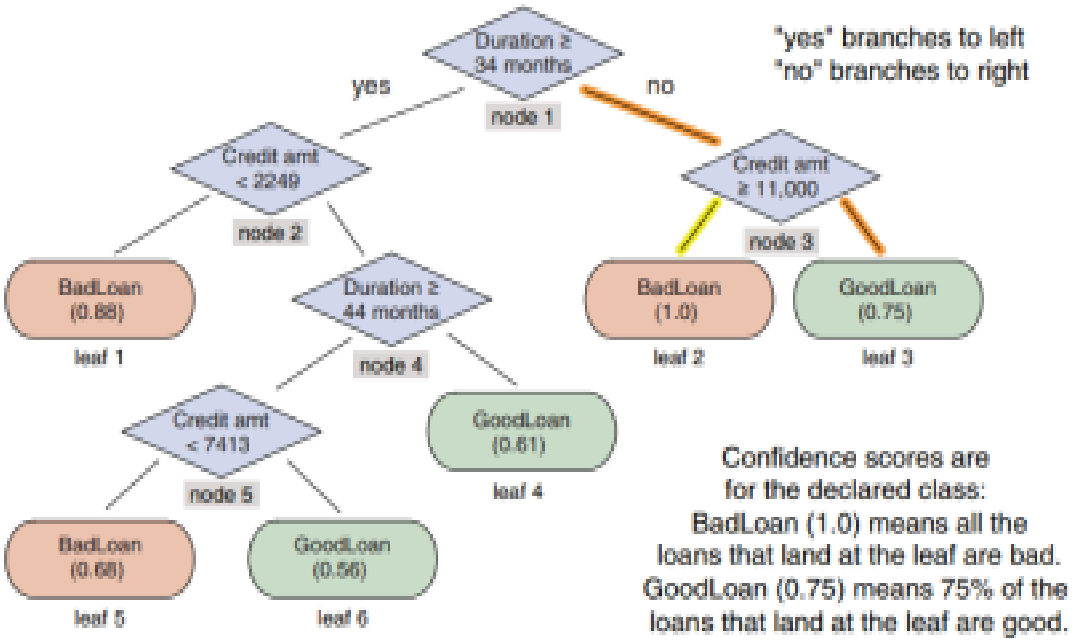


Figure 1.3 A decision tree model for finding bad loan applications. The outcome nodes show confidence scores.

Let's suppose that there is an application for a one-year loan of DM 10,000 (deutsche mark, the currency at the time of the study). At the top of the tree (node 1 in figure 1.3), the model checks if the loan is for longer than 34 months. The answer is "no," so the model takes the right branch down the tree. This is shown as the highlighted branch from node 1. The next question (node 3) is whether the loan is for more than DM 11,000. Again, the answer is "no," so the model branches right (as shown by the darker highlighted branch from node 3) and arrives at leaf 3.

Historically, 75% of loans that arrive at this leaf are good loans, so the model recommends that you approve this loan, as there is a high probability that it will be paid off.

On the other hand, suppose that there is an application for a

one-year loan of DM 15,000. In this case, the model would first branch right at node 1, and then left at node 3, to arrive at leaf 2. Historically, all loans that arrive at leaf 2 have defaulted, so the model recommends that you reject this loan application.

4. Model evaluation and critique

Once you have a model, you need to determine if it meets your goals:

- Is it accurate enough for your needs? Does it generalize well?
- Does it perform better than “the obvious guess”? Better than whatever estimate you currently use?
- Do the results of the model (coefficients, clusters, rules) make sense in the context of the problem domain?

A good summary of classifier accuracy is the confusion matrix, which tabulates actual classifications against predicted ones.

The model predicted loan status correctly 73% of the times better than chance (50%). In the original dataset, 30% of the loans were bad, so guessing GoodLoan all the time would be 70% accurate (though not very useful). So you know that the model does better than random and somewhat better than obvious guessing.

Overall accuracy is not enough. You want to know what kinds of mistakes are being made. Is the model missing too many bad loans, or is it marking too many good loans as bad?

Recall measures how many of the bad loans the model can actually find.

Precision measures how many of the loans identified as bad really are bad.

False positive rate measures how many of the good loans are mistakenly identified as bad.

Ideally, you want the recall and the precision to be high, and the false positive rate to be low.

5. Presentation and documentation

- Present your results to your project sponsor and other stakeholders.
- Document the model for those in the organization who are responsible for using, running, and maintaining the model once it has been deployed.
- **Focus on Business Metrics:** Emphasize how the model impacts key metrics like chargeoffs. For example, if the model identifies bad loans that constitute 22% of the total money lost to defaults, highlight the potential reduction in bank losses.
- **Key Findings**: Share notable insights or recommendations, such as the increased risk associated with new car loans compared to used car loans or the concentration of losses in bad car loans and equipment loans.

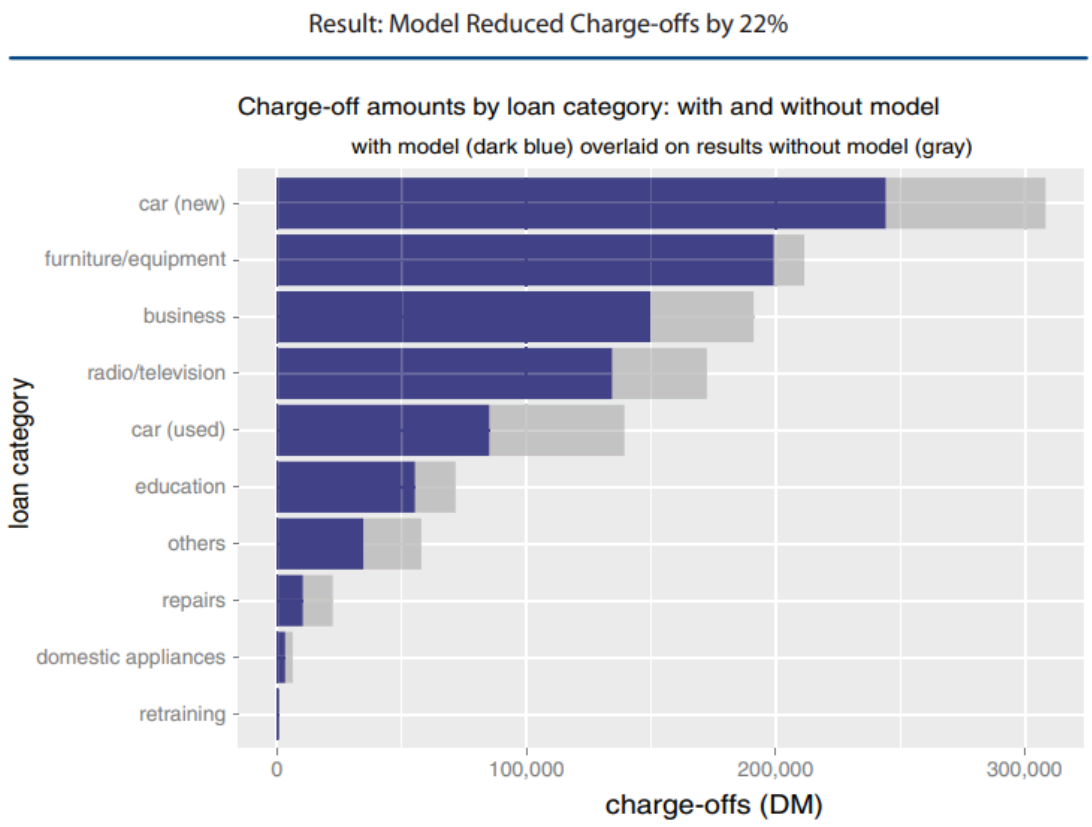


Figure 1.4 Example slide from an executive presentation

A presentation for the model's end users (the loan officers) would instead emphasize how the model will help them do their job better:

- How should they interpret the model?
- What does the model output look like?
- If the model provides a trace of which rules in the decision tree executed, how do they read that?
- If the model provides a confidence score in addition to a classification, how should they use the confidence score?
- When might they potentially overrule the model?

Presentations or documentation for operations staff should emphasize the impact of your model on the resources that they're responsible for.

6. Model deployment and maintenance

- When deploying a data science model, the data scientist's role shifts from daily operations to ensuring the model runs smoothly and remains effective.
- It's important to monitor and update the model as needed, especially when deploying it in a pilot program to catch and address unforeseen issues.
- During deployment, you might encounter situations where human intuition conflicts with the model's recommendations, raising questions about the model's accuracy or completeness.
- Conversely, if the model performs exceptionally well, there might be opportunities to extend its application, such as to additional loan categories.

Setting Expectations: Setting expectations is a crucial part of defining the project goals and success criteria.

- Define Project Goals: Clearly articulate the business goals and performance expectations (e.g., reducing losses by 10%).
- Assess Resources: Ensure the available resources (data, time, tools) are sufficient to meet the goals.
- Project Fluidity: Be prepared to revisit and adjust goals based on insights gained during data exploration and cleaning.

Determining lower bounds on model performance

- **Null Model**: In this case, the null model is the current process that classifies every loan as "GoodLoan." It serves as a baseline or lower bound for model performance. This model will correctly classify 70% of applications (all good loans) but will incorrectly classify 30% (bad loans).

Performance Metrics:

1. Accuracy:
    - The null model has an accuracy of 70%. Any new model must outperform this baseline to be considered useful.
    - If your new model achieves, for example, 75% accuracy, it is better than the null model.
2. Recall:
    - Recall measures how well the model identifies bad loans. The null model, which classifies all loans as "GoodLoan," has zero recall for bad loans because it never correctly identifies bad loans.
    - Suppose your new model has a recall of 80% for bad loans, meaning it correctly identifies 80% of the actual bad loans. This is an improvement over the null model's zero recall for bad loans.
3. Precision:
    - Precision measures how many of the loans classified as "GoodLoan" by the model are actually good. If the new model classifies 80% of loans as "GoodLoan" and 90% of these are indeed good, it has high precision.
4. False Positive Rate:
    - The false positive rate measures how often the model incorrectly classifies a bad loan as a good one. Lowering this rate is crucial to reducing losses.

Exercises

1. House Price Prediction
2. Customer Segmentation
3. Employee Job Role Mapping
4. Factors Affecting Student Performance
5. Sales and Advertising Impact
6. Document Clustering
7. Market Segmentation
8. Job Candidate Ranking
9. E-commerce Product Sorting
10. Credit Score Prediction
11. Risk Assessment for Insurance
12. Sentiment Analysis
13. Disease Diagnosis
14. Student Grade Prediction
15. Search Engine Results
16. Gene Expression Analysis
17. Predicting Sales Revenue

What metric(precision/recall) is prioritized in each use case:

1. Medical Diagnostics (Expensive/Invasive Tests):
2. Search Engines:
3. Fraud Detection:
4. Emergency Alerts:
5. Legal Document Review:
6. Churn Prediction:
7. Spam Filtering:
8. Customer Support (Escalation Alerts):
9. Medical Screening (Early Detection):
10. Quality Control in Manufacturing:

Topic : Starting with R

Vectors

● **Definition:** Vectors are a fundamental data structure in R used to store a sequence of elements of the same type (e.g., numeric, character, logical).

Creation: You use the `c()` function (concatenate) to create vectors. For example:

```
vec <- c(1, 2, 3, 4)  # Numeric vector
char_vec <- c("a", "b", "c")  # Character vector
```

Indexing: R vectors are 1-based indexed. This means indexing starts at 1 rather than 0.

```
vec[1]  # Returns 1, the first element
vec[2]  # Returns 2, the second element
```

Logical Indexing: You can use logical vectors to subset vectors.

```
vec[c(TRUE, FALSE, TRUE, FALSE)]  # Returns c(1, 3)
```

Lists

● **Definition:** Lists are more flexible than vectors; they can hold elements of different types, including other lists.

Creation: Lists are created using the `list()` function.

```
my_list <- list(a = 1, b = "text", c = c(1, 2, 3))
```

Indexing: Lists are also 1-based indexed. You can access elements with single square brackets `[ ]`, which returns a sublist, or double square brackets `[[]]`, which returns the element itself.

```
my_list[1]  # Returns a sublist containing the element named
'a'
```

```
my_list[[1]]  # Returns the value of 'a', which is 1
```

Named Access: Lists can use names for accessing elements:

```
my_list$a  # Accesses the element named 'a', returns 1
my_list["a"]]  # Also returns 1
```

Additional Points

Vector Creation with `c()`: When you use `c()` to combine vectors or values, R flattens the result into a single vector. For example:

```
combined_vec <- c(1, c(2, 3))  # Equivalent to c(1, 2, 3)
```

Vector Indexing:

```
example_vector <- c(10, 20, 30)
example_vector[1]
## [1] 10
example_vector[[2]]
## [1] 20
```

`example_vector[1]` retrieves the first element of the vector, which is `example_vector[[2]]` also retrieves the second element of the vector, which is `20`. In this case, both `[]` and `[[]]` work similarly for vectors.

List Indexing:

```
example_list <- list(a = 10, b = 20, c = 30)
example_list[1]
## $a
## [1] 10
example_list[[2]]
```

```
## [1] 20
```

`example_list[1]` returns a sublist containing the first element of `example_list`, which is itself a list with one element named `a` and its value `10`.

`example_list[[2]]` retrieves the value directly at the second position, which is `20`.

Logical Indexing:

```
example_vector[c(FALSE, TRUE, TRUE)]
## [1] 20 30
example_list[c(FALSE, TRUE, TRUE)]
## $b
## [1] 20
## $c
## [1] 30
```

For `example_vector`, logical indexing `[c(FALSE, TRUE, TRUE)]` selects elements where the logical condition is `TRUE`. This returns the second and third elements, which are `20` and `30`.

For `example_list`, the same logical indexing returns a sublist containing elements at the positions where the condition is `TRUE`. Thus, it returns a list with elements `b` and `c`.

Named Element Access:

```
example_list$b
## [1] 20
example_list[["b"]]
## [1] 20
```

`example_list$b` accesses the value associated with the name `b` directly. `example_list[["b"]]` also accesses the value associated with the name `b`, but using a different syntax. Both methods give you the value `20` associated with the name `b`.

In summary:

- `[]` is used for subsetting and returns a list when used with lists.
- `[[]]` is used for direct access and retrieves the actual value from a list.
- Logical indexing works similarly for vectors and lists but returns different types of results.

Vectorized Operations in R

Vectorized Functions:

- **Definition:** A function is vectorized if it can operate on an entire vector of inputs at once, performing the function on each element independently. This means you can apply the function to a vector and get a vector of results.

Example with **`nchar()`**:

```
nchar("a string")
# [1] 8
```

Here, `nchar("a string")` returns `8` because there are 8 characters in the string "a string".

Vectorized `nchar()` Example:

Using `nchar()` on a vector of strings:

```
nchar(c("a", "aa", "aaa", "aaaa"))
# [1] 1 2 3 4
```

Lists and Vectors as Maps

Definition: Both lists and vectors can act as mappings from values to objects. In R, you can think of lists as mapping named elements to values, and vectors as mapping indices to values.

Primary Operations:

- `[]` (single brackets): Used for subsetting lists and vectors.
- `match()`: Finds the positions of matches of its first argument in its second argument.
- `%in%`: Checks if elements are in a vector or list.
- These operations are vectorized, meaning they work with vectors of values and return a vector of results.

Examples:

Subsetting with `[]`:

```
my_list <- list(a = 1, b = 2, c = 3)
my_list[c("a", "c")]
# $a
# [1] 1
# $c
# [1] 3
```

Subsetting `my_list` with `[]` and a vector of names returns a new list containing the elements corresponding to those names.

Using `match()`:

```
names <- c("a", "b", "c")
match(c("b", "c"), names)
# [1] 2 3
```

`match()` finds the positions of "b" and "c" in the vector `names` and returns their indices.

Using `%in%`:

```
c("a", "b", "d") %in% names
# [1] TRUE TRUE FALSE
```

`%in%` checks if each element of the first vector is present in the second vector, returning a logical vector.

Key Points:

- Vectorized functions apply the function to each element of the vector independently.
- Lists and vectors are powerful data structures in R, and their operations can be applied in a vectorized manner.
- Understanding these operations helps in efficiently manipulating and analyzing data in R.

Logical Operators in R

R provides two types of logical operators that are used for different purposes. Understanding the distinction between these operators is crucial for effective programming in R.

1. Scalar Logical Operators

Operators: `&&` and `||`

- Usage: These operators are designed for use with single (scalar) logical values.
- Behavior: They perform logical operations on just the first element of each logical vector.
- Common Use Case: Typically used in control flow statements such as `if` conditions.

Examples:

&& (Logical AND for Scalars):

```
a <- TRUE
b <- FALSE
result <- a && b
# result is FALSE, because FALSE && anything is FALSE
```

|| (Logical OR for Scalars):

```
a <- TRUE
b <- FALSE
result <- a || b
# result is TRUE, because TRUE || anything is TRUE
```

Key Points:

- && and || evaluate only the first element of each logical vector.
- Useful for control flow operations where you need to make decisions based on single logical conditions.

2. Vectorized Logical Operators

Operators: & and |

- Usage: These operators are intended for use with vectors of logical values.
- Behavior: They perform element-wise logical operations on entire vectors.
- Common Use Case: Typically used in vectorized data operations and logical comparisons involving multiple elements.

Examples:

& (Logical AND for Vectors):

```
vec1 <- c(TRUE, FALSE, TRUE)
vec2 <- c(TRUE, TRUE, FALSE)
```

```
result <- vec1 & vec2
# result is c(TRUE, FALSE, FALSE), performing element-wise AND
```

`|` **(Logical OR for Vectors)**:

```
vec1 <- c(TRUE, FALSE, TRUE)
vec2 <- c(FALSE, TRUE, FALSE)
result <- vec1 | vec2
# result is c(TRUE, TRUE, TRUE), performing element-wise OR
```

Key Points:

- `&` and `|` perform element-wise operations on entire vectors.
- Useful for applying logical conditions across vectors and matrices.

Summary

- `&&` and `||` should be used when dealing with single logical values, like in control flow statements (`if`, `while`).
- `&` and `|` should be used when working with vectors of logical values, for operations across multiple elements.

NULL

Definition: `NULL` in R represents the absence of a value or an empty object. It is equivalent to an empty or length-zero vector.

Characteristics:

Empty Vector: `c()` with no arguments returns `NULL`.

```
c()  # Returns NULL
```

Concatenation with `NULL`: Concatenating `NULL` with other values or vectors results in the other values being returned unchanged.

```
c(c(), 1, NULL)  # Returns 1
```

Usage:

- Indicates Absence: Used to represent the absence of a value or an empty structure.
- Safe Operations: Concatenation and other operations involving `NULL` are well-defined and do not produce errors.

NA

Definition: `NA` stands for "Not Available" and represents a missing or undefined value. It is used to denote values that are not available in a dataset.

Characteristics:

Support for Multiple Types: `NA` can be used with different data types (numeric, character, logical, etc.).

```
vec <- c("a", NA, "c")  # Character vector with NA in the second position
```

Behaves Like NaN: `NA` is similar to `NaN` in that it indicates an undefined or missing value, but unlike `NaN`, `NA` is not restricted to floating-point types.

Logical Expressions: Logical operations involving `NA` behave according to specific rules.

```
FALSE & NA  # Returns FALSE
```

Usage:

- Data Annotation: Useful for indicating missing or unavailable data in a vector or data frame.
- Operations with `NA`: Logical expressions involving `NA` follow specific rules. For example, `FALSE & NA` evaluates to `FALSE`, while `TRUE | NA` evaluates to `NA`.

Key Points

`NULL` vs. `NA`:

- `NULL`: Represents an absence of value or an empty object. Concatenating with `NULL` does not affect the result.
- `NA`: Represents missing or undefined values within data structures. It can be used with various types and influences logical operations based on its presence.
- Handling `NA`: Functions like `is.na()` can be used to detect `NA` values, and operations involving `NA` require careful handling to ensure accurate data processing.

Identifier Naming Conventions in R

1. **CamelCase**:
   - **Description**: Words are concatenated with each word starting with an uppercase letter (e.g., `CamelCase`).
   - **Source**: Google R Style Guide.
2. **Underscore Style**:
   - **Description**: Words are separated by underscores (e.g., `day_one`).
   - **Source**: Recommended by the Advanced R guide.
3. **Dot Notation**:
   - **Description**: Words are separated by dots (e.g., `day.one`).

- **Usage**: Common in built-in R types and packages (e.g., `data.frame`, `data.table`).

Recommendations

- Preferred Style: Use underscore notation (`day_one`) for clarity.
- Avoid Dot Notation: It can be confusing due to its use in object-oriented languages and databases.

Line Breaks in R Code

1. **Line Length:** Keep R source code lines at 80 columns or fewer.
2. **Multi-line Statements:** R accepts multiple-line statements if it's clear where the statement ends.
3. **Correct Line Breaks:** Ensure the statement is split in a way that unambiguously continues to the next line.

   Example:

   ```
   1 +
   2
   ```

4. **Avoid Ambiguity:** Avoid placing line breaks that could make the first line appear as a valid statement on its own.

   Incorrect Example:

   ```
   1
   + 2
   ```

5. **Rule:** Ensure that breaking the statement across lines does not terminate the statement early and cause syntax errors.

Semicolons in R Code

1. **Usage:** R allows semicolons (`;`) as end-of-statement markers.

2. **Recommendation:**
    - Most style guides recommend **not** using semicolons in R code.
    - **Avoid** placing semicolons at the ends of lines.
3. **Best Practice:**
    - Write each statement on a new line without semicolons to enhance code readability and maintain consistency.

ASSIGNMENT

R has many assignment operators (see table 2.1); the preferred one is <-. = can be used for assignment in R, but is also used to bind argument values to argument names by name during function calls (so there is some potential ambiguity in using =).

Table 2.1 Primary R assignment operators

Operator	Purpose	Example
<-	Assign the value on the right to the symbol on the left.	`x <- 5 # assign the value of 5 to the symbol x`
=	Assign the value on the right to the symbol on the left.	`x = 5 # assign the value of 5 to the symbol x`
->	Assign left to right, instead of the traditional right to left.	`5 -> x # assign the value of 5 to the symbol x`

Left-Hand Sides of Assignments in R

1. **Assignment Flexibility:**
    - R allows assignments not just to variable names, but also to slice expressions and indices.
2. **Powerful Array-Slicing:**
    - You can perform assignments using numeric and logical indexing for array slicing.

Example:

Replacing Missing Values:

Initial Data Frame:

```
d <- data.frame(x = c(1, NA, 3))
```

Output:

```
x
1 1
2 NA
3 3
```

Replace `NA` with Zero:

```
d$x[is.na(d$x)] <- 0
```

Resulting Data Frame:

```
x
1 1
2 0
3 3
```

Factors in R

1. **Definition: Factors** are a data type in R used to encode a fixed set of strings as integers. They are useful for representing categorical data.
2. **Advantages:**
   - **Storage Efficiency:** Factors can save storage space compared to character vectors by encoding strings as integers.
   - **Categorical Representation:** Useful for representing categorical variables with a fixed set of values.
3. **Considerations:**

- **Conversion Confusion:** Using `as.numeric()` on factors returns the underlying integer codes, not the original strings. This can be confusing if not managed properly.
- **Value Encoding:** Factors encode all possible values, which can complicate combining datasets with different sets of values.

4. **Best Practices:**

Delay Conversion: Postpone converting strings to factors until late in the analysis to avoid issues. This can be managed by setting `stringsAsFactors = FALSE` in functions like `data.frame()` or `read.table()`.

```
df <- data.frame(x = c("A", "B", "C"), stringsAsFactors = FALSE)
```

Use When Appropriate: Factors are beneficial when you need to:

- Use functions like `summary()` that work well with factors.
- Prepare data for creating dummy indicators for categorical variables.

5. **Example Usage:**

Creating a Factor:

```
f <- factor(c("low", "medium", "high", "medium", "low"))
print(f)
# [1] low    medium high   medium low
# Levels: high low medium
```

Named Arguments in R

1. **Purpose:**
   - Named arguments help clarify function calls and make them more readable, especially for functions with many arguments.
2. **Usage Example:**

Without Named Arguments:

```
setwd("/tmp")
```

With Named Arguments:

```
setwd(dir = "/tmp")
```

Here, `dir` is the name of the argument, and `"/tmp"` is its value.

3. **Advantages:**
   - **Readability:** Makes it clear which argument is being set, improving code legibility.
   - **Optional Arguments:** Useful for functions with optional arguments, making it easier to set specific values without worrying about the order.
4. **Important Note:**

Syntax: Named arguments must be specified using `=`. Do not use the assignment operator `<-` for named arguments.

```
# Correct
function_name(arg1 = value1, arg2 = value2)

# Incorrect
function_name(arg1 <- value1, arg2 <- value2)
```

Named arguments enhance code clarity and maintainability, especially when working with functions that have a large number of parameters.

Package Notation in R

In R, there are two primary methods to use functions from packages:

1. **Attaching the Package:**
   - **Command:** `library(package_name)`

Usage: After attaching the package with `library()`, you can use its functions directly by name.

```
library(dplyr)
result <- filter(mtcars, cyl == 4)
```

2. **Using Package Name with `::`:**
    - **Command:** `package_name::function_name()`

Usage: This method specifies the package explicitly, which helps avoid ambiguity and makes it clear which package the function is from.

```
result <- stats::sd(1:5)
```

Advantages of `::` Notation:

- **Avoid Ambiguity:** Prevents conflicts if multiple packages have functions with the same name.
- **Clarity:** Clearly indicates which package the function belongs to, which can be helpful for readability and debugging.

Using `::` notation is especially useful when working with multiple packages that may have overlapping function names or when you want to make your code more readable and maintainable.

Value Semantics in R

1. **Definition:**
    - R uses "copy by value" semantics, where changes to one reference do not affect other references to the same data.
2. **Behavior:**

- When you create a new reference to data, such as by assigning one variable to another, R ensures that each reference evolves independently.

Example:

```
d <- data.frame(x = 1, y = 2)
d2 <- d
d$x <- 5
print(d)
#   x y
# 1 5 2
print(d2)
#   x y
# 1 1 2
```

Here, d2 retains the original value of x (1), even though d$x was changed to 5.

3. **Advantages:**
   - **Prevents Aliasing Bugs:** Changes in one reference do not affect others, reducing the risk of bugs related to data aliasing.
   - **Safe Coding:** Provides a convenient and safe programming model, as each reference operates independently.
4. **Sharing Changes:** To share changes between references, you need to explicitly reassign the updated data.

```
d2 <- d  # After making changes to `d`, if you want `d2` to reflect those changes
```

R's value semantics simplify coding by eliminating many common data management issues and ensuring that each reference to data operates independently.

Organizing Intermediate Values in R

1. **Use of Temporary Variable `.`:**
    - **Purpose:** Helps in organizing long sequences of calculations into manageable steps.

Example Workflow:

```
data <- data.frame(revenue = c(2, 1, 2),
                   sort_key = c("b", "c", "a"),
                   stringsAsFactors = FALSE)
. <- data
. <- .[order(.$sort_key), , drop = FALSE]
.$ordered_sum_revenue <- cumsum(.$revenue)
.$fraction_revenue_seen <- .$ordered_sum_revenue /
sum(.$revenue)
result <- .
print(result)
```

This approach keeps the original data intact and allows you to step-debug or restart calculations.

- **Advantages:**
- **Clarity:** Each step of the calculation is explicitly shown.
- **Debugging:** Intermediate results are stored in `.` for easier debugging and verification.

2. **dplyr and Piped Notation:**
- **Piped Notation:** Replaces dot notation with `%>%` for cleaner, more readable code.

Example Using dplyr:

```
library(dplyr)
result <- data %>%
  arrange(sort_key) %>%
  mutate(ordered_sum_revenue = cumsum(revenue)) %>%
  mutate(fraction_revenue_seen = ordered_sum_revenue /
sum(revenue))
```

Key Functions:

- `arrange()` for sorting (replaces `order()`).
- `mutate()` for adding or modifying columns (replaces direct assignment).

3. **Comparison and Best Practices:**
   - **dplyr Notation:** Popular for its readability and chaining capabilities.
   - **Intermediate Notation (`.`):** Useful for step-by-step debugging and when dealing with complex sequences of operations.
4. **Considerations:**
   - **Complex Pipelines:** While `dplyr` pipelines are concise, they can be harder to debug. Breaking them into smaller steps with temporary variables can be helpful.
   - **Choosing Notation:** Use the style that best fits your workflow, keeping in mind readability, debugging ease, and personal or team preferences.

In summary, using intermediate variables or piped notation helps manage and debug complex sequences of calculations in R, with each method offering different benefits for organizing and understanding your code.

The `data.frame` Class in R

1. **Definition:**
    - **`data.frame`:** A two-dimensional array-like structure where each column represents a variable, and each row represents an individual instance or observation.
2. **Structure:**
    - **Columns:** Each column can contain different types of data (numeric, character, etc.) and is of equal length across the data.frame.
    - **Rows:** Each row represents a single instance or record, with each cell containing data for one variable of that instance.
    - **Implementation:** Internally, `data.frame` is implemented as a named list of column vectors. While list columns are possible, they are less common.
3. **Operations:**
    - **Efficient Column Operations:** Adding, removing, and accessing columns is generally fast.
    - **Vectorized Operations:** Operations on entire columns are efficient and vectorized.
    - **Row-wise Operations:** Can be more expensive; prefer column-based operations for performance with large datasets.
4. **Schema-like Information:**
    - **Column Names and Types:** `data.frame` includes schema-like information with explicit column names and types, similar to database tables.
    - **Analysis:** Most data manipulations and analyses are performed as transformations on columns rather than rows.
5. **Best Practices:**
    - **Vectorization:** Use vectorized operations for efficiency, especially with large data.frames.
    - **Column-based Transformations:** Prefer working with columns to optimize performance and clarity.

Example:

```
# Creating a data.frame
df <- data.frame(
  Name = c("Alice", "Bob", "Charlie"),
  Age = c(25, 30, 35),
  Score = c(90, 80, 85),
  stringsAsFactors = FALSE
)
print(df)
#     Name Age Score
# 1  Alice  25    90
# 2    Bob  30    80
# 3 Charlie  35    85

# Adding a new column
df$Pass <- df$Score > 85
print(df)
#     Name Age Score Pass
# 1  Alice  25    90 TRUE
# 2    Bob  30    80 FALSE
# 3 Charlie  35    85 FALSE
```

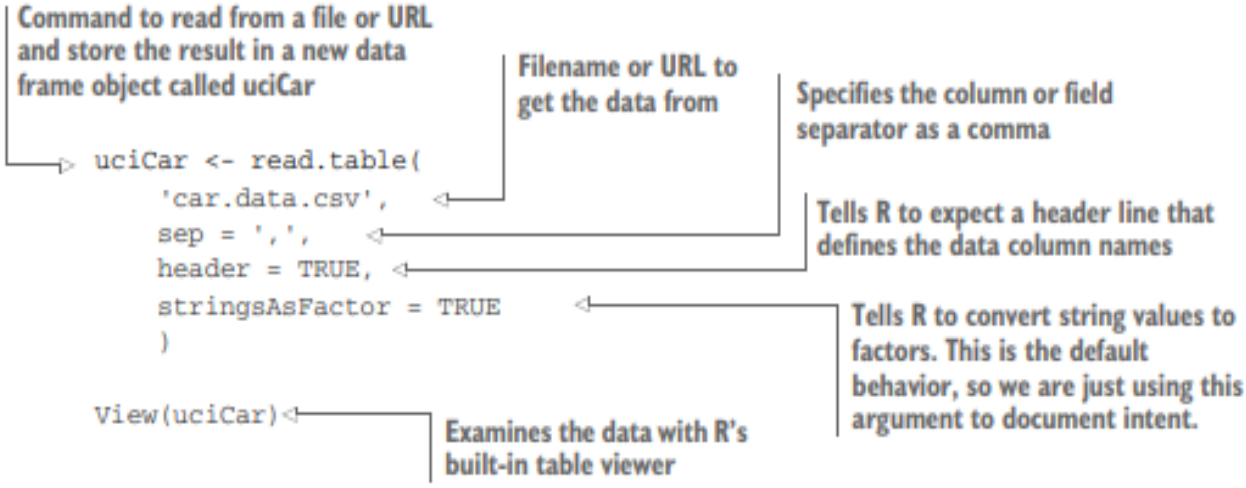
Working with Data from files

Table-Structured Data: The easiest data format to read in R is table-structured data with headers, where data is arranged in rows (instances) and columns (facts or measurements), with column names in the header.

Data Source Reference: The UCI car dataset can be found at http://archive.ics.uci.edu/ml/machine-learning-databases/car/.

Loading well-structured data into R can be done with a single command: read.table(). This command efficiently imports the data into R for analysis.

Listing 2.1 Reading the UCI car data



Listing 2.1 loads the data and stores it in a new R data frame object called `uciCar`, which we show a `View()` of in figure 2.2.

uciCar

Filter

	buying	maint	doors	persons	lug_boot	safety	rating
1	vhigh	vhigh	2	2	small	low	unacc
2	vhigh	vhigh	2	2	small	med	unacc
3	vhigh	vhigh	2	2	small	high	unacc
4	vhigh	vhigh	2	2	med	low	unacc
5	vhigh	vhigh	2	2	med	med	unacc
6	vhigh	vhigh	2	2	med	high	unacc
7	vhigh	vhigh	2	2	big	low	unacc
8	vhigh	vhigh	2	2	big	med	unacc
9	vhigh	vhigh	2	2	big	high	unacc
10	vhigh	vhigh	2	4	small	low	unacc
11	vhigh	vhigh	2	4	small	med	unacc

Showing 1 to 12 of 1,728 entries

Figure 2.2 Car data viewed as a table

Loading Data with read.table()

- Powerful and Flexible: Can handle various data separators (commas, tabs, spaces, pipes, etc.) and offers options for quoting and escaping data.
- Local and Remote Data: Capable of reading from both local files and remote URLs.
- Automatic Decompression: If the file name ends with .gz, read.table() will automatically decompress it.

Examining Data in R

1. class(): Reveals the type of R object (e.g., class(uciCar) will show that uciCar is a data.frame).
2. dim(): Displays the dimensions (number of rows and columns) of a data frame.
3. head():Shows the first few rows of the data frame (e.g., head(uciCar)).
4. help(): Provides documentation for a class (e.g., help(class(uciCar))).

5. str():Gives the structure of an object, detailing types and contents of columns (e.g., str(uciCar)).
6. summary():Summarizes the data, providing statistical summaries for columns (e.g., summary(uciCar)).
7. print():Prints all the data. Caution: For large datasets, this can be time-consuming.
8. View():Opens a spreadsheet-like viewer for the data in RStudio.

Listing 2.2 Exploring the car data

```
class(uciCar)
## [1] "data.frame"
summary(uciCar)
##    buying       maint        doors
```

The loaded object uciCar is of type data.frame.


```
##  high :432   high :432   2    :432
##  low  :432   low  :432   3    :432
##  med  :432   med  :432   4    :432
##  vhigh:432   vhigh:432   5more:432
##
##  persons     lug_boot    safety
##  2   :576   big  :576   high:576
##  4   :576   med  :576   low :576
##  more:576   small:576   med :576
##
##    rating
##  acc  : 384
##  good :  69
##  unacc:1210
##  vgood:  65

dim(uciCar)
## [1] 1728    7
```

[1] is merely an output sequence marker. The actual information is this: uciCar has 1728 rows and 7 columns. Always try to confirm you got a good parse by at least checking that the number of rows is exactly one fewer than the number of lines of text in the original file. The difference of one is because the column header counts as a line of text, but not as a data row.

CSV Files

To use the **read_csv** function, the **readr** package must be installed and accessed using the install.packages and library functions

The csv file is a text file in which the values in the columns are separated by a comma.

File should be named with input.csv.

```
id,name,salary,start_date,dept
1,Rick,623.3,2012-01-01,IT
2,Dan,515.2,2013-09-23,Operations
3,Michelle,611,2014-11-15,IT
4,Ryan,729,2014-05-11,HR
5,Gary,843.25,2015-03-27,Finance
6,Nina,578,2013-05-21,IT
7,Simon,632.8,2013-07-30,Operations
8,Guru,722.5,2014-06-17,Finance
```

R Code:

```
# Reading a CSV file
csv_data <- read.csv("path_to_your_file.csv")
# Display the data
print(csv_data)
```

Analyzing the CSV File

By default the read.csv() function gives the output as a data frame. This can be easily checked as follows. Also we can check the number of columns and rows.

```
data <- read.csv("input.csv")
print(is.data.frame(data))
print(ncol(data))
print(nrow(data))
```

Once we read data in a data frame, we can apply all the functions applicable to data frames

```
#Get the maximum salary
sal <- max(data$salary)
print(sal)

# Get the person detail having max salary.
retval <- subset(data, salary == max(salary))
print(retval)
```

The `subset` function is a common way to filter data in R
The `dplyr` package provides a more modern and readable approach to data manipulation. You can use the `filter` function along with `max` to achieve the same result:

```
library(dplyr)
retval <- data %>%
filter(salary == max(salary))
```

The `data.table` package is another powerful package for data manipulation. You can use its syntax for efficient filtering:

```
library(data.table)
# Convert data to a data.table if it isn't already
data <- as.data.table(data)
retval <- data[salary == max(salary)]
```

You can use base R indexing to achieve the same result without using the `subset` function:

```
retval <- data[data$salary == max(data$salary), ]
```

If you want to use `which.max` to get the index of the maximum value, you can then use this index to filter the data:

```
retval <- data[which(data$salary == max(data$salary)), ]
```

output:

```
    id   name  salary  start_date    dept
5    NA    Gary  843.25  2015-03-27    Finance
```

```
#Get all the people working in IT department
retval <- subset( data, dept == "IT")
print(retval)
```

Output:

```
     id   name     salary   start_date   dept
1     1    Rick     623.3    2012-01-01   IT
3     3    Michelle  611.0    2014-11-15   IT
6     6    Nina     578.0    2013-05-21   IT
```

```
#Get the persons in IT department whose salary is greater than 600
info <- subset(data, salary > 600 & dept == "IT")
print(info)
```

Output:

```
     id   name     salary   start_date   dept
1     1    Rick     623.3    2012-01-01   IT
3     3    Michelle  611.0    2014-11-15   IT
```

```
#Get the people who joined on or after 2014
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))
print(retval)
```

Output:

```
     id   name    salary   start_date    dept
3     3    Michelle 611.00   2014-11-15    IT
4     4    Ryan     729.00   2014-05-11    HR
5    NA    Gary     843.25   2015-03-27    Finance
8     8    Guru     722.50   2014-06-17    Finance
```

Writing into a CSV File

The write.csv() function is used to create the csv file. This file gets created in the working directory.

```
# Create a data frame.
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.
write.csv(retval,"output.csv")
newdata <- read.csv("output.csv")
print(newdata)
```

Output:

```
  X  id   name    salary   start_date    dept
1 3   3    Michelle  611.00   2014-11-15    IT
2 4   4    Ryan     729.00   2014-05-11    HR
3 5   NA    Gary     843.25   2015-03-27    Finance
4 8   8    Guru     722.50   2014-06-17    Finance
```

```
# Create a data frame.
data <- read.csv("input.csv")
retval <- subset(data, as.Date(start_date) > as.Date("2014-01-01"))

# Write filtered data into a new file.
write.csv(retval,"output.csv", row.names = FALSE)
newdata <- read.csv("output.csv")
print(newdata)
```

Output:

```
    id    name     salary   start_date    dept
1    3    Michelle  611.00   2014-11-15    IT
2    4    Ryan      729.00   2014-05-11    HR
3   NA    Gary      843.25   2015-03-27    Finance
4    8    Guru      722.50   2014-06-17    Finance
```

TSV Files

Dataset: Sample TSV data

```
Name      Age  Height
John 25    175
Jane 30    165
Doe 22     180
```

R Code:

```
# Reading a TSV file
tsv_data <- read.table("path_to_your_file.tsv", sep="\t", header=TRUE)
# Display the data
print(tsv_data)
```

Excel Files

Microsoft Excel is the most widely used spreadsheet program which stores data in the .xls or .xlsx format. R can read directly from these files using some excel specific packages.

R Code:

```
library(readxl)
# Reading an Excel file
excel_data <- read_excel("path_to_your_file.xlsx")
# Display the data
print(excel_data)
# Reading an Excel file
library(readxl)
data4 <- read_excel("data/Tesla Deaths.xlsx", sheet = 1)
head(data4, 5)
```

JSON Files

JSON file stores data as text in human-readable format. Json stands for JavaScript Object Notation. R can read JSON files using the rjson package.

Dataset: Sample JSON data

```
 {"Name": "John", "Age": 25, "Height": 175},
  {"Name": "Jane", "Age": 30, "Height": 165},
  {"Name": "Doe", "Age": 22, "Height": 180}
```

To load a JSON file, we will load the rjson package and use `fromJSON` to parse the JSON file.

R Code:

```
library(rjson)
JsonData <- fromJSON(file = 'data/drake_data.json')
print(JsonData[1])
library(jsonlite)
```

```
# Reading a JSON file
json_data <- fromJSON("path_to_your_file.json")
# Display the data
print(json_data)
```

Convert JSON to a Data Frame

We can convert the extracted data above to a R data frame for further analysis using the as.data.frame() function.

```
# Load the package required to read JSON files.
library("rjson")
# Give the input file name to the function.
result <- fromJSON(file = "input.json")
# Convert JSON file to a data frame.
json_data_frame <- as.data.frame(result)
print(json_data_frame)
```

XML Files

XML is a file format which shares both the file format and the data on the World Wide Web, intranets, and elsewhere using standard ASCII text. It stands for Extensible Markup Language (XML). Similar to HTML it contains markup tags. But unlike HTML where the markup tag describes structure of the page, in xml the markup tags describe the meaning of the data contained into the file.

Just like the `read_csv` function, we can load the XML data by providing a URL link to the XML site. It will load the page and parse XML data.

Dataset: Sample XML data

```
<root>
  <person>
    <name>John Doe</name>
    <age>30</age>
    <city>New York</city>
  </person>
  <person>
    <name>Jane Smith</name>
    <age>25</age>
    <city>Los Angeles</city>
  </person>
  <person>
    <name>Emily Johnson</name>
    <age>35</age>
    <city>Chicago</city>
  </person>
</root>
```

R Code:

```
library(XML)
xml_data <- xmlParse("people.xml")

#Get the Root Node:
root <- xmlRoot(xml_data)
```

xmlRoot retrieves the root node of the XML document, which is <root> in this case.

```
#Get the Size of the Root Node:
root_size <- xmlSize(root)
print(paste("Number of top-level nodes:", root_size))
```

xmlSize(root) returns the number of child nodes under the <root> element. In this case, it will be 3 (one for each <person> element).

```
#Extract and Convert person Nodes to a Data Frame:
person_nodes <- getNodeSet(xml_data, "//person")
xml_data_df <- xmlToDataFrame(nodes = person_nodes)
```

getNodeSet selects all <person> nodes from the XML document.
xmlToDataFrame converts these nodes into a data frame. Each row corresponds to a <person> element, with columns for name, age, and city.

```
#Display the Data Frame:
print(xml_data_df)
head(xml_data_df, 5)
```

print shows the entire data frame.
head displays the first 5 rows of the data frame (though in this case, there are only 3 rows).

```
# Print the first node
print(rootnode[1])
```

This prints the first <person> node. The output will include all elements inside this node:

<person>
<name>John Doe</name>
<age>30</age>
<city>New York</city>
</person>

```
# Get the first element of the first node (name of the first person)
print(rootnode[[1]][[1]])
```

This retrieves the first element within the first `<person>` node, which is the `<name>` element:
<name>John Doe</name>

```
# Get the fifth element of the first node (phone number of the first person)
print(rootnode[[1]][[3]])
```

This retrieves the fifth element within the first `<person>` node, which is the
<city>New York</city>

```
# Get the second element of the third node (age of the third person)
print(rootnode[[3]][[2]])
```

This retrieves the second element within the third `<person>` node, which is the `<age>` element:
<age>35</age>

Summary

- xmlRoot allows you to inspect the root of the XML document and understand its structure.
- xmlSize tells you how many top-level nodes (e.g., <person> elements) are present under the root.
- xmlToDataFrame converts XML nodes to a data frame for easy manipulation and analysis in R.

Importing HTML Table into R

scraping the Wikipedia page of Argentina national football team to extract the HTML table and convert it into a data frame

FIFA World Cup

Main article: Argentina at the FIFA World Cup

FIFA World Cup record										Qualification record					
Year	Round	Position	Pld	W	D*	L	GF	GA	Squad	Pld	W	D	L	GF	GA
1930	**Runners-up**	**2nd**	**5**	**4**	**0**	**1**	**18**	**9**	**Squad**	*Qualified as invitees*					
1934	Round of 16	9th	1	0	0	1	2	3	Squad	*Qualified automatically*					
1938	*Withdrew*									*Withdrew*					
1950															
1954															
1958	Group stage	13th	3	1	0	2	5	10	Squad	4	3	0	1	10	2
1962		10th	3	1	1	1	2	3	Squad	2	2	0	0	11	3
1966	Quarter-finals	5th	4	2	1	1	4	2	Squad	4	3	1	0	9	2
1970	*Did not qualify*									4	1	1	2	4	6
1974	Second group stage	8th	6	1	2	3	9	12	Squad	4	3	1	0	9	2
1978	**Champions**	**1st**	**7**	**5**	**1**	**1**	**15**	**4**	**Squad**	*Qualified as hosts*					
1982	Second group stage	11th	5	2	0	3	8	7	Squad	*Qualified as defending champions*					
1986	**Champions**	**1st**	**7**	**6**	**1**	**0**	**14**	**5**	**Squad**	6	4	1	1	12	6
1990	**Runners-up**	**2nd**	**7**	**2**	**3**	**2**	**5**	**4**	**Squad**	*Qualified as defending champions*					
1994	Round of 16	10th	4	2	0	2	8	6	Squad	8	4	2	2	9	10
1998	Quarter-finals	6th	5	3	1	1	10	4	Squad	16	8	6	2	23	13

Image from Wikipedia

To load an HTML table, we will use XML and RCurl packages

Provide the Wikipedia URL to the `getURL` function and then add the object to the `readHTMLTable` function

The function will extract all of the HTML tables from the website, and we just have to explore them individually to select the one we want.

```
library(XML)
library(RCurl)
# Fetch the content of the Wikipedia page for the Brazil national football team
url<- getURL("https://en.wikipedia.org/wiki/Brazil_national_football_team")
# Read the HTML tables from the fetched content
tables <- readHTMLTable(url)
# Extract the 1st table from the list of tables
data<- tables[1]
data$`nameofcolumn`
```

	V1	V2	V3	V4	V5	V6	V7	V8	V9
1	FIFA World Cup record		Qualification record	null	null	null	null	null	null
2	Year	Round	Position	Pld	W	D*	L	GF	GA
3	1930	Group stage	6th	2	1	0	1	5	2
4	1934	Round of 16	14th	1	0	0	1	1	3
5	1938	Third place	3rd	5	3	1	1	14	11
6	1950	Runners-up	2nd	6	4	1	1	22	6
7	1954	Quarter-finals	5th	3	1	1	1	8	5

Moreover, you can use the rvest package to read HTML using URL, extract all tables, and display it as a dataframe.

- read_html(URL) for extracting HTML data from websites.
- html_nodes(file, "table") for extracting tables from HTML data.
- html_table(tables[1]) for converting HTML tables to dataframe.

```
library(rvest)
url <- "https://en.wikipedia.org/wiki/Argentina_national_football_team"
file<-read_html(url)
tables<-html_nodes(file, "table")
data8 <- html_table(tables[1])
View(data8)
```

Data Frame:

A data frame is a table or a two-dimensional array-like structure in which each column contains values of one variable and each row contains one set of values from each column.

Following are the characteristics of a data frame.

- The column names should be non-empty.
- The row names should be unique.
- The data stored in a data frame can be of numeric, factor or character type.
- Each column should contain same number of data items.

```
# Create the data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),
      Start_date  =  as.Date(c("2012-01-01",  "2013-09-23",  "2014-11-15",
"2014-05-11", "2015-03-27")), stringsAsFactors = FALSE)
# Print the data frame.
print(emp.data)
```

When we execute the above code, it produces the following result –

```
  emp_id    emp_name     salary     start_date
1     1     Rick        623.30     2012-01-01
2     2     Dan         515.20     2013-09-23
3     3     Michelle    611.00     2014-11-15
4     4     Ryan        729.00     2014-05-11
5     5     Gary        843.25     2015-03-27
```

Get the Structure of the Data Frame

The structure of the data frame can be seen by using str() function.

```
str(emp.data)
```

When we execute the above code, it produces the following result –

```
'data.frame':   5 obs. of  4 variables:
 $ emp_id    : int  1 2 3 4 5
 $ emp_name  : chr  "Rick" "Dan" "Michelle" "Ryan" ...
 $ salary    : num  623 515 611 729 843
 $ start_date: Date, format: "2012-01-01" "2013-09-23" "2014-11-15" "2014-05-11" ...
```

Summary of Data in Data Frame

The statistical summary and nature of the data can be obtained by applying summary() function.

```
print(summary(emp.data))
```

When we execute the above code, it produces the following result –

```
    emp_id    emp_name            salary       start_date
 Min.   :1   Length:5           Min.   :515.2   Min.   :2012-01-01
 1st Qu.:2   Class :character   1st Qu.:611.0   1st Qu.:2013-09-23
 Median :3   Mode  :character   Median :623.3   Median :2014-05-11
 Mean   :3                      Mean   :664.4   Mean   :2014-01-14
 3rd Qu.:4                      3rd Qu.:729.0   3rd Qu.:2014-11-15
 Max.   :5                      Max.   :843.2   Max.   :2015-03-27
```

Extract Data from Data Frame

Extract specific column from a data frame using column name.

```
result <- data.frame(emp.data$emp_name,emp.data$salary)
print(result)
```

When we execute the above code, it produces the following result –

```
  emp.data.emp_name emp.data.salary
1              Rick          623.30
2               Dan          515.20
3          Michelle          611.00
4              Ryan          729.00
5              Gary          843.25
```

```
# Extract first two rows.
result <- emp.data[1:2,]
print(result)

# Extract 3rd and 5th row with 2nd and 4th column.
result <- emp.data[c(3,5),c(2,4)]
print(result)
```

Expand Data Frame

A data frame can be expanded by adding columns and rows.

Add Column

Just add the column vector using a new column name.

```
# Add the "dept" coulmn.
emp.data$dept <- c("IT","Operations","IT","HR","Finance")
v <- emp.data
print(v)
```

When we execute the above code, it produces the following result –

```
  emp_id   emp_name    salary    start_date       dept
1    1     Rick        623.30    2012-01-01       IT
2    2     Dan         515.20    2013-09-23       Operations
3    3     Michelle    611.00    2014-11-15       IT
4    4     Ryan        729.00    2014-05-11       HR
5    5     Gary        843.25    2015-03-27       Finance
```

Add Row

To add more rows permanently to an existing data frame, we need to bring in the new rows in the same structure as the existing data frame and use the rbind() function.

```
# Create the first data frame.
emp.data <- data.frame(
   emp_id = c (1:5),
   emp_name = c("Rick","Dan","Michelle","Ryan","Gary"),
   salary = c(623.3,515.2,611.0,729.0,843.25),
    start_date   =   as.Date(c("2012-01-01",   "2013-09-23",   "2014-11-15",
"2014-05-11",  "2015-03-27")),
  dept = c("IT","Operations","IT","HR","Finance"), stringsAsFactors = FALSE)

# Create the second data frame
emp.newdata <-     data.frame(
   emp_id = c (6:8),
   emp_name = c("Rasmi","Pranab","Tusar"),
   salary = c(578.0,722.5,632.8),
```

```
start_date = as.Date(c("2013-05-21","2013-07-30","2014-06-17")),
dept = c("IT","Operations","Fianance"),
stringsAsFactors = FALSE)

# Bind the two data frames.
emp.finaldata <- rbind(emp.data,emp.newdata)
print(emp.finaldata)
```

When we execute the above code, it produces the following result –

	emp_id	emp_name	salary	start_date	dept
1	1	Rick	623.30	2012-01-01	IT
2	2	Dan	515.20	2013-09-23	Operations
3	3	Michelle	611.00	2014-11-15	IT
4	4	Ryan	729.00	2014-05-11	HR
5	5	Gary	843.25	2015-03-27	Finance
6	6	Rasmi	578.00	2013-05-21	IT
7	7	Pranab	722.50	2013-07-30	Operations
8	8	Tusar	632.80	2014-06-17	Fianance

Introduction to Data Science

Topic : Exploring the Data(Unit-1)

Before jumping into modeling, it's crucial to thoroughly examine your dataset. No dataset is flawless; it may have missing or incorrect data and inconsistent fields. By investigating and cleaning the data beforehand, you avoid the risk of repeatedly revising your work or creating a model that produces inaccurate predictions. Addressing data issues early helps save time and prevent future complications.

Data exploration uses a combination of summary statistics—means and medians, variances, and counts— and visualization, or graphs of the data. You can spot some problems just by using summary statistics; other problems are easier to find visually.

Using summary statistics to spot problem

Example:

1. **Objective:** Develop a model to predict the likelihood that a customer does not have health insurance.
2. **Data Collection:** You have a dataset with known health insurance status for each customer.
3. **Predictor Variables:** You've identified several customer properties that you believe are relevant for predicting health insurance coverage. These properties include:
    - **Age**
    - **Employment Status**
    - **Income**
    - **Information about Residence**
    - **Vehicle Ownership**

By analyzing these variables, you aim to build a model that estimates the probability of health insurance coverage and identifies customers who are likely uninsured.

The goal is to assess:

1. **Predictive Potential:** Determine if the customer information you have (e.g., age, employment status, income, residence, and vehicle details) is useful for predicting health insurance coverage.
2. **Data Quality:** Evaluate whether the collected data is of sufficient quality and completeness to provide meaningful and reliable predictions. This includes checking for accuracy, consistency, and relevance of the data to ensure it can effectively support the model.

The `summary()` command in R offers a quick overview of your dataset, which helps you:

1. **Assess Data Utility:** Determine if the data contains relevant information that could be useful for predicting health insurance coverage.
2. **Evaluate Data Quality:** Review basic statistics and distributions to gauge whether the data is of sufficient quality for analysis. This includes checking for completeness, consistency, and any potential issues.
3. **Guide Exploration:** Use the summary output to identify areas that may require further exploration or preprocessing, such as handling missing values, correcting inaccuracies, or transforming variables.

Listing 3.1 The `summary()` command

Change this to your actual path to the directory where you unpacked PDSwR2

```
setwd("PDSwR2/Custdata")
customer_data = readRDS("custdata.RDS")
summary(customer_data)
##     custid              sex        is_employed       income
##  Length:73262       Female:37837   FALSE: 2351   Min.   : -6900
##  Class :character   Male  :35425   TRUE :45137   1st Qu.:  10700
##  Mode  :character                 NA's :25774   Median :  26200
##                                                 Mean   :  41764
##                                                 3rd Qu.:  51700
##                                                 Max.   :1257000
##
```

The variable is_employed is missing for about a third of the data. The variable income has negative values, which are potentially invalid.


```
##             marital_status  health_ins
##  Divorced/Separated:10693   Mode :logical
##  Married           :38400   FALSE:7307
##  Never married     :19407   TRUE :65955
##  Widowed           : 4762
##
##
##
##                       housing_type   recent_move      num_vehicles
##  Homeowner free and clear    :16763   Mode :logical   Min.   :0.000
##  Homeowner with mortgage/loan:31387   FALSE:62418     1st Qu.:1.000
##  Occupied with no rent       : 1138   TRUE :9123      Median :2.000
##  Rented                      :22254   NA's :1721      Mean   :2.066
##  NA's                        : 1720                   3rd Qu.:3.000
##                                                       Max.   :6.000
##                                                       NA's   :1720
##       age              state_of_res     gas_usage
##  Min.   :  0.00   California  : 8962   Min.   :  1.00
##  1st Qu.: 34.00   Texas       : 6026   1st Qu.:  3.00
##  Median : 48.00   Florida     : 4979   Median : 10.00
##  Mean   : 49.16   New York    : 4431   Mean   : 41.17
##  3rd Qu.: 62.00   Pennsylvania: 2997   3rd Qu.: 60.00
##  Max.   :120.00   Illinois    : 2925   Max.   :570.00
##                   (Other)     :42942   NA's   :1720
```

About 90% of the customers have health insurance.

The variables housing_type, recent_move, num_vehicles, and gas_usage are each missing 1720 or 1721 values.

The average value of the variable age seems plausible, but the minimum and maximum values seem unlikely. The variable state_of_res is a categorical variable; summary() reports how many customers are in each state (for the first few states).

Typical problems revealed by data summaries

- Missing values
- Invalid values and outliers
- Data ranges that are too wide or too narrow
- The units of the data

Missing Values

Impact of Missing Values:

- Missing values can affect the quality and completeness of your data analysis.
- If a data field has a significant number of missing values, it may not be suitable for use in analysis without addressing these missing values.

Listing 3.2 Will the variable `is_employed` be useful for modeling?

```
## is_employed
## FALSE: 2321
## TRUE :44887
## NA's :24333
```

The variable is_employed is missing for more than a third of the data. Why? Is employment status unknown? Did the company start collecting employment data only recently? Does NA mean "not in the active workforce" (for example, students or stay-at-home parents)?

```
##                       housing_type   recent_move
## Homeowner free and clear    :16763   Mode :logical
## Homeowner with mortgage/loan:31387   FALSE:62418
## Occupied with no rent       : 1138   TRUE :9123
## Rented                      :22254   NA's :1721
## NA's                        : 1720
##
##
##   num_vehicles     gas_usage
##  Min.   :0.000   Min.   :  1.00
##  1st Qu.:1.000   1st Qu.:  3.00
##  Median :2.000   Median : 10.00
##  Mean   :2.066   Mean   : 41.17
##  3rd Qu.:3.000   3rd Qu.: 60.00
##  Max.   :6.000   Max.   :570.00
##  NA's   :1720    NA's   :1720
```

The variables housing_type, recent_move, num_vehicles, and gas_usage are missing relatively few values—about 2% of the data. It's probably safe to just drop the rows that are missing values, especially if the missing values are all in the same 1720 rows.

- For instance, if the `is_employed` variable in your dataset has many missing values, R's default behavior might cause it to ignore over a third of the data, potentially skewing results or reducing the dataset's effectiveness.

Summary:

- **Investigate Missing Values:** Determine why values are missing as this can provide valuable insights.
- **Assess Impact:** Evaluate the significance of the missing data on your analysis.
- **Decision on Handling:** Decide whether to include the variable in your model based on its importance.
- **Different Strategies:**
    - For less critical variables, you might exclude rows with missing data.
    - For key variables, consider treating missing values as a distinct category.
- **Model Training:** Address missing data effectively during model training to maintain model robustness and informativeness.

INVALID VALUES AND OUTLIERS

- Even when a data column has no missing values, it’s essential to verify the validity of the values present.
- This involves checking for invalid entries such as negative numbers in fields that should only contain non-negative values (e.g., age or income) or text in numeric fields.
- Additionally, identify outliers—data points that fall far outside the expected range of values.
- Ensuring that the values are both valid and within a reasonable range helps maintain data quality and accuracy in points

Listing 3.3 Examples of invalid values and outliers

```
summary(customer_data$income)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -6900   11200   27300   42522   52000 1257000

summary(customer_data$age)
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00   34.00   48.00   49.17   62.00  120.00
```

Negative values for income could indicate bad data. They might also have a special meaning, like "amount of debt." Either way, you should check how prevalent the issue is, and decide what to do. Do you drop the data with negative income? Do you convert negative values to zero?

Customers of age zero, or customers of an age greater than about 110, are outliers. They fall out of the range of expected customer values. Outliers could be data input errors. They could be special sentinel values: zero might mean "age unknown" or "refuse to state." And some of your customers might be especially long-lived.

DATA RANGE:

When analyzing your dataset, it's crucial to consider the variability in the values of predictor variables like age and income. If these variables are important for predicting health insurance coverage, you need to ensure that there is sufficient variation in their values.

Wide Data Range Issues: Data with a very broad range, such as income spanning from zero to over a million dollars, can pose challenges for some modeling methods.

Mitigating Range Issues: Techniques like logarithmic transformations can help manage wide data ranges

Narrow Data Range Issues: Data with a very narrow range, such as ages between 50 and 55, may not be useful for prediction due to limited variability.

Predictor Effectiveness: Variables with low variability may not provide meaningful insights or predictions in modeling scenarios.

Listing 3.5 Checking units; mistakes can lead to spectacular errors

```
IncomeK = customer_data$income/1000
summary(IncomeK)
 ##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   -6.90   10.70   26.20   41.76   51.70 1257.00
```

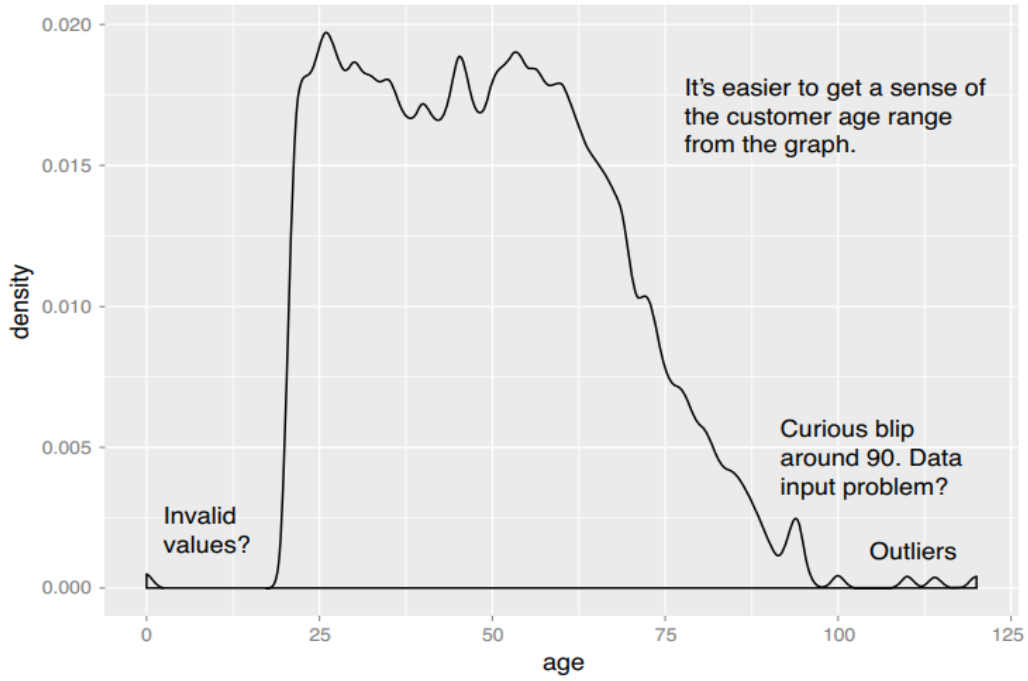
The variable IncomeK is defined as IncomeK = customer_data$income/1000. But suppose you didn't know that. Looking only at the summary, the values could plausibly be interpreted to mean either "hourly wage" or "yearly income in units of $1000."

Spotting problems using graphics and visualization

The use of graphics to examine data is called visualization

```
summary(custdata$age)
# Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
# 0.00   34.00   48.00   49.16   62.00  120.00
```

It's easier to read the mean, median, and central 50% of the customer population from the summary.



Visually checking distributions for a single variable

- Histograms
- Density plots
- Bar charts
- Dot plots

The visualizations help you answer questions like these:

1.Peak Value of the Distribution:

- **Purpose:** Identifies the most frequent value or the central tendency of the data.
- **Example:** If you're analyzing income data and find that the peak value is $30,000, this indicates that $30,000 is the most common income among your customers.

2. Number of Peaks (Unimodality vs. Bimodality):

- **Purpose:** Determines the distribution shape. Unimodal distributions have one peak, while bimodal distributions have two. This can indicate different subgroups within the data.
- **Example:** If you find two peaks in income data—one around $20,000 and another around $70,000—it might suggest that your dataset includes two distinct income groups, possibly representing different customer segments.

3. Normal or Lognormal Distribution:

- **Purpose:** Helps in choosing the appropriate statistical methods. Many techniques assume normality. If the data is lognormal, transformations might be needed.

- **Example:** If income data is heavily right-skewed, it may be lognormal. Applying a log transformation could help normalize the data for more accurate modeling.

4. Variation in Data:

- **Purpose:** Assesses the spread or dispersion of the data. High variation indicates diverse data points, while low variation suggests similar values.
- **Example:** If ages of customers vary from 20 to 70 years, there's high variation. This diversity can help in understanding how different age groups might affect health insurance coverage.

5.Concentration in Interval or Category:

1. **Purpose:** Identifies if the data is clustered within a certain range or category, which can influence analysis and modeling decisions.
2. **Example:** If most customers' incomes fall within $30,000 to $40,000, the data is concentrated in this interval. This could impact the effectiveness of income as a predictor if the range is too narrow.

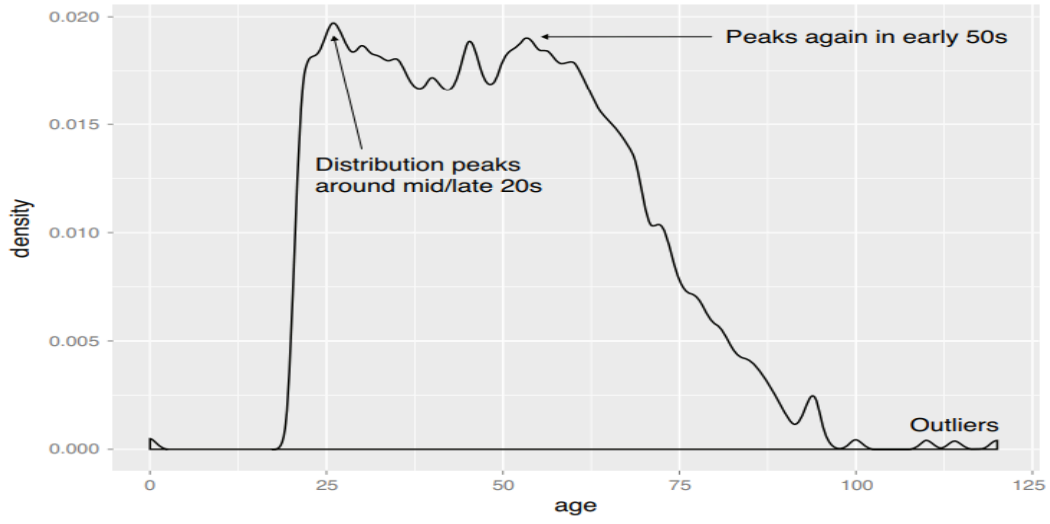


Figure 3.3 The density plot of age

The graph in figure 3.3 is somewhat flattish between the ages of about 25 and about 60, falling off slowly after 60. However, even within this range, there seems to be a peak at around the late-20s to early 30s range, and another in the early 50s. This data has multiple peaks: it is not unimodal.

The histogram and the density plot are two visualizations that help you quickly examine the distribution of a numerical variable.

HISTOGRAMS

A basic histogram bins a variable into fixed-width buckets and returns the number of data points that fall into each bucket as a height.

The example describes a method for summarizing and analyzing monthly gas heating bills by grouping them into intervals and counting the number of customers in each interval.

1. **Grouping Data:**
    - **Create Buckets:** Divide the gas bill amounts into predefined intervals or buckets (e.g., $0–10, $10–20, $20–30, etc.).
    - **Boundary Rule:** If a customer's bill is at the boundary (e.g., exactly $20), place them in the higher interval (e.g., $20–30).
2. **Counting Customers:**
    - **Count per Bucket:** For each interval, count the number of customers whose gas bills fall into that interval.

Purpose:

- **Understanding Distribution:** This method helps you understand the distribution of gas heating bills among customers, showing how many customers fall into each spending range.
- **Data Analysis:** It simplifies the data into manageable categories, making it easier to identify trends or patterns in customer spending on gas heating.

The resulting histogram is shown in figure 3.5.

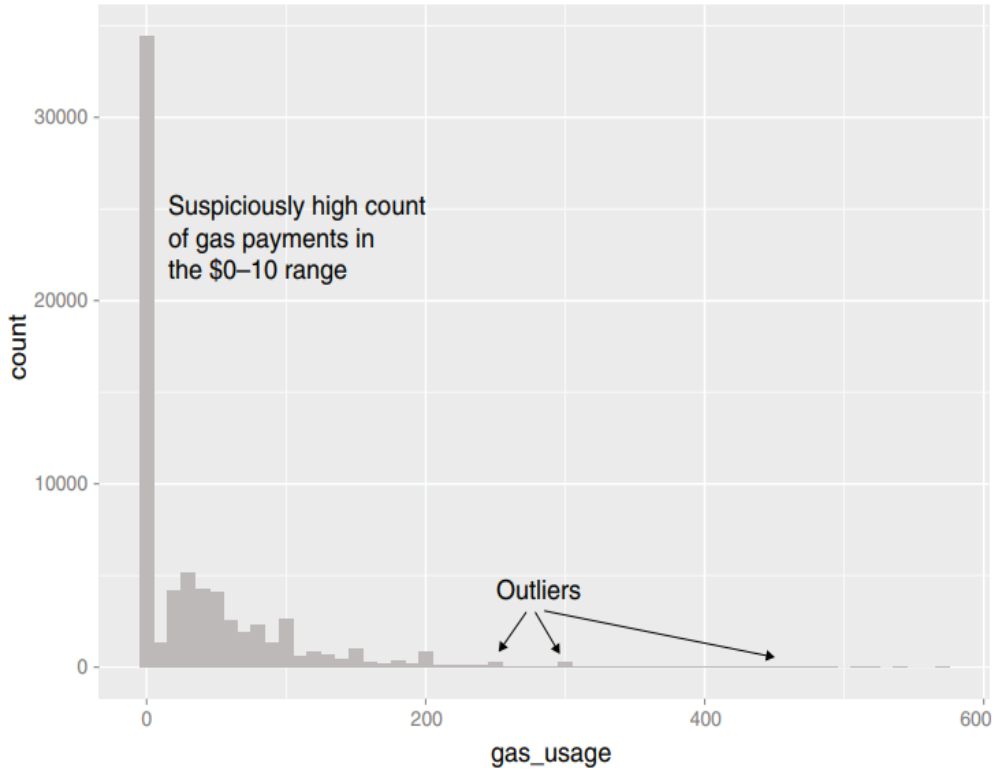


Figure 3.5 A histogram tells you where your data is concentrated. It also visually highlights outliers and anomalies.

You create the histogram in figure 3.5 in ggplot2 with the geom_histogram layer.

Listing 3.6 Plotting a histogram

```
library(ggplot2)
ggplot(customer_data, aes(x=gas_usage)) +
  geom_histogram(binwidth=10, fill="gray")
```

Load the ggplot2 library, if you haven't already done so.

The binwidth parameter tells the geom_histogram call how to make bins of $10 intervals (default is datarange/30). The fill parameter specifies the color of the histogram bars (default: black).

- `aes(x=gas_usage)`: Specifies the aesthetic mapping where `gas_usage` is mapped to the x-axis. This tells ggplot2 to use the `gas_usage` column from `customer_data` for the x-axis of the plot.
- `binwidth=10`: Sets the width of each bin (or interval) in the histogram to 10 units. This controls how the data is grouped into bins.
- `fill="gray"`: Fills the bars of the histogram with a gray color. This changes the color of the bars to make the plot visually appealing.

Histograms with an appropriate binwidth can reveal where data is concentrated and identify potential outliers and anomalies.

For example, a histogram might show some customers with unusually high gas bills, suggesting they could be outliers and possibly excluded from analysis.

It might also reveal a high concentration of customers with $0–10/month gas bills, which could indicate many do not use gas heating.

The primary disadvantage of histograms is that you must decide ahead of time how wide the buckets are. If the buckets are too wide, you can lose information about the shape of the distribution. If the buckets are too narrow, the histogram can look too noisy to read easily. An alternative visualization is the density plot.

Density Plot

- **Continuous Histogram**: A density plot serves as a continuous version of a histogram, representing the distribution of a variable.
- **Area Rescaling:** The area under the density plot curve is rescaled to equal one, ensuring the plot represents a probability distribution.
- **Point Interpretation:** Each point on the density plot represents the fraction or percentage of data points near a particular value, though these fractions are typically very small.
- **Focus on Shape:** The primary interest in a density plot is the overall shape of the curve rather than the exact values on the y-axis.

Listing 3.7 Producing a density plot

```
library(scales)

ggplot(customer_data, aes(x=income)) + geom_density() +
  scale_x_continuous(labels=dollar)
```

The scales package brings in the dollar scale notation.

Sets the x-axis labels to dollars

Loads the `scales` package, which provides functions for formatting and scaling data, such as converting numbers into currency formats.

`aes(x=income)`: Maps the `income` column from `customer_data` to the x-axis of the plot.

geom_density(): Adds a density plot layer to the ggplot object. This creates a smoothed curve representing the distribution of the `income` variable.

scale_x_continuous(labels=dollar): Formats the x-axis to display numbers as currency values.

`labels=dollar`: Uses the `dollar` function from the `scales` package to convert numeric values into a currency format on the x-axis, making it easier to interpret income values as monetary amounts

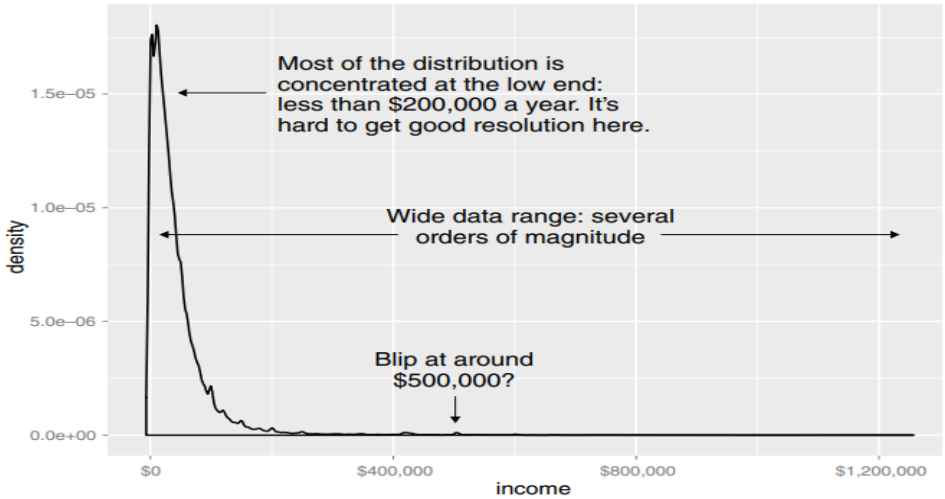


Figure 3.6 Density plots show where data is concentrated.

When the data range is very wide and the mass of the distribution is heavily concentrated to one side, like the distribution in figure 3.6, it's difficult to see the details of its shape. For instance, it's hard to tell the exact value where the income distribution has its peak

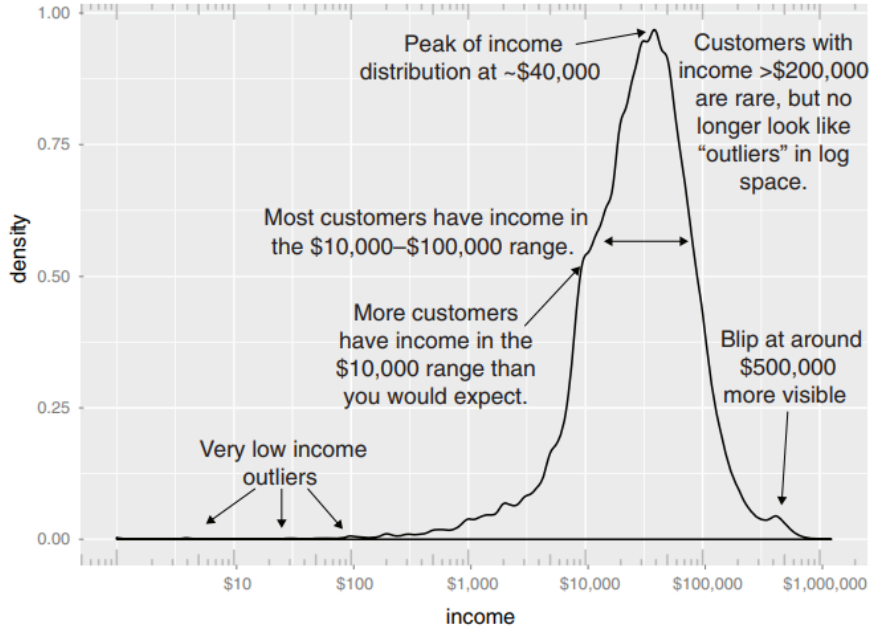


Figure 3.7 The density plot of income on a log10 scale highlights details of the income distribution that are harder to see in a regular density plot.

If the data is non-negative, then one way to bring out more detail is to plot the distribution on a logarithmic scale, as shown in figure 3.7. This is equivalent to plotting the density plot of log10(income).

In ggplot2, you can plot figure 3.7 with the geom_density and scale_x_log10 layers, such as in the following listing

Listing 3.8 Creating a log-scaled density plot

```
ggplot(customer_data, aes(x=income)) +
  geom_density() +
  scale_x_log10(breaks = c(10, 100, 1000, 10000, 100000, 1000000),
  ➥labels=dollar) +
   annotation_logticks(sides="bt", color="gray")
```

Sets the x-axis to be in log10 scale, with manually set tick points and labels as dollars

Adds log-scaled tick marks to the top and bottom of the graph

```
ggplot(customer_data, aes(x=income)) +
```

- Initializes a ggplot object using `customer_data` as the dataset.
- `aes(x=income)`: Maps the `income` variable to the x-axis of the plot.

```
geom_density() +
```

- Adds a density plot layer to the ggplot object. This creates a smoothed curve that represents the distribution of the `income` variable.

```
scale_x_log10(breaks = c(10, 100, 1000, 10000, 100000, 1000000),labels=dollar) +
```

- Transforms the x-axis to a logarithmic scale, which is useful for data with a wide range or exponential growth patterns.
- `breaks = c(10, 100, 1000, 10000, 100000, 1000000)`: Specifies the points where ticks (labels) should appear on the x-axis. These are values at which the axis will have labels.
- `labels=dollar`: Formats the tick labels as currency using the `dollar` function from the `scales` package, making the numbers easier to read as monetary amounts.

```
annotation_logticks(sides="bt", color="gray")
```

- Adds logarithmic scale ticks to the plot.
- `sides="bt"`: Specifies that the ticks should be drawn on the bottom and top of the plot.
- `color="gray"`: Sets the color of the ticks to gray.

You should use a logarithmic scale when:

1. **Percent Change is Important:** Log scales are useful when you want to emphasize changes in orders of magnitude rather than absolute values.

2. **Data is Skewed:** They help visualize data that is heavily skewed, making it easier to interpret.
3. **Income Example:** In income data, where the range spans several orders of magnitude, a logarithmic scale makes it clearer how significant changes are relative to the overall distribution. This helps in visualizing data where a few high values compress most of the data into a narrow range on a regular scale.

Bar Charts:

Display the frequency of discrete values for categorical data, similar to histograms but for categorical variables.

Example Use: For analyzing marital status, a bar chart helps visualize the distribution of different marital status categories among customers.

Predictive Analysis: Ensure there are enough customers in each marital status category to effectively explore the relationship between marital status and health insurance coverage.

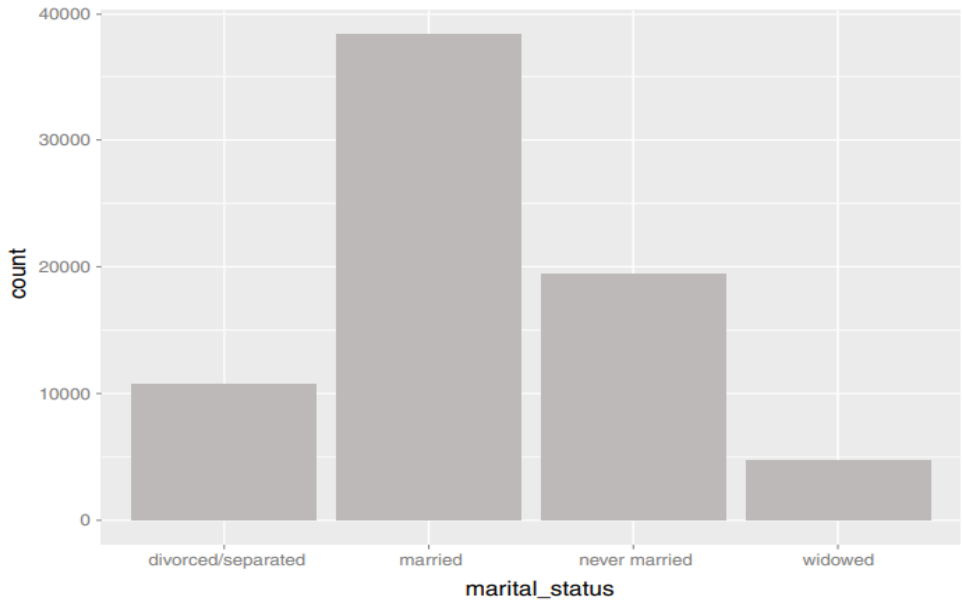


Figure 3.8 Bar charts show the distribution of categorical variables.

The ggplot2 command to produce figure 3.8 uses geom_bar:

ggplot(customer_data, aes(x=marital_status)) + geom_bar(fill="gray")

`aes(x=marital_status)`: Maps the `marital_status` variable from `customer_data` to the x-axis. This sets up the aesthetic mappings for the plot.

`geom_bar()`: Creates a bar plot, where each bar represents the frequency of each unique value in the `marital_status` variable.

`fill="gray"`: Sets the color of the bars to gray. This parameter specifies the fill color for the bars in the bar chart.

Listing 3.9 Producing a horizontal bar chart

```
ggplot(customer_data, aes(x=state_of_res)) +
  geom_bar(fill="gray") +
  coord_flip()
```

Plots bar chart as before: state_of_res is on x-axis, count is on y-axis

Flips the x and y axes: state_of_res is now on the y-axis

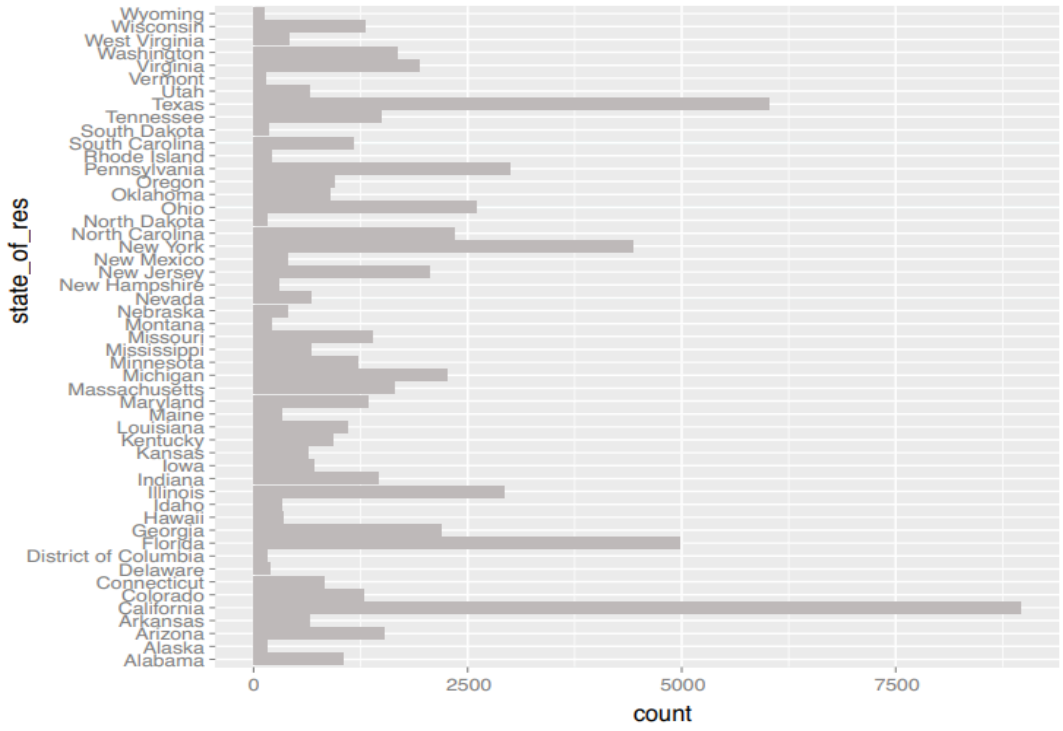
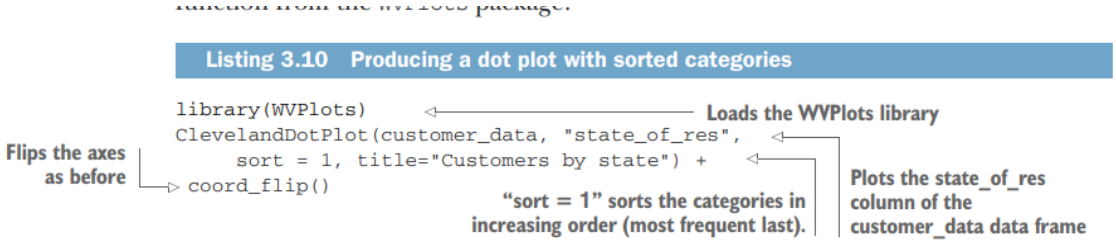


Figure 3.9 A horizontal bar chart can be easier to read when there are several categories with long names.

Dot Plot vs. Bar Chart: Prefer dot plots over bar charts for visualizing discrete counts because dot plots avoid the perceptual issue of area differences in bars, focusing only on height differences for comparison.

Sorting Data: Recommends when sorting data in both bar charts and dot plots to give more effectively extract insights, such as identifying states with the most or fewest customers.



`Customer_data`: Specifies the dataset to be used for the plot.

`"state_of_res"`: Indicates the column in the dataset that contains the categorical variable (state of residence) to be plotted.

`sort = 1`: Orders the dot plot by the frequency of occurrences of each category, with 1 indicating descending order (most frequent first).

`title="Customers by state"`: Adds a title to the plot.

coord_flip() : Flips the x and y axes of the plot. This turns the Cleveland dot plot into a horizontal layout, which often makes it easier to read categorical labels when there are many categories or long names.

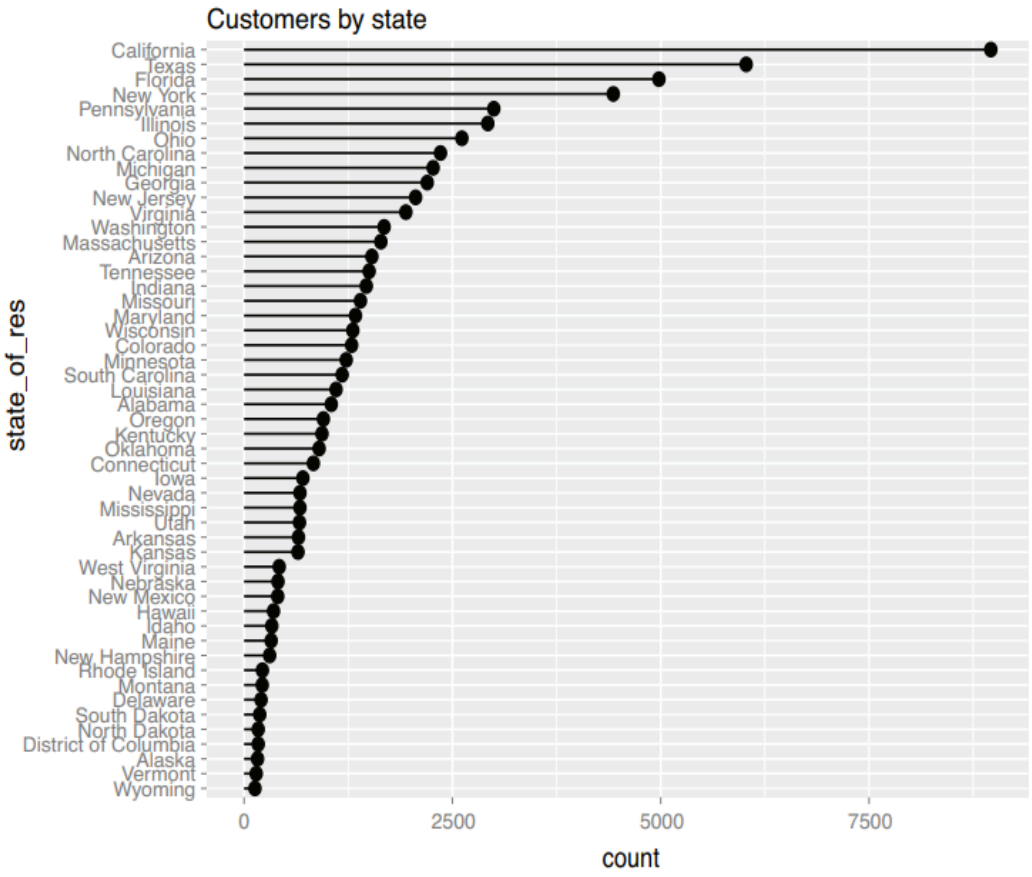


Figure 3.10 Using a dot plot and sorting by count makes the data even easier to read.

Table 3.2 Visualizations for one variable

Graph type	Uses	Examples
Histogram or density plot	Examine data range Check number of modes Check if distribution is normal/lognormal Check for anomalies and outliers	Examine the distribution of customer age to get the typical customer age range Examine the distribution of customer income to get typical income range
Bar chart or dot plot	Compare frequencies of the values of a categorical variable	Count the number of customers from different states of residence to determine which states have the largest or smallest customer base

Visually checking relationships between two variables

–Look at the relationship between two variables.

For example, you might want to answer questions like these:

- Is there a relationship between the two inputs age and income in my data?
- If so, what kind of relationship, and how strong?
- Is there a relationship between the input marital status and the output health insurance? How strong?

Various visualizations:

- Line plots and scatter plots for comparing two continuous variables
- Smoothing curves and hexbin plots for comparing two continuous variables at high volume
- Different types of bar charts for comparing two discrete variables
- Variations on histograms and density plots for comparing a continuous and discrete variable

LINE PLOTS

Line plots work best when the relationship between two variables is relatively clean: each x value has a unique (or nearly unique) y value

Listing 3.11 Producing a line plot

```
x <- runif(100)
y <- x^2 + 0.2*x
ggplot(data.frame(x=x,y=y), aes(x=x,y=y)) + geom_line()
```

The y variable is a quadratic function of x.

Plots the line plot

First, generate the data for this example. The x variable is uniformly randomly distributed between 0 and 1.

`runif(100)`: Creates a vector `x` of 100 random numbers uniformly distributed between 0 and 1.

`y <- x^2 + 0.2*x`: Computes the corresponding `y` values using a quadratic function with a linear term added. This creates a non-linear relationship between `x` and `y`.

`ggplot(data.frame(x=x, y=y), aes(x=x, y=y))`:

- **`data.frame(x=x, y=y)`**: Converts the `x` and `y` vectors into a data frame, which is a tabular format suitable for plotting with `ggplot2`.
- **`aes(x=x, y=y)`**: Maps the `x` values to the x-axis and `y` values to the y-axis for the plot.

`geom_line()`:

- Adds a line plot layer to the ggplot object. This plots the `x` and `y` values as connected lines, showing the relationship between them.

When the data is not so cleanly related, line plots aren't as useful; you'll want to use the **scatter plot** instead

SCATTER PLOTS AND SMOOTHING CURVES

You'd expect there to be a relationship between age and health insurance, and also a relationship between income and health insurance. But what is the relationship between age and income? If they track each other perfectly, then you might not want to use both variables in a model for health insurance.

The appropriate summary statistic is the correlation, which we compute on a safe subset of our data.

Listing 3.12 Examining the correlation between age and income

```
customer_data2 <- subset(customer_data,
                         0 < age & age < 100 &
                          0 < income & income < 200000)
```

Only consider a subset of data with reasonable age and income values.

```
cor(customer_data2$age, customer_data2$income)
 ## [1] 0.005766697
```

Gets correlation of age and income

Resulting correlation is positive but nearly zero.

```
customer_data2 <- subset(customer_data,
  0 < age & age < 100 &
  0 < income & income < 200000)
```

- Filters the `customer_data` dataset to include only rows where `age` and `income` fall within specified ranges.
  - `0 < age & age < 100`: Filters for `age` values greater than 0 and less than 100. This excludes ages that are unrealistic or out of a plausible range.
  - `0 < income & income < 200000`: Filters for `income` values greater than 0 and less than 200,000. This excludes extremely low or high income values that may be outliers or errors.

The result is a new dataset, `customer_data2`, that only contains rows meeting these criteria.

```
cor(customer_data2$age, customer_data2$income)
```

- Computes the correlation coefficient between the `age` and `income` variables in the filtered dataset.
  - `cor()`: Calculates the Pearson correlation coefficient, which measures the linear relationship between two variables.
  - `customer_data2$age`: The `age` variable from the filtered dataset.
  - `customer_data2$income`: The `income` variable from the filtered dataset.
- **Value:** The correlation coefficient is approximately 0.0058.
- **Interpretation:** This value is very close to 0, indicating a negligible or very weak linear relationship between `age` and `income` in the filtered dataset. This suggests that `age` and `income` are almost uncorrelated in this subset of the data.

A visualization gives you more insight into what's going on than a single number can. Let's try a scatter plot first (figure 3.12). Because our dataset has over 64,000 rows, which is too large for a legible scatterplot, we will sample the dataset down before plotting. You plot figure 3.12 with geom_point, as shown in listing 3.13.

Figure 3.12 A scatter plot of income versus age

Listing 3.13 Creating a scatterplot of age and income

```
set.seed(245566)
customer_data_samp <-
     dplyr::sample_frac(customer_data2, size=0.1, replace=FALSE)

ggplot(customer_data_samp, aes(x=age, y=income)) +
       geom_point() +
       ggtitle("Income as a function of age")
```

Make the random sampling reproducible by setting the random seed.

Creates the scatterplot

For legibility, only plot a 10% sample of the data. We will show how to plot all the data in a following section.

```
set.seed(245566)
```

- Sets the seed for random number generation to ensure reproducibility. By using `set.seed()`, you guarantee that the same random samples are selected each time you run the code, which is crucial for reproducibility of results.

```
customer_data_samp <- dplyr::sample_frac(customer_data2, size=0.1, replace=FALSE)
```

- Randomly samples 10% of the rows from the `customer_data2` dataset.
  - `dplyr::sample_frac()`: A function from the `dplyr` package that samples a fraction of rows from a dataset.
  - `customer_data2`: The dataset from which to sample.
  - `size=0.1`: Specifies that 10% of the data should be sampled.
  - `replace=FALSE`: Indicates that sampling is done without replacement, meaning that each row can only be sampled once.

The result is a new dataset, `customer_data_samp`, which contains a 10% random sample of the original dataset `customer_data2`.

```
ggplot(customer_data_samp, aes(x=age, y=income)) +
geom_point() +
ggtitle("Income as a function of age")
```

- Creates a scatter plot to visualize the relationship between `age` and `income` in the sampled dataset.
  - `ggplot(customer_data_samp, aes(x=age, y=income))`:
    - Initializes a `ggplot` object using the `customer_data_samp` dataset.
    - `aes(x=age, y=income)`: Maps `age` to the x-axis and `income` to the y-axis.
  - `geom_point()`:

    - Adds a scatter plot layer to the ggplot object. Each point represents an observation in the dataset, with its position determined by the `age` and `income` values.
  - `ggtitle("Income as a function of age")`:
    - Adds a title to the plot. The title "Income as a function of age" helps to describe what the scatter plot represents.

The relationship between age and income isn't easy to see. You can try to make the relationship clearer by also **plotting a smoothing curve through the data**, as shown in figure 3.13.

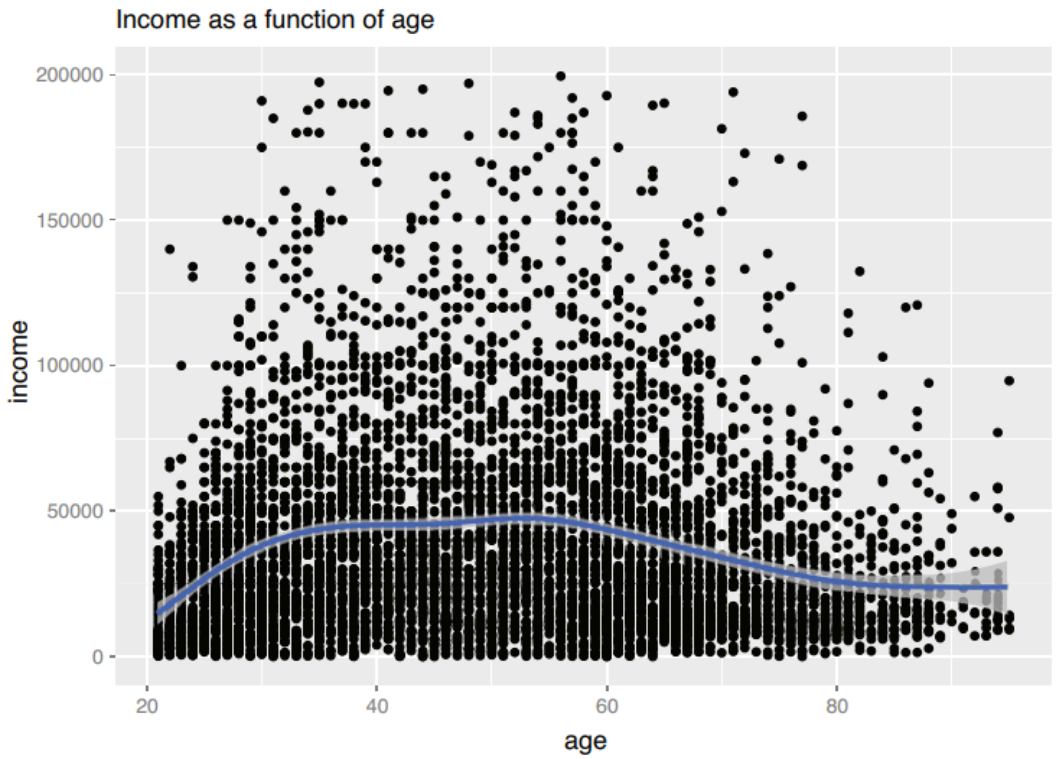


Figure 3.13 A scatter plot of income versus age, with a smoothing curve

The Smoothing curve makes it easier to see that in this population, income tends to increase with age from a person's twenties until their mid-thirties, after which income increases at a slower, almost flat, rate until about a person's mid-fifties. Past the mid fifties, income tends to decrease with age

In ggplot2, you can plot a smoothing curve to the data by using geom_smooth :
ggplot(customer_data_samp, aes(x=age, y=income)) + geom_point() + geom_smooth() + ggtitle("Income as a function of age")

By default, `geom_smooth()` includes a "standard error" ribbon around the smoothing curve, which indicates the uncertainty of the estimate. The ribbon is wider where there are fewer data points and narrower where the data is dense. In dense scatterplots, like the one in figure 3.13, this ribbon may not be visible except at the extremes. Since the scatterplot itself provides similar information, you can disable the ribbon by setting `se=FALSE`

`A scatter plot with a smoothing curve also makes a useful visualization of the relationship between a continuous variable and a Boolean`

Suppose you're considering using age as an input to your health insurance model. You might want to plot health insurance coverage as a function of age, as shown in figure 3.14.

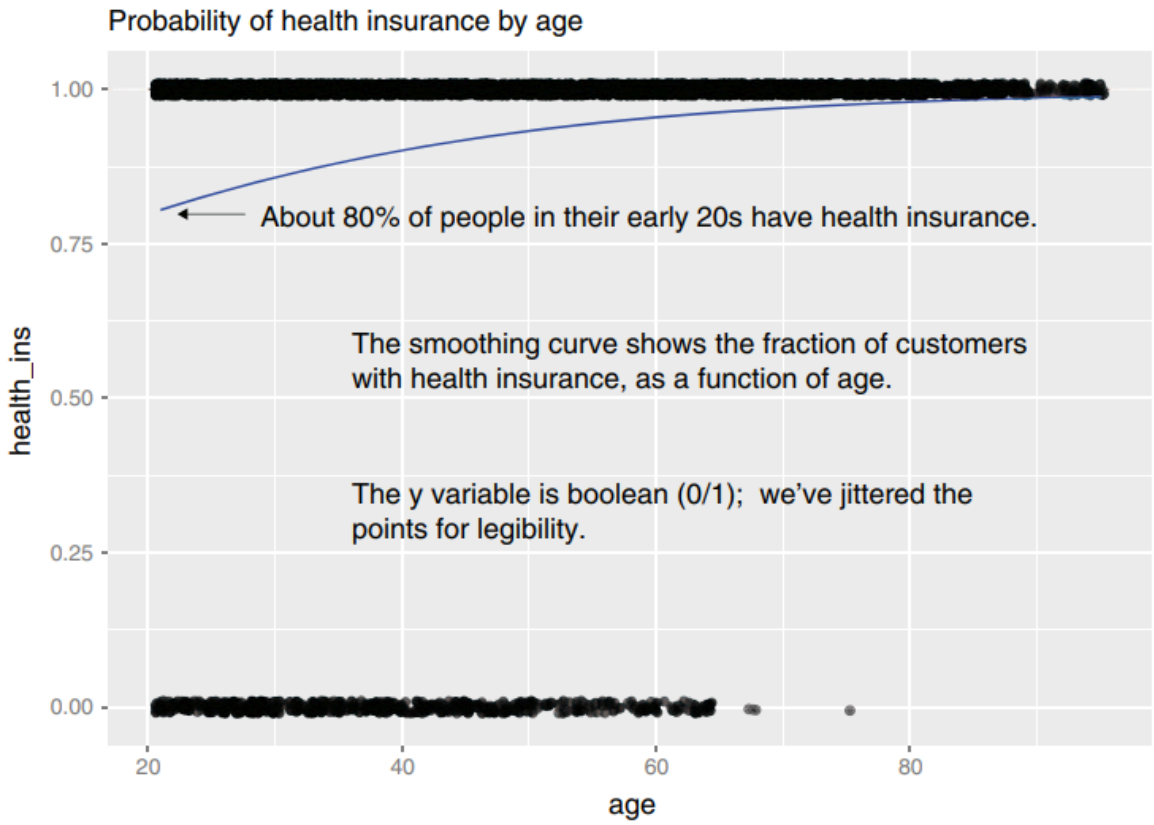


Figure 3.14 Fraction of customers with health insurance, as a function of age

The variable health_ins has the value 1 (for TRUE) when the person has health insurance, and 0 (for FALSE) otherwise. A scatterplot of the data will have all the y-values at 0 or 1, which may not seem informative, but a smoothing curve of the data estimates the average value of the 0/1 variable health_ins as a function of age. The average value of health_ins for a given age is simply the probability that a person of that age in your dataset has health insurance

Figure 3.14 shows you that the probability of having health insurance increases as customer age increases, from about 80% at age 20 to nearly 100% after about age 75.

An easy way to plot figure 3.14 is with the BinaryYScatterPlot function from WVPlots:

```
BinaryYScatterPlot(customer_data_samp, "age", "health_ins", title = "Probability of health insurance by age")
```

By default, BinaryYScatterPlot fits a logistic regression curve through the data. You will learn more about logistic regression in chapter 8, but for now just know that a logistic regression tries to estimate the probability that the Boolean outcome y is true, as a function of the data x. If you tried to plot all the points from the customer_data2 dataset, the scatter plot would turn into an illegible smear. To plot all the data in higher volume situations like this, try an aggregated plot, like a **hexbin** plot.

Topic : Exploring data and Visualization

HEXBIN PLOTS

- Hexbin plots are an excellent tool for visualizing the relationship between **two continuous variables**, especially when **dealing with large datasets**.
- A hexbin plot is like a two-dimensional histogram.
- The data is divided into hexagon, and the number of data points in each hexagon is represented by color or shading
- Color intensity of each hexagon represents the density of data points.

Example:

Listing 3.14 Producing a hexbin plot

```
library(WVPlots)
HexBinPlot(customer_data2, "age", "income", "Income as a function of age") +
   geom_smooth(color="black", se=FALSE)
```

Loads the WVPlots library

Plots the hexbin of income as a function of age

Adds the smoothing line in black; suppresses standard error ribbon (se=FALSE)

`library(WVPlots)`: Loads the `WVPlots` package, which provides various convenient functions for plotting, including `HexBinPlot`.

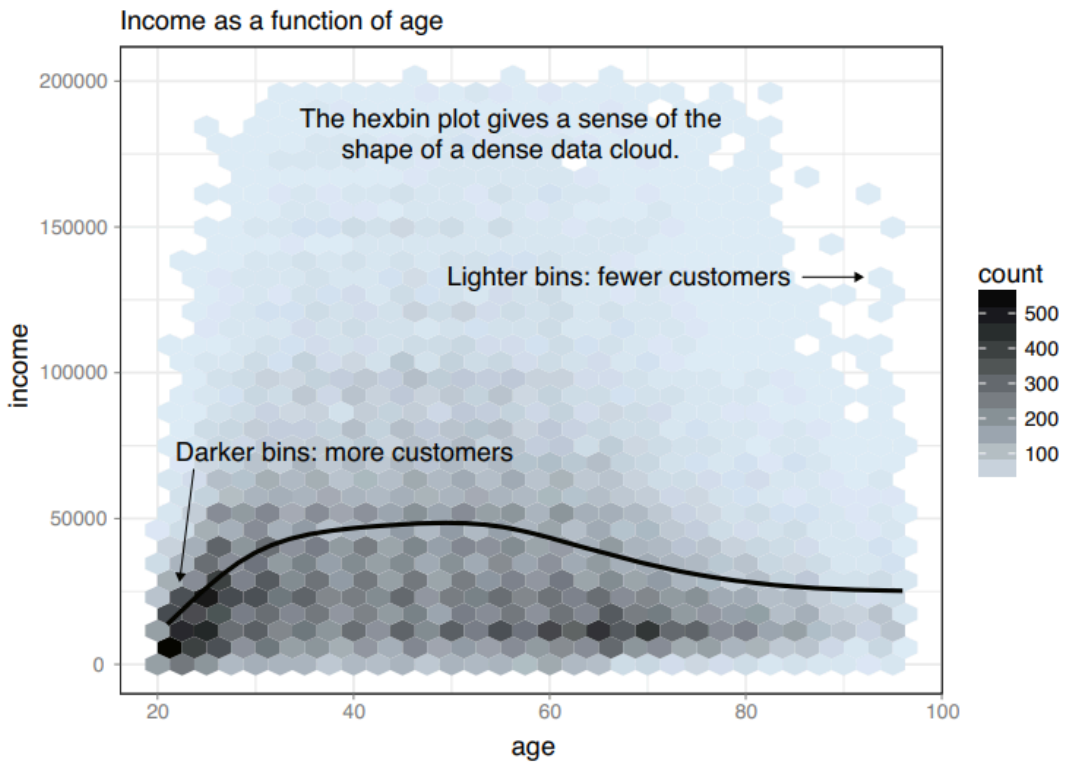
`HexBinPlot(customer_data2, "age", "income", "Income as a function of age")`

- **`customer_data2`**: This is the dataset you're working with. It should be a data frame containing at least two columns: one for `age` and another for `income`.
- **`"age"`**: This is the name of the column in `customer_data2` that contains the age data.

- `"income"`: This is the name of the column in `customer_data2` that contains the income data.
- `"Income as a function of age"`: This is the title of the plot.

```
+ geom_smooth(color = "black", se = FALSE)
```

- `geom_smooth()`: This function adds a smoothing curve to the plot, which helps visualize the overall trend in the data. It fits a smooth line (typically a linear regression) through the data points.
- `color = "black"`: Specifies that the smoothing line should be black.
- `se = FALSE`: This argument turns off the confidence interval shading around the smoothing line, providing a cleaner look to the plot.



What the Plot Represents

- **Hexbin Plot:** The plot itself displays a hexbin plot of income versus age. Each hexagon represents a "bin" of data points, and the color intensity within the hexagon indicates the density of data points in that bin.
- **Smoothing Curve:** The black smoothing curve overlays the hexbin plot, tracing the central tendency of income across different ages. This curve helps in identifying the general trend or relationship between age and income, showing how income typically varies as age changes.

For example, if the curve trends upwards, it indicates that income generally increases with age. Conversely, if it trends downwards, it indicates that income generally decreases with age. The absence of the confidence interval (due to `se = FALSE`) means the plot focuses solely on the trend without showing uncertainty around the trend line.

BAR CHARTS FOR TWO CATEGORICAL VARIABLES

Let's examine the relationship between marital status and the probability of health insurance coverage.

Stacked Bar Chart

- **Purpose:** Visualize the total number of people in each marital status category and the breakdown of insured vs. uninsured individuals within each category.
- **Advantages:** The stacked bar chart makes it easy to compare the total number of people in each marital category, and to compare the number of uninsured people in each marital category.
- **Disadvantages:** It is difficult to compare the number of insured people across different categories because the bars don't start at the same level. The focus is more on the totals and the proportion within each category rather than direct comparisons between categories.

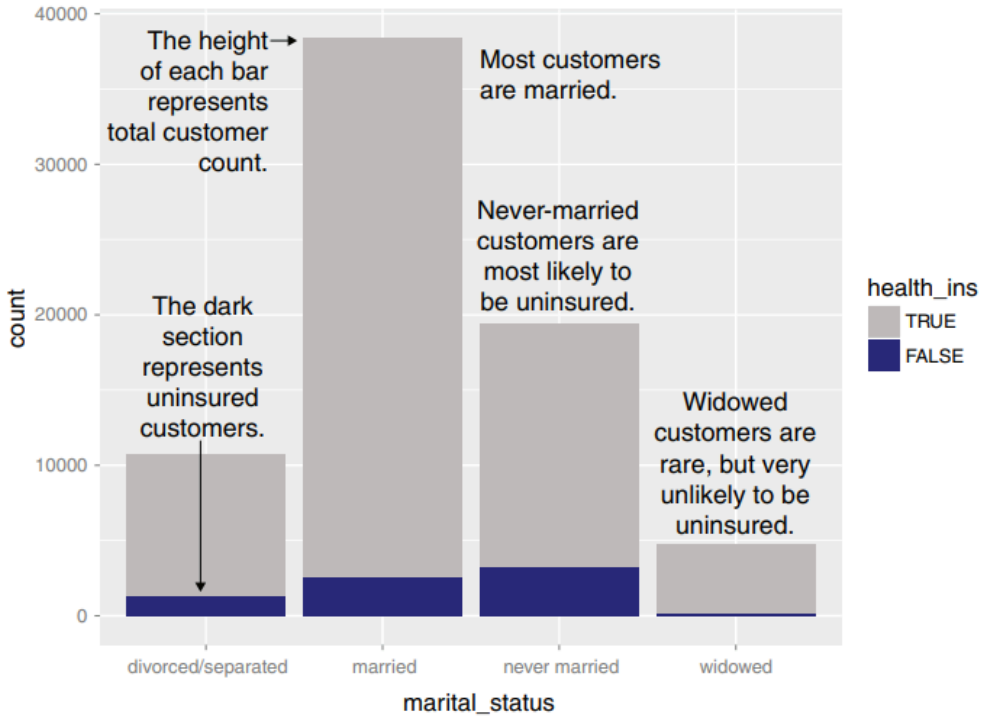


Figure 3.16 Health insurance versus marital status: stacked bar chart

Side-by-Side Bar Chart

- **Purpose:** Compare the number of insured and uninsured people directly across different marital status categories.
- **Advantages:** Provides a clearer comparison of the number of insured and uninsured people across categories, as the bars for each category are aligned side by side.
- **Disadvantages:** The total number of people in each category is less obvious because the bars for insured and uninsured are separated, making it harder to visualize the total population within each category.

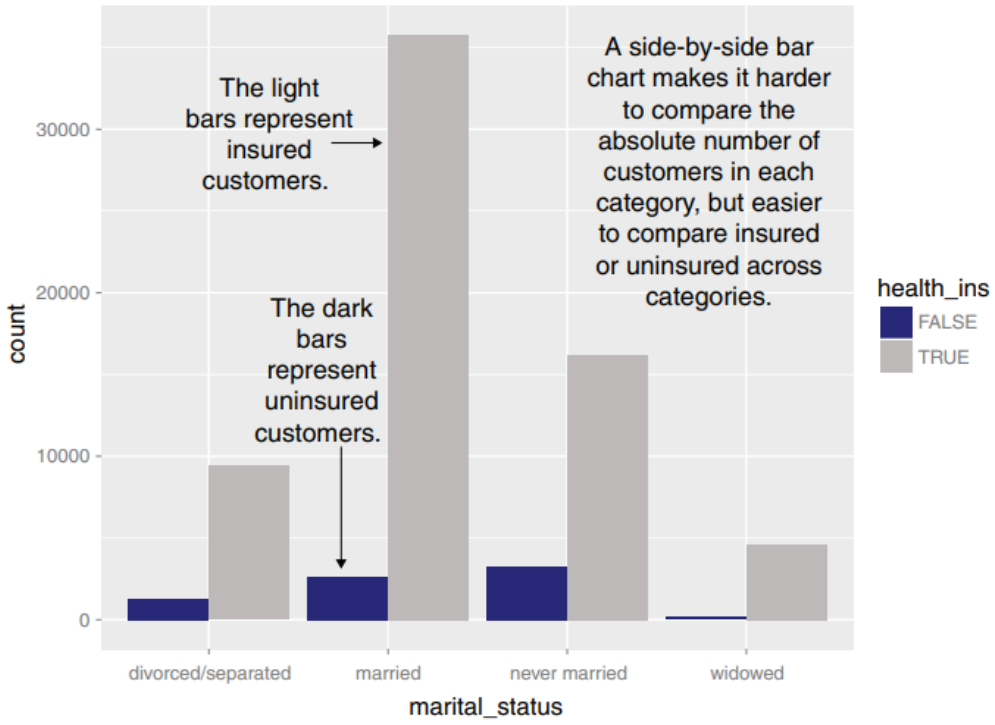


Figure 3.17 Health insurance versus marital status: side-by-side bar chart

Shadow Plot

- **Purpose:** Combine the benefits of comparing insured vs. uninsured populations across categories while maintaining a sense of the total number of people in each category.
- **Description:** The shadow plot superimposes two graphs (one for insured and one for uninsured) against a “shadow graph” of the total population. This approach allows for comparisons both within and across categories while preserving information about category totals.
- **Advantages:** Balances the need to compare within and across categories while keeping a sense of the overall population size in each category.
- **Disadvantages:** Shadow plots can be more complex and harder to interpret at a glance compared to simpler bar charts.

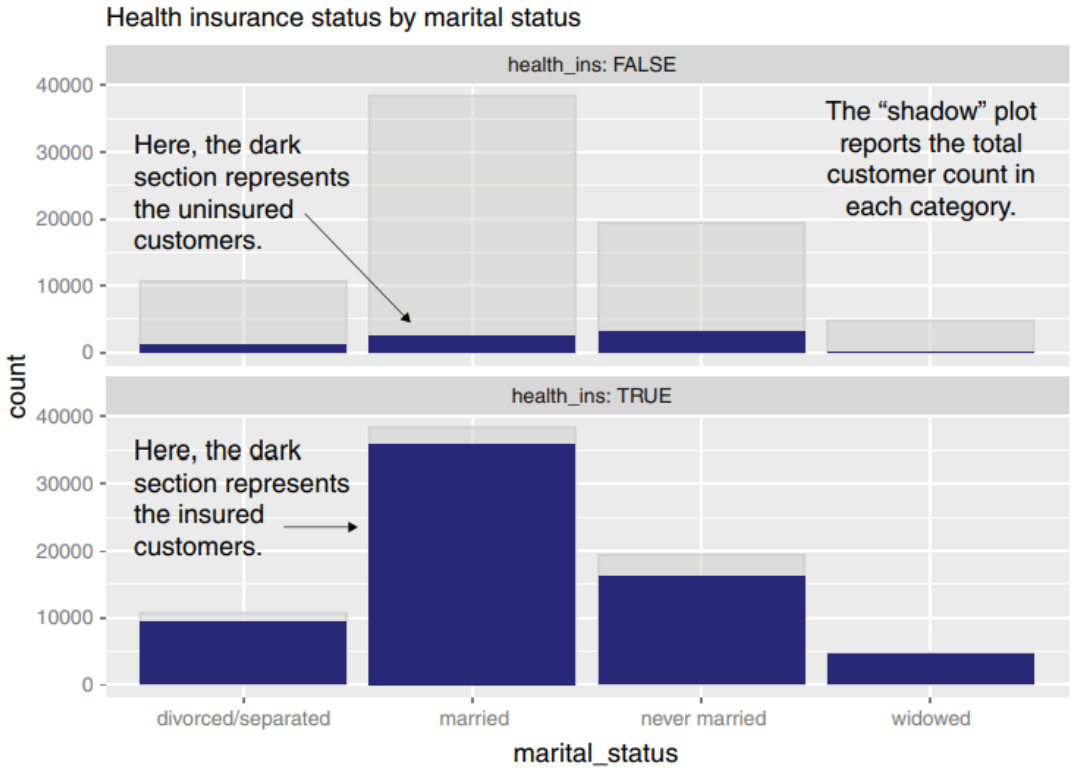


Figure 3.18 Health insurance versus marital status: shadow plot

The main shortcoming of all the preceding charts is that you can't easily compare the ratios of insured to uninsured across categories, especially for rare categories like Widowed. You can use what ggplot2 calls a filled bar chart to plot a visualization of the ratios directly, as in figure 3.19.

Filled Bar Chart

- **Purpose:** Visualize the ratio of insured to uninsured people within each marital status category.
- **Advantages:** Clearly shows the proportions of insured vs. uninsured individuals within each category, making it easy to compare these ratios across categories.
- **Disadvantages:** While the filled bar chart effectively conveys ratios, it loses information about the total number of people in each category. Rare categories, like "Widowed," may be less visible despite being significant in terms of insurance coverage.

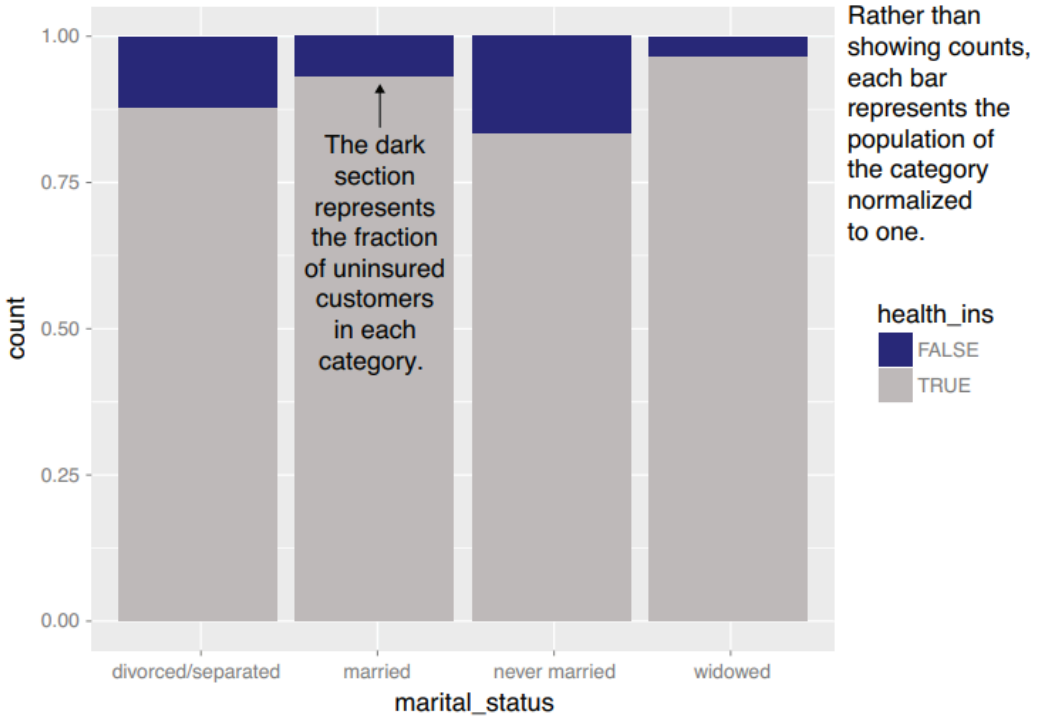


Figure 3.19 Health insurance versus marital status: filled bar chart

Key Considerations:

- **Choosing the Right Chart:** The choice of bar chart depends on what aspect of the data is most important to convey:
  - **Total Population and Distribution:** Use a stacked bar chart.
  - **Direct Comparison Across Categories:** Use a side-by-side bar chart.
  - **Combining Totals and Comparisons:** Consider a shadow plot.
  - **Focusing on Ratios:** Use a filled bar chart.

Listing 3.15 Specifying different styles of bar chart

```
ggplot(customer_data, aes(x=marital_status, fill=health_ins)) +
                      geom_bar()  <-------- Stacked bar chart, the default

ggplot(customer_data, aes(x=marital_status, fill=health_ins)) +
                    geom_bar(position = "dodge") <-------- Side-by-side bar chart

ShadowPlot(customer_data, "marital_status", "health_ins",
                       title = "Health insurance status by marital status") <-+
ggplot(customer_data, aes(x=marital_status, fill=health_ins)) +               |
                    geom_bar(position = "fill")  <-+                          |
                                                   |                          |
                                Filled bar chart <-+   Uses the ShadowPlot <---+
                                                       command from the WVPlots
                                                       package for the shadow plot
```

Let's break down each of the R code snippets for visualizing the relationship between marital status and health insurance status in the `customer_data` dataset.

1. Stacked Bar Chart

```
ggplot(customer_data, aes(x = marital_status, fill = health_ins)) +
  geom_bar()
```

- **Purpose:** This code creates a stacked bar chart.
- **Explanation:**
  - `aes(x = marital_status, fill = health_ins)`: Maps the `marital_status` variable to the x-axis and `health_ins` (health insurance status) to the fill color of the bars.
  - `geom_bar()`: By default, `geom_bar()` creates a stacked bar chart where each bar represents a marital status category, and the segments within each bar represent the proportion of insured and uninsured individuals.

2. Side-by-Side Bar Chart

```
ggplot(customer_data, aes(x = marital_status, fill = health_ins)) +
  geom_bar(position = "dodge")
```

- **Purpose:** This code creates a side-by-side bar chart.
- **Explanation:**
  - `aes(x = marital_status, fill = health_ins)`: Maps the `marital_status` variable to the x-axis and `health_ins` to the fill color of the bars.
  - `geom_bar(position = "dodge")`: The `position = "dodge"` argument places the bars for each health insurance status side by side within each marital status category.

3. Shadow Plot

```
ShadowPlot(customer_data, "marital_status", "health_ins", title
= "Health insurance status by marital status")
```

- **Explanation:**
  - `ShadowPlot()`: This function is designed to create two graphs (one for insured and one for uninsured) superimposed against a “shadow” graph of the total population for each marital status category.
  - **Advantages:** This plot allows for comparison both within and across marital status categories while maintaining information about the total population size in each category. It helps in understanding both the proportions and the total numbers.
  - **Disadvantages:** The specific implementation details of `ShadowPlot` are not standard in `ggplot2`, so its effectiveness depends on how well it is implemented.

4. Filled Bar Chart

```
ggplot(customer_data, aes(x = marital_status, fill = health_ins)) +
  geom_bar(position = "fill")
```

- **Explanation:**
  - `aes(x = marital_status, fill = health_ins)`: Maps the `marital_status` variable to the x-axis and `health_ins` to the fill color of the bars.
  - `geom_bar(position = "fill")`: The `position = "fill"` argument normalizes each bar to 100% height, showing the proportion of insured vs. uninsured within each marital status category.
  - **Advantages:** This chart makes it easy to compare the ratios of insured vs. uninsured individuals across categories. The height of each bar is the same, so the proportional differences are clear.
  - **Disadvantages:** It loses the information about the absolute total number of people in each category. Rare categories might not be well-represented in terms of their total count.

Summary:

- **Stacked Bar Chart:** Shows the total number and breakdown of insured vs. uninsured within each category.

- **Side-by-Side Bar Chart:** Facilitates direct comparison of insured vs. uninsured across categories.
- **Shadow Plot:** Combines the benefits of both stacked and side-by-side bar charts, though it depends on a custom or additional function.
- **Filled Bar Chart:** Focuses on the ratio of insured vs. uninsured, but loses absolute count information.

When dealing with multiple categorical variables, especially if each variable has several categories, visualizations can become cluttered. Here's how you can manage such situations effectively in `ggplot2`:

1. Side-by-Side Bar Chart

```
ggplot(data, aes(x = marital_status, fill = housing_type)) +
  geom_bar(position = "dodge")
```

- **Purpose:** This side-by-side bar chart displays the distribution of marital status across different housing types.
- **Disadvantages:** Can become cluttered and hard to read if there are many categories, as each housing type will have a separate set of bars for each marital status.

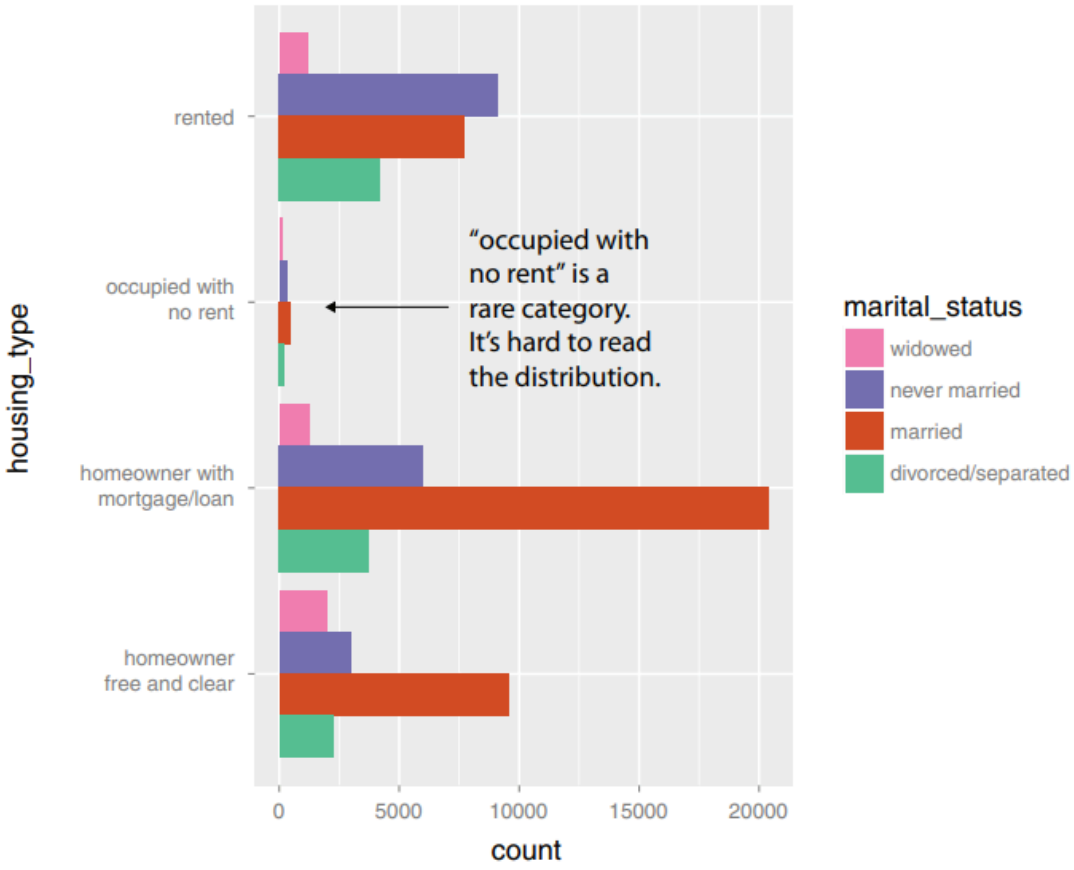


Figure 3.20 Distribution of marital status by housing type: side-by-side bar chart

A graph like figure 3.20 gets cluttered if either of the variables has a large number of categories. A better alternative is to break the distributions into different graphs, one for each housing type. In ggplot2 this is called **faceting the graph**, and you use the **facet_wrap** layer. The result is shown in figure 3.21

Listing 3.16 Plotting a bar chart with and without facets

Side-by-side bar chart

```
cdata <- subset(customer_data, !is.na(housing_type))

ggplot(cdata, aes(x=housing_type, fill=marital_status)) +
   geom_bar(position = "dodge") +
  scale_fill_brewer(palette = "Dark2") +
  coord_flip()
```

Restricts to the data where housing_type is known

Uses coord_flip () to rotate the graph so that marital_status is legible

The faceted bar chart

```
ggplot(cdata, aes(x=marital_status)) +
   geom_bar(fill="darkgray") +
  facet_wrap(~housing_type, scale="free_x") +
   coord_flip()
```

Uses coord_flip() to rotate the graph

Facets the graph by housing.type. The scales="free_x" argument specifies that each facet has an independently scaled x-axis; the default is that all facets have the same scales on both axes. The argument "free_y" would free the y-axis scaling, and the argument "free" frees both axes.

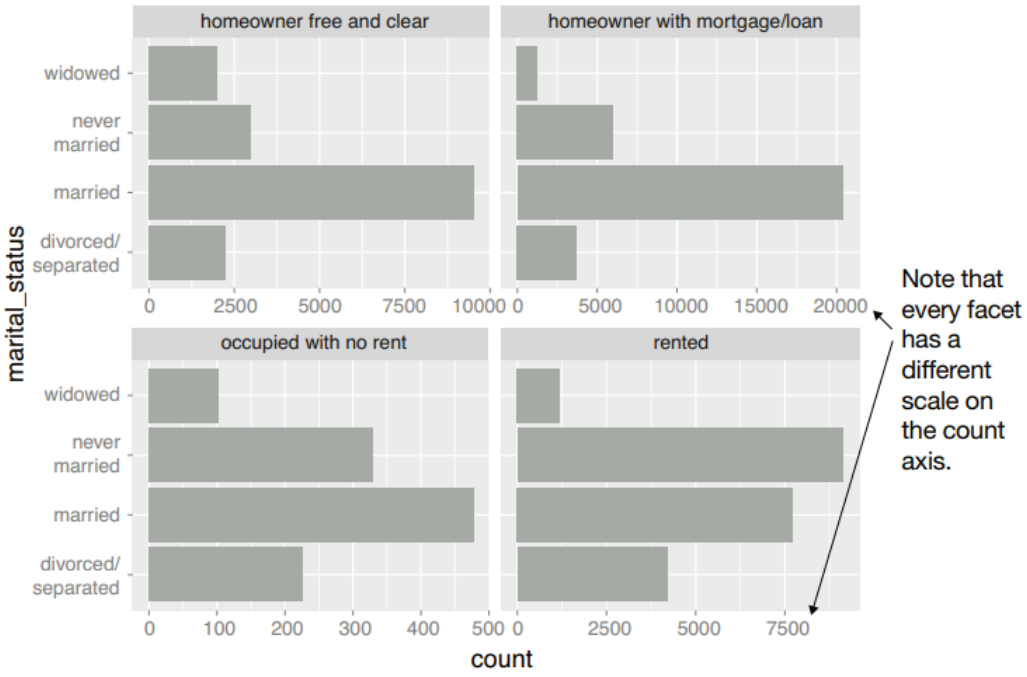


Figure 3.21 Distribution of marital status by housing type: faceted side-by-side bar chart

Faceting

Faceting involves splitting the data into multiple subplots based on one of the categorical variables. In `ggplot2`, this is done using the `facet_wrap()` function.

Advantages of Faceting:

- **Clarity:** Faceting reduces clutter by splitting data into smaller, more manageable plots.
- **Comparability:** Allows easy comparison of distributions across different categories by viewing each subplot individually.
- **Customization:** Each facet can be customized independently, making it easier to highlight specific patterns within subsets of data.

When to Use Faceting:

- **Many Categories:** When you have multiple categories in one or both variables.
- **Complex Data:** When the relationships between categories are complex and need clearer visualization.

Summary

- **Side-by-Side Bar Chart:** Useful for direct comparison across categories but may become cluttered with many categories.
- **Faceting:** Provides a clearer view by breaking down the data into multiple plots based on one categorical variable, making it easier to interpret complex distributions.

```
cdata <- subset(customer_data, !is.na(housing_type))
```

- **Purpose:** This line of code creates a subset of `customer_data`, filtering out any rows where `housing_type` is `NA` (i.e., missing).
- By removing rows with missing `housing_type` values, you ensure that the subsequent plots are based only on complete data, which helps avoid potential issues with visualizations that might arise due to missing data.

Side-by-Side Bar Chart with Flipped Coordinates

```
ggplot(cdata, aes(x = housing_type, fill = marital_status)) +
  geom_bar(position = "dodge") +
  scale_fill_brewer(palette = "Dark2") +
  coord_flip()
```

- **Data and Aesthetics (`aes`)**:
  - `x = housing_type`: Maps the `housing_type` variable to the x-axis.
  - `fill = marital_status`: Fills the bars with different colors based on the `marital_status` variable.
- `geom_bar(position = "dodge")`:
  - Creates a side-by-side (dodge) bar chart where bars for different `marital_status` categories are placed next to each other for each `housing_type`.
  - This allows for easy comparison of marital statuses within each housing type.
- `scale_fill_brewer(palette = "Dark2")`:
  - Applies the `"Dark2"` color palette from the `RColorBrewer` package to differentiate the `marital_status` categories. The palette is chosen to provide distinct, visually appealing colors.
- `coord_flip()`:
  - Flips the x and y axes, making the bars horizontal instead of vertical.
  - Horizontal bars can be easier to read, especially when the category names are long or there are many categories.

Summary of side by side:

This plot shows the distribution of different marital statuses across various housing types. By using side-by-side bars and distinct colors, it allows for straightforward comparison of marital statuses within each housing type. Flipping the axes enhances readability, especially for categorical data with longer labels.

Second Plot: Faceted Bar Chart with Flipped Coordinates

```
ggplot(cdata, aes(x = marital_status)) +
  geom_bar(fill = "darkgray") +
  facet_wrap(~ housing_type, scales = "free_x") +
  coord_flip()
```

- **Data and Aesthetics (`aes`)**:
  - **`x = marital_status`**: Maps the `marital_status` variable to the x-axis.
  - No `fill` aesthetic is specified here, so all bars are uniformly colored.
- **`geom_bar(fill = "darkgray")`**:
  - Creates a bar chart where the bars represent the counts of different `marital_status` categories.
  - **Purpose:** The uniform dark gray color focuses attention on the distribution of marital statuses rather than on the distinction between categories.
- **`facet_wrap(~ housing_type, scales = "free_x")`**:
  - **`facet_wrap(~ housing_type)`**: Splits the plot into separate panels, one for each `housing_type`.
  - **`scales = "free_x"`**: Allows the x-axis (which shows `marital_status`) to have different scales in each facet. This is useful if the distribution of `marital_status` varies widely across housing types.
- **`coord_flip()`**:
  - Flips the axes, making the bars horizontal.
  - **Purpose:** Improves readability, especially when dealing with categorical data and multiple facets.

Summary of Second Plot:

This faceted plot breaks down the distribution of marital statuses by housing type, with each housing type shown in a separate panel. The use of a single color helps to highlight the differences in distributions without distracting from the patterns. The free

x-axis scaling ensures that each housing type is displayed in the most effective way, given the variation in data.

Comparison and Use Cases:

- **First Plot (Side-by-Side Bar Chart with Flipped Coordinates):**
  - **Use Case:** Ideal for comparing the distribution of marital statuses within each housing type. The side-by-side bars and distinct colors make these comparisons clear.
  - **Advantages:** Easy to compare across categories within each housing type. The flipped coordinates make it easier to read.
- **Second Plot (Faceted Bar Chart with Flipped Coordinates):**
  - **Use Case:** Best for comparing how marital status distribution varies across different housing types. Each housing type gets its own panel, making it easier to see differences without clutter.
  - **Advantages:** Provides a clear view of how each housing type is associated with different marital statuses. The free x-axis scaling allows for effective visualization of varying distributions.

Comparing a Continuous and a Categorical Variable

When comparing a continuous variable (like age) across different categories of a categorical variable (like marital status), one effective method is to use density plots. A density plot helps visualize the distribution of the continuous variable, showing where the data is concentrated and how it spreads across different values.

Scenario:

You want to compare the **age distributions** of people with different **marital statuses**. Specifically, you're interested in comparing the age distribution of people who are **widowed** with those who are **never married**.

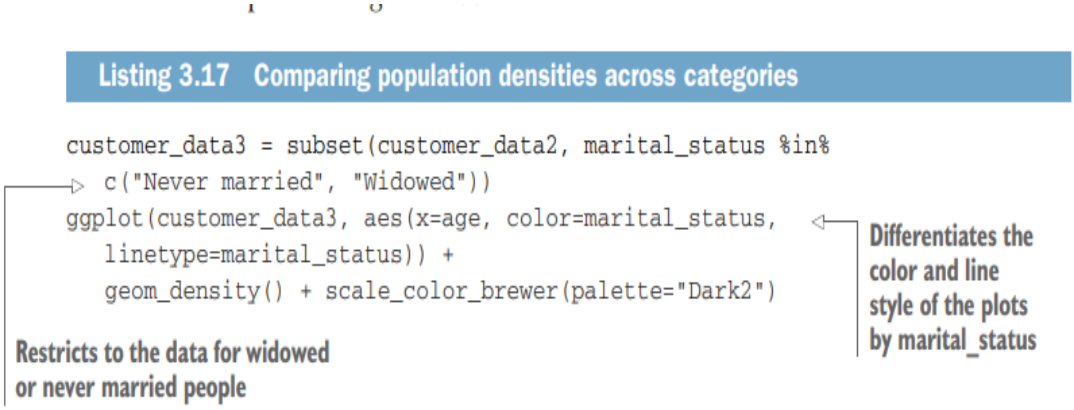
Approach:

- **Superimposed Density Plots**: By overlaying density plots for each marital status category in the same graph, you can visually compare how the distributions

differ. This method allows you to see where the distributions overlap and where they differ, indicating differences in age patterns between the two groups.

Key Insights:

- **Widowed Population**: The age distribution for the widowed group is expected to skew older because people are typically widowed later in life.
- **Never Married Population**: Conversely, the never-married group is likely to skew younger, as younger people are more likely to have never been married.




```
customer_data3 = subset(customer_data2, marital_status %in%
c("Never married", "Widowed"))
```

- **Subset Data**: First, the data is filtered to include only those individuals who are either **never married** or **widowed**. This step ensures that the density plot focuses on the comparison of these two specific groups.

```
ggplot(customer_data3, aes(x=age, color=marital_status,
linetype=marital_status)) +

geom_density() + scale_color_brewer(palette="Dark2")
```

- **ggplot Function**:
  - `aes(x=age, color=marital_status, linetype=marital_status)`: This sets the **age** as the variable for the x-axis. It also maps the **marital status** to both color and line type, which allows the different marital status groups to be distinguished by color and line style in the plot.
  - `geom_density()`: This adds the density plot to the graph, which shows the smoothed distribution of ages for each marital status group.
  - `scale_color_brewer(palette="Dark2")`: This function applies a color palette from the RColorBrewer package to the lines, making the plot more visually appealing and easier to differentiate between groups.

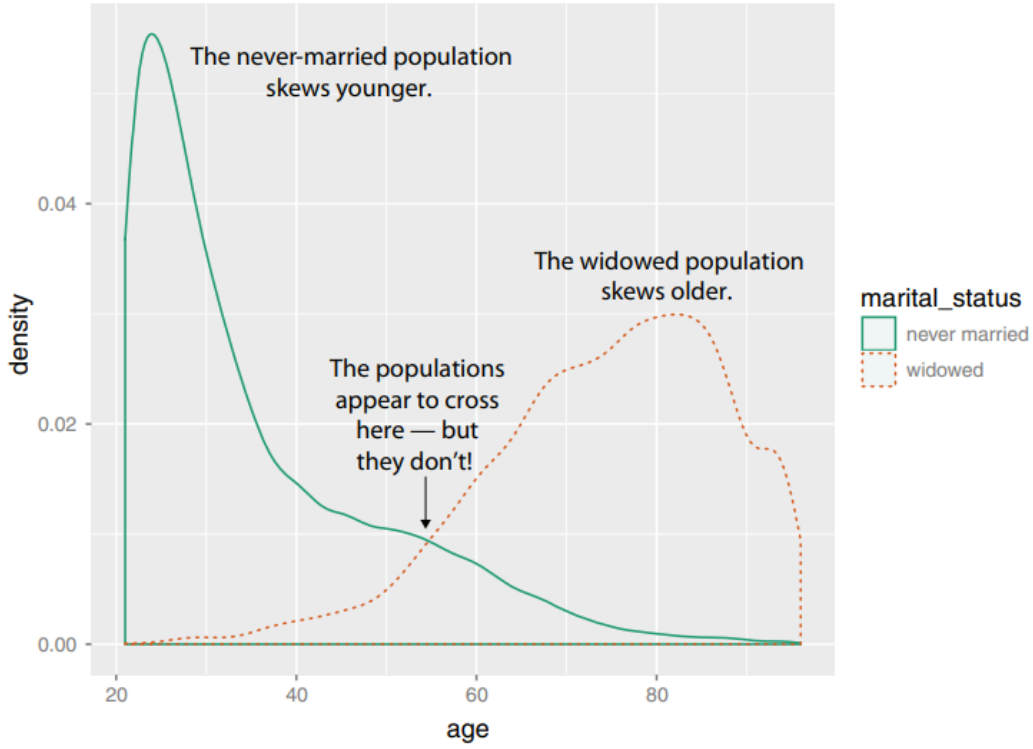


Figure 3.22 Comparing the distribution of marital status for widowed and never married populations

Visualization Result:

- The resulting graph will show two lines, each representing the age distribution of one of the marital status groups:
    - **Dashed Line**: Represents the age distribution of the **widowed** group, likely showing a peak at older ages.
    - **Solid Line**: Represents the age distribution of the **never married** group, likely peaking at younger ages.

Advantages:

- **Distribution Shape**: The superimposed density plots clearly **show the shape of each distribution, indicating where the population is most concentrated.**
- **Comparison of Groups**: The overlay allows **for easy comparison between the groups**, showing how they differ in terms of age distribution.

Limitations:

- **Relative Size**: While this method is excellent for visualizing distribution shape, it doesn't convey the relative size of each group. For example, if one group is much larger than the other, this won't be apparent from the density plot alone.

To maintain information about the relative size of each population, histograms are a better choice than density plots. Since histograms don't overlay well, you can use the `facet_wrap()` command with `geom_histogram()` to create separate histograms for each group, similar to faceted bar charts. Another option is to use the `ShadowHist()` function from the WVPlots package, which provides a shadow plot version of the histogram, allowing for comparison while retaining the sense of population size.

Listing 3.18 Comparing population densities across categories with `ShadowHist()`

```
ShadowHist(customer_data3, "age", "marital_status",
 "Age distribution for never married vs. widowed populations", binwidth=5)
```

Sets the bin widths of the histogram to 5

The `ShadowHist()` function from the WVPlots package creates a histogram-based shadow plot, which is useful for comparing distributions across categories while retaining a sense of the relative size of each group.

```
ShadowHist(customer_data3, "age", "marital_status", "Age distribution for never married vs. widowed populations", binwidth=5)
```

1. **`customer_data3`:**
    - This is the dataset being used, which contains information on individuals, including their age and marital status.
    - In this case, the dataset has been filtered to include only those who are never married or widowed.
2. **`"age"`:**
    - This is the continuous variable you want to plot, representing the age of individuals.
    - The histogram will show the distribution of ages within each marital status category.
3. **`"marital_status"`:**
    - This is the categorical variable that separates the data into different groups.
    - The plot will compare the age distributions between the never married and widowed categories**.**
4. **`"Age distribution for never married vs. widowed populations"`:**
    - This is the title of the plot. It will appear at the top of the chart, providing context for what the plot is showing.
5. **`binwidth=5`:**
    - The `binwidth` parameter specifies the width of the bins in the histogram.
    - A binwidth of 5 means that each bar in the histogram represents an age range of 5 years (e.g., 20-25, 25-30, etc.).
    - This helps smooth out the histogram and makes patterns in the data more apparent.

What the Plot Shows:

- The ShadowHist plot will display two histograms, one for each marital status group (never married and widowed).
- The histograms are overlaid with a shadow that represents the total population, allowing you to see both the distribution within each group and how the groups compare in size.
- This visualization helps in understanding both the shape of the distribution and the relative size of the populations, addressing the limitation of overlaid density plots.

The result is shown in figure 3.23. Now you can see that the widowed population is quite small, and doesn't exceed the never married population until after about age 65—10 years later than the crossover point in figure 3.22.

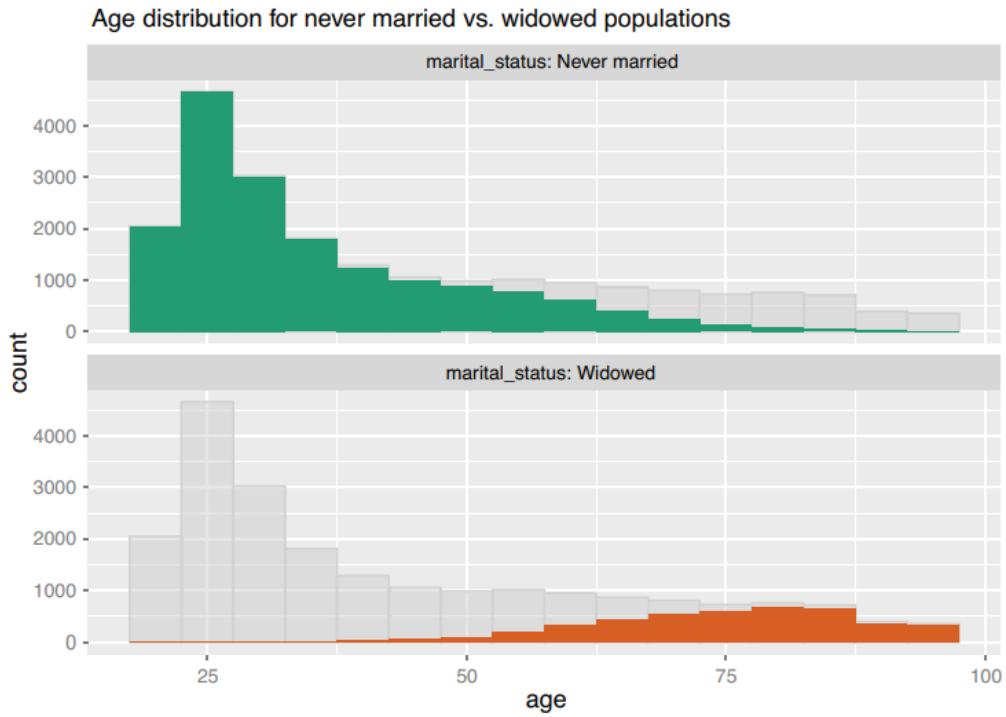


Figure 3.23 `ShadowHist` **comparison of the age distributions of widowed and never married populations**

When comparing distributions across more than two categories, using faceting is a practical approach to ensure clarity in your visualizations. Faceting allows you to split the data into separate panels, each showing the distribution for one category, rather

than overlaying all distributions in a single plot, which can become cluttered and difficult to interpret.

Function Call:

```
ggplot(customer_data2, aes(x=age)) +
  geom_density() + facet_wrap(~marital_status)
```


```
ggplot(customer_data2, aes(x=age)) +

geom_density() + facet_wrap(~marital_status)
```


1. **`ggplot(customer_data2, aes(x=age))`:**
    - Data: `customer_data2` is the dataset being used, which contains information about individuals, including their age and marital status.
    - Mapping: `aes(x=age)` maps the `age` variable to the x-axis of the plot. This sets up the plot to show the distribution of ages within the dataset.
2. **`geom_density()`:**
    - This adds a density plot to the graph. A density plot is a smoothed version of a histogram, showing the distribution of the age variable as a continuous curve.
    - The curve illustrates where the data points are concentrated (peaks) and where they are sparse (valleys).
3. **`facet_wrap(~marital_status)`:**
    - Faceting: The `facet_wrap(~marital_status)` function splits the data into multiple panels, with each panel representing a different category of the `marital_status` variable.
    - Panels: In this case, you'll get separate density plots for each category of marital status (e.g., Never Married, Married, Divorced, Widowed).
    - Layout: The panels are arranged in a grid, making it easy to compare the age distributions across all marital status categories without overlaying them in a single plot.

What the Plot Shows:

- Separate Panels: Each marital status category will have its own panel, displaying the age distribution for that specific group.
- Clear Comparison: By faceting, you avoid the confusion that can arise from overlaid density plots, especially when there are more than two categories. This makes it easier to compare how age is distributed within each marital status category.
- Interpretation: You can easily observe how the distribution of age varies across different marital statuses. For instance, one might notice that the age distribution for "Widowed" skews older, while "Never Married" skews younger.

Advantages:

- Clarity: Faceting keeps the visualization clear and interpretable, even when comparing multiple groups.
- Focus: Each panel allows you to focus on one category at a time, while still seeing the others in the context of the overall dataset.

This approach is particularly useful when dealing with multiple categories, as it provides a clean and organized way to compare distributions without losing important details or overwhelming the viewer.

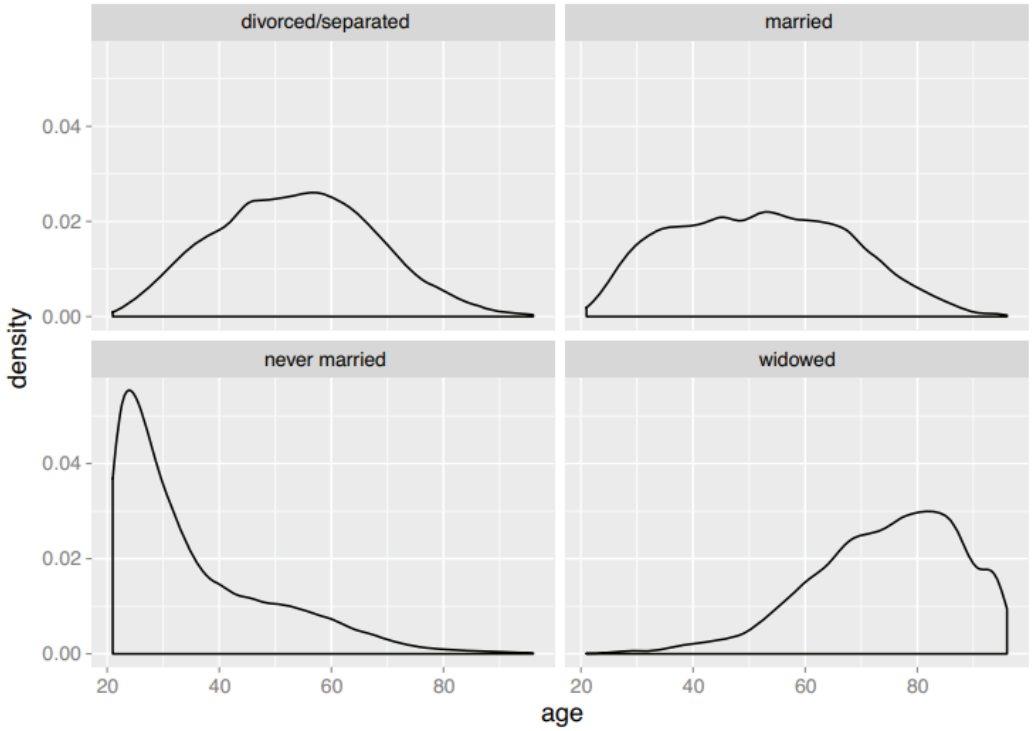


Figure 3.24 Faceted plot of the age distributions of different marital statuses

Density Plot:

```
ggplot(customer_data3, aes(x=age, color=marital_status,
linetype=marital_status)) + geom_density() +
scale_color_brewer(palette="Dark2")
```

Histogram with Faceting:

```
ggplot(customer_data2, aes(x=age)) + geom_density() +
facet_wrap(~marital_status)
```

Each type of bar chart or distribution plot has its strengths and weaknesses, and the choice depends on what aspect of the data you want to emphasize—whether it's total counts, proportions, or distributions within and across categories.

OVERVIEW OF VISUALIZATIONS FOR TWO VARIABLES

Table 3.3 summarizes the visualizations for two variables that we've covered.

Table 3.3 Visualizations for two variables

Graph type	Uses	Examples
Line plot	Shows the relationship between two continuous variables. Best when that relationship is functional, or nearly so.	Plot $y = f(x)$.
Scatter plot	Shows the relationship between two continuous variables. Best when the relationship is too loose or cloud-like to be easily seen on a line plot.	Plot income vs. years in the workforce (income on the y-axis).

Table 3.3 Visualizations for two variables *(continued)*

Graph type	Uses	Examples
Smoothing curve	Shows underlying "average" relationship, or trend, between two continuous variables. Can also be used to show the relationship between a continuous and a binary or Boolean variable: the fraction of `true` values of the discrete variable as a function of the continuous variable.	Estimate the "average" relationship of income to years in the workforce.
Hexbin plot	Shows the relationship between two continuous variables when the data is very dense.	Plot income vs. years in the workforce for a large population.
Stacked bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Highlights the frequencies of each value of `var1`. Works best when `var2` is binary.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to retain information about the number of people in each marital category.
Side-by-side bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Good for comparing the frequencies of each value of `var2` across the values of `var1`. Works best when `var2` is binary.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to directly compare the number of insured and uninsured people in each marital category.
Shadow plot	Shows the relationship between two categorical variables (`var1` and `var2`). Displays the frequency of each value of `var1`, while allowing comparison of `var2` values both within and across the categories of `var1`.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to directly compare the number of insured and uninsured people in each marital category *and* still retain information about the total number of people in each marital category.

Filled bar chart	Shows the relationship between two categorical variables (`var1` and `var2`). Good for comparing the relative frequencies of each value of `var2` within each value of `var1`. Works best when `var2` is binary.	Plot insurance coverage (`var2`) as a function of marital status (`var1`) when you wish to compare the *ratio* of uninsured to insured people in each marital category.
Bar chart with faceting	Shows the relationship between two categorical variables (`var1` and `var2`). Best for comparing the relative frequencies of each value of `var2` within each value of `var1` when `var2` takes on more than two values.	Plot the distribution of marital status (`var2`) as a function of housing type (`var1`).

Table 3.3 Visualizations for two variables *(continued)*

Graph type	Uses	Examples
Overlaid density plot	Compares the distribution of a continuous variable over different values of a categorical variable. Best when the categorical variable has only two or three categories. Shows whether the continuous variable is distributed differently or similarly across the categories.	Compare the age distribution of married vs. divorced populations.
Faceted density plot	Compares the distribution of a continuous variable over different values of a categorical variable. Suitable for categorical variables with more than three or so categories. Shows whether the continuous variable is distributed differently or similarly across the categories.	Compare the age distribution of several marital statuses (never married, married, divorced, widowed).
Faceted histogram or shadow histogram	Compares the distribution of a continuous variable over different values of a categorical variable while retaining information about the relative population sizes.	Compare the age distribution of several marital statuses (never married, married, divorced, widowed), while retaining information about relative population sizes.