Statistics with Rust

50+ Statistical Techniques Put into Action

Keiko Nakamura

Published by: GitforGits
Publisher: Sonal Dhandre
www.gitforgits.com
support@gitforgits.com

Printed in India

First Printing: April 2023

Cover Design by: Kitten Publishing

Content

Preface

Are you an experienced statistician or data professional looking for a powerful, efficient, and versatile programming language to turbocharge your data analysis and machine learning projects? Look no further! "Statistics with Rust" is your comprehensive resource to unlock Rust's true potential in modern statistical methods.

This book is tailored specifically for statisticians and data professionals who are already familiar with the fundamentals of statistics and want to leverage the speed and reliability of Rust in their projects. Over 11 in-depth chapters, you will discover how Rust outperforms Python in various aspects of data analysis and machine learning and learn to implement popular statistical methods using Rust's unique features and libraries.

"Statistics with Rust" begins by introducing you to Rust's programming environment and essential libraries for data professionals. You'll then dive into data handling, preprocessing, and visualization techniques that form the backbone of any statistical analysis. As you progress through the book, you'll explore descriptive and inferential statistics, probability distributions, regression analysis, time series analysis, Bayesian statistics, multivariate statistical methods, and nonlinear models. Additionally, the book covers essential machine-learning techniques, model evaluation and validation, natural language processing, and advanced techniques in emerging topics.

In this book you will learn how to:

Discover Rust's unique advantages for statistical analysis and machine learning projects.

Learn to efficiently handle, preprocess, and visualize data using Rust libraries.
Implement descriptive and inferential statistics with Rust for powerful data insights.
Master probability distributions and random variables in Rust for robust simulations.
Perform advanced regression analysis with Rust's capabilities.
Explore Bayesian statistics and Markov Chain Monte Carlo methods in Rust.
Uncover multivariate techniques, including PCA and Factor Analysis, using Rust libraries.
Implement cutting-edge machine learning algorithms and model evaluation techniques in Rust.
Delve into text analysis, and natural language processing with Rust.

To ensure you get the most out of this book, each chapter includes hands-on examples and exercises to reinforce your understanding of the concepts presented. You'll also learn to optimize your Rust code and select the best tools and libraries for each task, maximizing your productivity and efficiency.

GitforGits

Prerequisites

"Statistics with Rust" is your indispensable guide to harnessing the power of Rust for modern statistical analysis and machine learning. Whether you are a seasoned data professional or a Rust enthusiast looking to expand your knowledge, this book provides the tools and insights to elevate your projects.

Codes Usage

Acknowledgement

I owe a tremendous debt of gratitude to GitforGits, for their unflagging enthusiasm and wise counsel throughout the entire process of writing this book. Their knowledge and careful editing helped make sure the piece was useful for people of all reading levels and comprehension skills. In addition, I'd like to thank everyone involved in the publishing process for their efforts in making this book a reality. Their efforts, from copyediting to advertising, made the project what it is today.

Finally, I'd like to express my gratitude to everyone who has shown me unconditional love and encouragement throughout my life. Their support was crucial to the completion of this book. I appreciate your help with this endeavour and your continued interest in my career.

Chapter 1: Introduction to Rust for Statisticians

Why Rust for Data Analysis and Statistics?

In recent years, the Rust programming language has attracted considerable attention from developers for its safety, speed, and concurrency capabilities. Originating as a systems programming language, Rust has grown in popularity and has been adopted across various domains, including web development, embedded systems, and even data analysis. With its focus on performance and safety, Rust is a formidable choice for data analysis and statistical computing, providing unique advantages over traditional languages such as Python, R, and Julia.

This book aims to guide you through the world of statistics and data analysis using Rust, offering a comprehensive understanding of Rust's potential in these fields. By the end of this journey, you will be equipped with the knowledge and practical skills to leverage Rust's power for your data analysis projects.

Rust's high-performance capabilities are one of its most appealing features. As a compiled language, Rust offers performance that is on par with or even surpasses C and C++. This is particularly important for data analysis and statistics, where large datasets and complex computations are common. With Rust, you can execute data processing tasks and run algorithms with lower latency, enabling faster and more efficient analysis.

Memory safety is a critical aspect of any programming language, especially when dealing with large datasets or complex data structures. Rust's unique ownership system and strong type system ensure memory safety at compile time, eliminating common bugs such as data races, null pointer dereferences, and buffer overflows. This guarantees that your data analysis programs will be more robust and less prone to crashes, without the need for a garbage collector that might impact performance.

Modern hardware often features multiple cores or processors, and utilizing this parallelism is essential for high-performance computing. Rust's built-in concurrency support, based on its ownership and borrowing system, allows you to build concurrent and parallel programs with ease. By leveraging Rust's concurrency features, you can efficiently distribute data processing tasks across multiple cores or even multiple machines, significantly reducing the time required for complex calculations.

Rust's C-compatible FFI (Foreign Function Interface) enables seamless integration with existing C and C++ libraries. This means you can easily use existing high-performance libraries for data analysis, such as BLAS (Basic Linear Algebra Subprograms), LAPACK (Linear Algebra PACKage), or FFTW (Fastest Fourier Transform in the West), alongside Rust's native libraries. Moreover, Rust's WebAssembly support allows you to run your data analysis code on the web, opening up new possibilities for interactive data visualization and analysis tools.

Although Rust is a relatively young language, its ecosystem has grown rapidly, with an ever-increasing number of libraries and tools catering to data analysis and statistics. Libraries such as ndarray, statrs, and plotly offer robust support for data manipulation, statistical computation, and visualization. Additionally, the Rust community is highly active and committed to developing new libraries and improving existing ones, ensuring that the Rust ecosystem will continue to expand and evolve.

Rust's syntax is clear, concise, and expressive, making it easier for you to write and read your code. This improves the maintainability of your data analysis programs, allowing you to quickly identify and fix issues or add new features. Moreover, Rust's strong type system and the Rust compiler's helpful error messages significantly reduce the likelihood of introducing bugs, resulting in higher-quality code.

Comparing Rust and Python for Statistics

Python has long been a popular choice for data analysis and statistics due to its simplicity, vast ecosystem, and numerous libraries such as NumPy, pandas, and SciPy. However, as Rust gains traction and its ecosystem grows, it is becoming an increasingly viable alternative for statistics and data analysis tasks. In this section, we will compare Rust and Python in terms of their suitability for statistical computing and highlight the advantages that Rust brings to the table.

Performance

One of the most significant differences between Rust and Python lies in their performance. Python, as an interpreted language, has inherent performance limitations that may become a bottleneck for large-scale data analysis or computationally intensive tasks. While Python offers tools like NumPy and Cython to mitigate these performance issues, they come with their own learning curve and may not always provide the desired speedup.

On the other hand, Rust is a compiled language with performance on par with or even surpassing C and C++. This makes Rust an excellent choice for high-performance computing, where complex calculations and large datasets are common. By using Rust, you can achieve faster data processing times and reduced latency, allowing you to analyze larger datasets and perform more complex calculations in less time.

Memory Safety and Resource Management

Python relies on a garbage collector to manage memory, which can introduce latency and unpredictability in performance-sensitive applications. Additionally, while Python's dynamic typing and garbage collection make it easy to work with, they can also lead to runtime errors and memory leaks if not managed carefully.

Rust's strong static type system and unique ownership model ensure memory safety at compile time, eliminating common bugs such as data races, null pointer dereferences, and buffer overflows. This results in more reliable and robust data analysis programs, with fewer runtime errors and crashes. Rust's manual memory management also provides fine-grained control over resource usage, enabling you to optimize your programs for specific performance or memory constraints.

Concurrency

Python's Global Interpreter Lock (GIL) restricts multi-threading and can hinder the efficient utilization of modern multi-core hardware. While Python provides alternative concurrency models such as multiprocessing and asynchronous programming, these approaches have their own limitations and complexities.

Rust's built-in concurrency support, based on its ownership and borrowing system, allows for more straightforward and efficient parallelism. With Rust, you can easily build concurrent and parallel programs that take full advantage of modern multi-core hardware. This enables you to process data and perform complex calculations significantly faster than with

Python, especially when working with large datasets or computationally intensive tasks.

Interoperability

Python is known for its extensive library ecosystem, which includes many high-performance libraries for data analysis and statistics. Rust's C-compatible FFI allows you to leverage these existing libraries, ensuring that you do not lose access to the tools you are familiar with when transitioning to Rust. This interoperability also extends to C and C++ libraries, further expanding the range of libraries and tools available for your data analysis projects.

Ecosystem Growth and Future Prospects

While Python's ecosystem for data analysis and statistics is well-established and mature, Rust's ecosystem is growing rapidly. Libraries such as ndarray, statrs, and plotly provide robust support for data manipulation, statistical computation, and visualization. As the Rust community continues to develop new libraries and improve existing ones, Rust's capabilities for data analysis and statistics will only increase.

Readability and Maintainability

Python's simplicity and readability are some of its greatest strengths, making it easy for developers to write and maintain code. Rust, while having a steeper learning curve, also emphasizes readability and maintainability through its clear, concise, and expressive syntax. Rust's strong type system and the Rust compiler's helpful error messages significantly reduce the likelihood of introducing bugs, resulting in high-quality code that is easier to maintain and debug.

Scalability

As data analysis projects grow in size and complexity, the need for a language that can scale with the demands becomes crucial. Rust's performance, concurrency support, and memory safety features make it an ideal choice for scalable data analysis applications. With Rust, you can efficiently process large datasets, distribute workloads across multiple cores or machines, and build fault-tolerant systems that remain performant and reliable as they grow.

Cross-platform and Deployment

Both Rust and Python are cross-platform languages that can run on various operating systems, making them suitable for a wide range of applications. However, Rust's compiled binaries can offer advantages in deployment scenarios. Rust programs can be easily compiled into standalone binaries, eliminating the need for a runtime interpreter or complex dependencies. This can make deploying Rust applications in production environments more straightforward and less prone to runtime issues compared to Python.

Learning Curve

Python's simplicity and ease of use make it an excellent choice for beginners and non-programmers. However, Rust's steeper learning curve, while initially more challenging, can lead to more robust and maintainable code in the long run. As you become familiar with Rust's strong type system, ownership model, and memory safety features, you will develop a deeper understanding of programming concepts that can improve your overall programming skills.

While Python remains a popular and powerful language for data analysis and statistics, Rust offers several advantages that make it a compelling alternative. Rust's high performance, memory safety, concurrency support, and growing ecosystem position it as a strong contender for future data analysis tasks. By choosing Rust for your statistical computing projects, you can leverage its unique features to build fast, reliable, and scalable applications that are well-suited for the increasing demands of data-driven industries.

Setting up Rust Environment

To install Rust, you can follow these step-by-step instructions. The installation process is straightforward and utilizes a tool called "rustup," which manages Rust versions and associated tools for you.

Download rustup-init

Visit the official Rust website (https://www.rust-lang.org/tools/install) to download the rustup-init executable for your platform (Windows, macOS, or Linux). The website should automatically detect your operating system and provide the appropriate download link.

Run rustup-init

Once you have downloaded rustup-init, follow the instructions specific to your operating system.

For Windows
Locate the downloaded rustup-init.exe file in your Downloads folder.

Double-click the file to run the installer.

A command prompt window will open, asking you to proceed with the installation. Press "Enter" to continue with the default installation, or follow the on-screen instructions to customize the installation.

For macOS and Linux
Open a terminal window.

Navigate to the directory where you downloaded the rustup-init file (typically the Downloads folder).

Make the file executable by running the following command: chmod +x rustup-init.

Execute the rustup-init script by running ./rustup-init.

The script will prompt you to proceed with the installation. Press "Enter" to continue with the default installation, or follow the on-screen instructions to customize the installation.

Configure PATH Environment Variable

During the installation, rustup will attempt to configure your PATH environment variable to include the Rust toolchain. In most cases, this will happen automatically.

For Windows
Close the command prompt window and open a new one.

Run the command rustc --version. If the installation was successful, you should see the Rust compiler version displayed.

For macOS and Linux
Run the command source $HOME/.cargo/env to update the current terminal session with the new PATH settings.

Run the command rustc --version. If the installation was successful, you should see the Rust compiler version displayed.

Note: For macOS and Linux, you may need to add the following line to your shell profile file (e.g., ~/.bashrc, ~/.bash_profile, or ~/.zshrc) to ensure the PATH is configured correctly for future terminal sessions:

```
export PATH="$HOME/.cargo/bin:$PATH"
```

Verify the Installation

To verify that Rust has been installed correctly, open a terminal or command prompt window and run the following command:

```
rustc --version
```

If the installation was successful, you should see the Rust compiler version displayed.

Essential Rust Libraries for Statistics

There are several Rust libraries that are stable and useful for performing various statistical operations and data analysis tasks. In this introduction, we will cover four essential libraries that will help you get started with implementing statistical functions in Rust: ndarray, statrs, statis, and plotly.

ndarray

The ndarray library is a versatile and powerful crate for handling n-dimensional arrays in Rust. It is inspired by NumPy and provides similar functionality. The ndarray crate is essential for any data analysis project, as it offers efficient array operations and various linear algebra functions.

It is available in following GitHub repository:
https://github.com/rust-ndarray/ndarray

Features of ndarray

- Support for multi-dimensional arrays with shape and strides.
- Efficient operations, including element-wise, broadcasting, and linear algebra operations.
- Various array manipulation methods, such as slicing, stacking, reshaping, and concatenation.
- Interoperability with BLAS and LAPACK libraries for high-performance linear algebra computations.

statrs

Statrs is a comprehensive statistics library for Rust that provides a wide range of statistical functions, probability distributions, and other mathematical tools. It aims to deliver a high-quality, easy-to-use, and well-documented interface for performing statistical computations in Rust.

It is available in following GitHub repository:
https://github.com/boxtown/statrs

Features of statrs

- Support for common probability distributions, such as Normal, Poisson, Bernoulli, and more.
- Various descriptive statistics functions, like mean, variance, standard deviation, skewness, and kurtosis.
- Hypothesis testing functions, such as t-test, chi-square test, and F-test.
- Correlation and regression analysis functions.

statis

Statis is a library focused on providing efficient implementations of statistical algorithms and data structures in Rust. While not as comprehensive as statrs, statis offers a more focused set of tools for specific statistical tasks.

It is available in following GitHub repository:
https://github.com/dylanede/statis

Features of statis

- Data structures for handling histograms, frequency tables, and probability mass functions.
- Support for basic descriptive statistics, such as mean, median, mode, and standard deviation.

Efficient implementation of various statistical algorithms, including order statistics, kernel density estimation, and k-means clustering.

plotly

Plotly is a popular library for creating interactive and visually appealing plots in various programming languages, including Rust. The plotly crate for Rust provides a convenient and high-level API for creating Plotly.js-compatible plots, which can be rendered in a web browser or saved to a file.

It is available in following GitHub repository:
https://github.com/igiagkiozis/plotly

Features of plotly

- Support for various plot types, such as scatter, line, bar, pie, and more.
- Customizable plot aesthetics, such as colors, markers, and axis labels.
- Interactive plot features, including zooming, panning, and hover tooltips.
- Ability to save plots as HTML, SVG, or PNG files.

By combining these libraries, you can create a robust environment for performing statistical analysis and visualizing your data using Rust. These libraries offer a solid foundation for implementing a wide range of statistical functions and provide a flexible and efficient alternative to more traditional data analysis tools like Python, R, or Julia.

Setting up Statistical Project

To use the ndarray, statrs, statis, and plotly libraries in your Rust project, you'll need to add them as dependencies in your project's Cargo.toml file. Following is a step-by-step walkthrough on how to set up a new Rust project and install these libraries.

Create a New Rust Project

Open a terminal or command prompt, navigate to the directory where you want to create your project, and run the following command:

```
cargo new my_statistics_project
```

This will create a new Rust project called "my_statistics_project" with the following structure:

```
my_statistics_project/

  ├── Cargo.toml

  └── src/

      └── main.rs
```

Add Library Dependencies

Open the Cargo.toml file in the my_statistics_project directory using your favorite text editor.

You will see a section called [dependencies]. Under this section, add the following lines to include the ndarray, statrs, statis, and plotly libraries as dependencies:

```
[dependencies]

ndarray = "0.15"

statrs = "0.14"

statis = "0.1"

plotly = "0.6"
```

The numbers after the equals sign represent the version of each library. You can check for the latest versions of these libraries on their respective GitHub repositories or by searching for them on crates.io.

Build and Run the Project

In the terminal or command prompt, navigate to the my_statistics_project directory and run the following command:

```
cargo build
```

This command will download and compile the specified dependencies for your project. Once the build is complete, you can start using the libraries in your Rust code.

Import the Libraries in Rust Code

Open the src/main.rs file in your project directory and import the libraries by adding the following lines at the beginning of the file:

```
extern crate ndarray;

extern crate statrs;

extern crate statis;

extern crate plotly;
```

Now you're ready to use these libraries in your Rust project.

To get started with using each library, refer to their respective documentation for examples, usage guides, and API references.

ndarray: https://docs.rs/ndarray/0.15.4/ndarray/
statrs: https://docs.rs/statrs/0.14.0/statrs/
statis: https://docs.rs/statis/0.1.3/statis/
plotly: https://docs.rs/plotly/0.6.0/plotly/

Summary

This chapter began with a discussion about the benefits of using Rust for data analysis and statistics compared to Python. Rust offers advantages in performance, memory safety, concurrency, interoperability, ecosystem growth, readability, maintainability, scalability, and deployment. While Python remains a popular choice for data analysis, Rust's unique features make it an increasingly attractive option for future data analysis tasks.

Next, we covered the installation process for Rust using the rustup tool. The step-by-step instructions included downloading rustup-init, running the installer, configuring the PATH environment variable, and verifying the installation by checking the Rust compiler version. With Rust installed, you can begin exploring Rust programming for data analysis and statistics projects.

We then introduced four essential Rust libraries for performing statistical operations and data analysis: ndarray, statrs, statis, and plotly. The ndarray library is designed for handling n-dimensional arrays and offers efficient array operations and various linear algebra functions. Statrs provides a comprehensive set of statistical functions, probability distributions, and mathematical tools. Statis is a library focused on efficient implementations of statistical algorithms and data structures. Finally, plotly is a popular library for creating interactive and visually appealing plots.

To install these libraries in your Rust environment, you need to create a new Rust project using Cargo, add the libraries as dependencies in the Cargo.toml file, build the project, and import the libraries in your Rust code. After setting up the project and installing the libraries, you can refer to their respective documentation for examples, usage guides, and API references.

In summary, Rust is emerging as a powerful alternative to Python for data analysis and statistics due to its unique features and growing ecosystem. By using Rust and the recommended libraries (ndarray, statrs, statis, and plotly), you can build fast, reliable, and scalable applications for data analysis. The installation process for Rust and its libraries is straightforward, allowing you to quickly start exploring and implementing statistical functions using Rust.

Chapter 2: Data Handling and Preprocessing

Data Handling and Preprocessing

Data handling and preprocessing are crucial steps in the data analysis pipeline. They involve collecting, cleaning, transforming, and organizing raw data to prepare it for further analysis or processing. These steps ensure that the data is accurate, consistent, and suitable for use in machine learning algorithms, statistical models, or visualization tools.

Process of Data Handling and Preprocessing

Conceptually, data handling and preprocessing encompass several tasks, including:
Data collection: Acquiring and gathering data from various sources, such as databases, APIs, sensors, or user-generated content.
Data cleaning: Identifying and correcting errors, inconsistencies, and inaccuracies in the data. This may involve handling missing values, removing duplicates, correcting typos, or standardizing formats.
Data transformation: Converting data into a suitable format or structure for subsequent analysis. Examples of data transformations include normalization, scaling, encoding categorical variables, and aggregating data at different levels of granularity.
Data integration: Combining data from multiple sources and ensuring consistency and compatibility among different datasets. This may require resolving conflicts in data formats, units, or variable names.

Feature engineering: Deriving new features or variables from the raw data that can improve the performance of machine learning algorithms or provide additional insights during analysis.

Data splitting: Dividing data into training, validation, and test sets for machine learning algorithms to prevent overfitting and evaluate model performance accurately.

Exploring CSV crate

In Rust, the csv crate is a popular and efficient library for handling and preprocessing CSV (Comma Separated Values) files, which are commonly used for storing tabular data. The csv crate provides a high-performance, flexible, and user-friendly API for reading and writing CSV data. Some features of the csv crate include:
Reading and writing CSV data with a variety of configurations, such as custom delimiters, quoting rules, and handling of headers.
Support for streaming large CSV files with a low memory footprint, enabling efficient processing of large datasets.
Serde integration for deserializing and serializing CSV records directly into Rust structs or other data structures.
Comprehensive error handling, making it easier to detect and manage issues during CSV parsing or writing.

Dataset Loading with CSV crate

Let us consider a dataset on Data Scientist having their salaries recorded along with various details recorded. The raw github data is accessible on the following link:
https://raw.githubusercontent.com/kittenpub/database-repository/main/ds_salaries.csv

A brief summary of the dataset shows the Salaries of Different Data Science Fields in the Data Science Domain and contains 11 columns, each are:

work_year: The year the salary was paid.
experience_level: The experience level in the job during the year
employment_type: The type of employment for the role
job_title: The role worked during the year.
salary: The total gross salary amount paid.
salary_currency: The currency of the salary paid as an ISO 4217 currency code.
salaryinusd: The salary in USD
employee_residence: Employee's primary country of residence during the work year as an ISO 3166 country code.
remote_ratio: The overall amount of work done remotely
company_location: The country of the employer's main office or contracting branch
company_size: The median number of people that worked for the company during the year.

To load and process the dataset in Rust, we'll be using the csv crate for reading CSV files and reqwest crate for fetching the dataset from the provided URL. Make sure you have these crates added to your Cargo.toml:

```
[dependencies]

csv = "1.1"

reqwest = { version = "0.11", features = ["blocking"] }
```

```
serde = { version = "1.0", features = ["derive"] }
```

Now let's define the structure for the dataset:

```
use serde::Deserialize;

#[derive(Debug, Deserialize)]

struct SalaryRecord {

  work_year: i32,

  experience_level: String,

  employment_type: String,

  job_title: String,

  salary: f64,

  salary_currency: String,

  salaryinusd: f64,

  employee_residence: String,
  remote_ratio: f64,

  company_location: String,

  company_size: i32,

}
```

Next, we will create a function to fetch the dataset from the provided URL and return the contents as a String:

```
use reqwest::blocking::get;

use std::io::Read;

fn fetch_dataset(url: &str) -> ResultBoxstd::error::Error>> {

  let mut response = get(url)?;

  let mut content = String::new();

  response.read_to_string(&mut content)?;

  Ok(content)

}
```

Now we can create a function to load the dataset into a Vec:

```
use csv::ReaderBuilder;

use std::error::Error;

fn load_dataset(csv_data: &str) -> Result, BoxError>> {

  let mut reader =
ReaderBuilder::new().from_reader(csv_data.as_bytes());

  let mut records = Vec::new();

  for result in reader.deserialize() {

     let record: SalaryRecord = result?;

     records.push(record);

  }

  Ok(records)

}
```

Finally, in the main function, fetch and load the dataset:

```
fn main() {

  let url = "https://raw.githubusercontent.com/kittenpub/database-
repository/main/ds_salaries.csv";

  match fetch_dataset(url) {

      Ok(csv_data) => {

          match load_dataset(&csv_data) {

              Ok(dataset) => {

                  // The dataset is ready for processing

                  println!("Loaded {} records", dataset.len());

              }

              Err(error) => {

                  eprintln!("Error loading dataset: {}", error);

              }

          }
```

```
        }

        Err(error) => {

            eprintln!("Error fetching dataset: {}", error);

        }

    }

}
```

Now you have the dataset loaded as a Vec, and you can perform further operations on it.

Parsing the Data

Data parsing is the process of interpreting and converting raw data into a structured format that can be easily understood and manipulated by computers or humans. In the context of datasets, parsing often involves extracting relevant information from a file, transforming it into a standardized format, and organizing it into a data structure suitable for further processing, analysis, or visualization.

In the previous section, we have already performed data parsing by reading the CSV file and deserializing its contents into a Vec. The csv crate takes care of parsing the CSV data, and the serde crate is responsible for deserializing the parsed data into the SalaryRecord struct.

Now, let's say you want to filter the dataset based on a specific condition or modify certain fields of the records. You can do this by parsing the data further using Rust's iterator and pattern matching capabilities. For example, let's say you want to filter the dataset to only include records where the experience_level is "Senior" and convert the salaryinusd field to integers for easier processing. You can achieve this as follows:

```
fn filter_and_convert(dataset: &[SalaryRecord]) -> Vec<(i32, String, f64)> {

  dataset

    .iter()
```

```
        .filter(|record| record.experience_level == "Senior")

        .map(|record| {

            let salary_in_usd_rounded = record.salaryinusd.round();

            (


                record.work_year,

                record.job_title.clone(),

                salary_in_usd_rounded,

            )

        })

        .collect()

}
```

In the function above, we use the filter method to only include records with the "Senior" experience level. Then, we use the map method to create a new tuple with the work_year, job_title, and the rounded salaryinusd. Finally, we collect the filtered and transformed records into a Vec.

You can call the filter_and_convert function with the loaded dataset as follows:

```
fn main() {

  let url = "https://raw.githubusercontent.com/kittenpub/database-repository/main/ds_salaries.csv";

  match fetch_dataset(url) {

    Ok(csv_data) => {

      match load_dataset(&csv_data) {

        Ok(dataset) => {

          println!("Loaded {} records", dataset.len());

          let parsed_data = filter_and_convert(&dataset);

          println!("Filtered and converted data: {:?}", parsed_data);

        }

        Err(error) => {

          eprintln!("Error loading dataset: {}", error);
```

```
            }

          }

        }

        Err(error) => {

          eprintln!("Error fetching dataset: {}", error);

        }

    }

}
```

This example demonstrates a simple case of data parsing in Rust. Depending on your needs, you may need to apply more advanced parsing techniques or use additional libraries to handle specific data formats.

Data Structures in Rust

In Rust, various built-in data structures facilitate statistical analysis on datasets. Some common data structures include Vec, HashMap, HashSet, and BTreeMap. These structures enable storage, organization, and manipulation of data, allowing efficient computation of descriptive statistics, data transformations, and application of machine learning algorithms.

The primary data structures you'll be working with are:

Arrays

Arrays are an essential data structure in programming, representing fixed-size, contiguous sequences of elements with the same type. They offer several benefits, including efficient memory usage and fast access times due to their known size and regular structure. Arrays are particularly useful when you need to store a small, predetermined number of elements, and their size remains constant throughout the program's execution.

In Rust, you can define an array using the following syntax:

```
let my_array: [i32; 5] = [1, 2, 3, 4, 5];
```

In the above said example, my_array is an array of five elements, each of type i32 (32-bit signed integer). The elements are enclosed in square brackets, separated by commas. The type and size of the array are specified using a semicolon, as shown in [i32; 5].

Once an array is defined, you can access its elements using their index, which starts at zero:

```
let first_element = my_array[0]; // Access the first element

let third_element = my_array[2]; // Access the third element
```

Keep in mind that attempting to access an index outside the array's bounds will result in a panic at runtime. To avoid this, you can use methods like .get() that return an Option:

```
if let Some(element) = my_array.get(7) {

    println!("Element at index 7 is: {}", element);

} else {

    println!("Index 7 is out of bounds");

}
```

In addition to element access, arrays provide basic operations like .len() for obtaining their size or .iter() for iterating over their elements. However, for more complex operations, consider using other data structures, such as vectors (Vec), which offer greater flexibility and dynamic resizing capabilities.

Vectors

Vectors in Rust are dynamic arrays that can expand or contract during runtime, providing a versatile and efficient way to store elements of the same type. Their ability to accommodate a varying number of elements makes them ideal for numerous programming scenarios.

The following example demonstrates how to create and manipulate vectors in Rust:

```
// Creating an empty vector of i32 elements

let mut my_vector: Vec = Vec::new();

// Adding elements to the vector using the `push` method

my_vector.push(1);

my_vector.push(2);

my_vector.push(3);

// Accessing elements in the vector using indexing or the `get` method

let second_element = my_vector[1];
let third_element = my_vector.get(2).unwrap();


// Iterating over the elements of the vector

for element in &my_vector {

  println!("Element: {}", element);

}

// Removing the last element from the vector using the `pop` method

my_vector.pop();

// Modifying the elements of the vector

my_vector[0] = 42;

// Obtaining the length of the vector using the `len` method

let vector_length = my_vector.len();
```

In the above said example, we create a mutable vector of i32 elements and use various methods to add, access, modify, and remove elements. We also demonstrate how to iterate over the elements in the vector and retrieve its length. Vectors are an essential data structure in Rust that offers both convenience and flexibility when working with collections of elements.

Tuples

Tuples are a versatile data structure in programming languages like Rust, providing a convenient way to store ordered collections of elements with varying types. They are particularly useful when you need to group related values together without defining a custom struct. Tuples have a fixed size, meaning that the number of elements they contain is predetermined and cannot be changed after creation.

The following is an example of how to define a tuple in Rust:

```
let my_tuple: (i32, f64, &str) = (42, 3.14, "hello");
```

In the above said example, my_tuple is a tuple containing three elements: an i32 integer, an f64 floating-point number, and a string slice &str. The tuple is defined using parentheses, with the elements separated by commas.

Tuples provide a simple way to access their elements using dot notation followed by the index of the element:

```
let first_element = my_tuple.0; // 42
```

```
let second_element = my_tuple.1; // 3.14
let third_element = my_tuple.2; // "hello"
```

It is also possible to destructure tuples, assigning their elements to individual variables:

```
let (x, y, z) = my_tuple;

// x = 42, y = 3.14, z = "hello"
```

Tuples are an efficient way to group related values of different types together in a fixed-size, ordered collection. They are particularly useful for situations where creating a custom struct might be excessive or when you need to return multiple values from a function.

Structs

Structs, short for structures, are custom data types in Rust that allow you to group related values together into a single logical unit. They consist of fields, each having a specific data type, and can be used to represent various real-world entities or concepts in your code.

In the provided example, a SalaryRecord struct is defined, which contains information about an individual's work experience and salary details:

```
#[derive(Debug, Deserialize)]

struct SalaryRecord {

  work_year: i32,
  experience_level: String,

  // ...

}
```

The SalaryRecord struct has two fields: work_year of type i32 (integer) and experience_level of type String. These fields store an individual's work experience in years and their experience level, respectively. You can add more fields to the struct to include additional information as needed.

The #[derive(Debug, Deserialize)] attribute above the struct definition indicates that Rust should automatically derive the Debug and Deserialize traits for the SalaryRecord struct. The Debug trait enables pretty-printing of the struct's contents for debugging purposes, while the Deserialize trait, provided by the serde crate, allows for easy deserialization of CSV records or other serialized data directly into a SalaryRecord instance.

Structs play a crucial role in organizing and managing data in Rust, providing a convenient and efficient way to group and manipulate related values. They promote code readability and maintainability, as they encapsulate data into meaningful and reusable components that can be easily understood and manipulated.

HashMaps

HashMaps, as key-value data structures, enable efficient storage and rapid access of values using unique keys. They are particularly advantageous when dealing with unordered data. The following example demonstrates how to use a HashMap in Rust:

```
use std::collections::HashMap;

fn main() {

    // Create a new HashMap with String keys and i32 values



    let mut my_hashmap: HashMapi32> = HashMap::new();

    // Insert key-value pairs into the HashMap

   my_hashmap.insert("one".to_string(), 1);

   my_hashmap.insert("two".to_string(), 2);

   my_hashmap.insert("three".to_string(), 3);

    // Access a value in the HashMap using the key

   let two_value = my_hashmap.get("two");

    println!("The value for the key 'two' is: {:?}", two_value);
    // Iterate through the HashMap to display its contents

    for (key, value) in &my_hashmap {

        println!("Key: {}, Value: {}", key, value);

    }

    // Remove a key-value pair from the HashMap

   my_hashmap.remove("one");

    println!("HashMap after removing 'one': {:?}", my_hashmap);

}
```

This example covers creating a HashMap with String keys and i32 values, inserting key-value pairs, accessing a value using the key, iterating through the HashMap to display its contents, and removing a key-value pair. These operations demonstrate the versatility and efficiency of HashMaps when working with unordered key-value data in Rust.

To perform statistical analysis in Rust, you'll likely be using combinations of these data structures. For example, you might use a Vec to store a list of numerical values, and then use a combination of iterators and functional programming constructs to calculate statistical properties like the mean, median, and standard deviation.

Calculating Mean

To calculate the mean of a dataset using a Vec, you will first need to fetch the dataset from the provided link, parse the CSV data, and store the values in a Vec. Then, compute the mean of the values. In the below example, we will assume that the dataset has a single column of numeric values representing salaries.

Below is how to achieve this in Rust:

Add the required dependencies to your Cargo.toml:

```
[dependencies]

csv = "1.1"

reqwest = { version = "0.11", features = ["json"] }

tokio = { version = "1", features = ["full"] }
```

Write the code to fetch the CSV data, parse it, and calculate the mean:

```
use csv::ReaderBuilder;

use std::error::Error;
```

```
}

use reqwest::Error as ReqwestError;

async fn fetch_csv_data() -> Result, ReqwestError> {

  let url = "https://raw.githubusercontent.com/kittenpub/database-
repository/main/ds_salaries.csv";

  let response = reqwest::get(url).await?.text().await?;

  let mut reader =
ReaderBuilder::new().from_reader(response.as_bytes());

  let mut salaries = Vec::new();

  for result in reader.records() {

      let record = result?;

      let salary: f64 = record.get(0).unwrap().parse().unwrap();



      salaries.push(salary);

  }

  Ok(salaries)
```

```
}

fn calculate_mean(data: &[f64]) -> f64 {

   let sum: f64 = data.iter().sum();

   sum / data.len() as f64

}

#[tokio::main]

async fn main() -> Result<(), BoxError>> {

   let salaries = fetch_csv_data().await?;

   let mean_salary = calculate_mean(&salaries);

   println!("Mean salary: {:.2}", mean_salary);

   Ok(())

}
```

This code uses the csv crate to parse the CSV data, the reqwest crate to fetch the data, and the tokio crate for asynchronous processing. It fetches the dataset, stores the salaries in a Vec, and then calculates the mean salary using the calculate_mean function.

Calculating Median

To calculate the median of the dataset, you need to sort the values and then find the middle value (or the average of the two middle values for an even number of items).

Below is how to modify the previous example to compute the median:

Add a function to calculate the median:

```
fn calculate_median(data: &mut Vec) -> f64 {

  data.sort_unstable_by(|a, b| a.partial_cmp(b).unwrap());

  let len = data.len();

   if len % 2 == 0 {

      let mid1 = data[(len / 2) - 1];

      let mid2 = data[len / 2];

      (mid1 + mid2) / 2.0

  } else {
      data[len / 2]
  }
}
```

Update the main function to compute and display the median salary:

```
#[tokio::main]
async fn main() -> Result<(), BoxError>> {
    let mut salaries = fetch_csv_data().await?;
    let mean_salary = calculate_mean(&salaries);
    let median_salary = calculate_median(&mut salaries);
    println!("Mean salary: {:.2}", mean_salary);
    println!("Median salary: {:.2}", median_salary);
    Ok(())
}
```

The calculate_median function sorts the data using sort_unstable_by and computes the median based on the length of the dataset. If the dataset has an even number of items, it calculates the average of the two middle values. If it has an odd number of items, it selects the middle value directly. The main function now computes and displays both the mean and median salaries.

Common Data Cleaning and Preprocessing Techniques

Data cleaning and preprocessing are essential steps in preparing your dataset for analysis or machine learning models. They typically involve handling missing values, converting data types, scaling/normalizing data, and transforming categorical variables.

Below is an overview of common data cleaning and preprocessing techniques:

Handling Missing Values

Missing values in a dataset can lead to biased or incorrect analyses and predictions. There are several techniques to handle missing values:

Imputation
Filling in missing values with default values based on the rest of the dataset. Common imputation methods include:
Mean: Replace missing values with the mean of the available data for the feature.
Median: Replace missing values with the median of the available data for the feature.
Mode: Replace missing values with the mode (most frequent value) of the available data for the feature.
k-Nearest Neighbors: Fill in missing values based on the values of their k-nearest neighbors in the feature space.

● Regression: Predict missing values using a regression model trained on the available data.

Deletion
Removing instances with missing values from the dataset. This approach can lead to a loss of information, especially if a significant number of instances have missing values.

Data Type Conversion

Data type conversion is essential for ensuring compatibility between features and the algorithms used for analysis or modeling. Conversions may include:
● Numeric conversions: For example, converting integer values to floating-point values or vice versa.
Date and time conversions: Parsing date and time strings into appropriate date/time objects, such as chrono::NaiveDate in Rust.
String conversions: Parsing or formatting strings to match the required structure, such as extracting numbers or removing special characters.

Scaling/Normalizing Data

To ensure that features with different magnitudes are treated equally by machine learning algorithms, it's essential to scale or normalize the data. Common techniques include:
Min-Max scaling: Rescales the data to fit within a specified range, typically [0, 1]. It is calculated using the formula: (x - min) / (max - min).
Standard (Z-score) scaling: Centers the data around the mean with a standard deviation of 1. It is calculated using the formula: (x - mean) /

standard_deviation.

Encoding Categorical Variables

Many machine learning algorithms require numerical input data, necessitating the transformation of categorical variables into numerical values. Common techniques include:
One-hot encoding: Creates a binary variable for each category, with a value of 1 indicating the presence of the category and 0 otherwise.

Ordinal encoding: Assigns an integer value to each category based on its rank in the categorical variable.
Target encoding: Replaces each category with the mean of the target variable for instances within that category. This technique is particularly useful when dealing with high cardinality categorical variables.

Feature Engineering

Feature engineering aims to create new, meaningful features from the existing data to improve the performance of machine learning models or provide additional insights. Some techniques include:
Interaction terms: Multiplying or dividing two or more features to capture their combined effect on the target variable.
Polynomial features: Generating higher-order terms of the original features to capture non-linear relationships between variables.
Domain-specific transformations: Applying expert knowledge from the problem domain to derive new, meaningful features (e.g., calculating the age of a customer from their birth date or extracting keywords from text data).

Performing Data Cleaning and Preprocessing

Let's perform some of these preprocessing techniques on the given dataset:

Replace missing values: In our dataset, there are no explicitly missing values (e.g., "NaN"). However, if you encounter missing values, you can use the Option type in Rust to handle them.

Data type conversion: We've already performed data type conversions when deserializing the CSV into the SalaryRecord struct.

Scaling/normalizing data: Let's standardize the salaryinusd field. We'll create a function that takes a slice of SalaryRecord and returns a new Vec with the standardized salaryinusd field.

```
fn standardize_salaries(dataset: &[SalaryRecord]) -> Vec {

  let mean_salary = mean(&dataset.iter().map(|record|
record.salaryinusd).collect::>());

  let std_dev_salary = statis::stddev(&dataset.iter().map(|record|
record.salaryinusd).collect::>());

  dataset

    .iter()
```

```
        .map(|record| (record.salaryinusd - mean_salary) / std_dev_salary)

        .collect()

}
```

Encoding categorical variables: Let's encode the job_title field using one-hot encoding. We'll first create a function to generate a mapping of unique job titles to indices.

```
use std::collections::HashMap;

fn create_job_title_mapping(dataset: &[SalaryRecord]) ->
HashMapusize> {

    let mut job_title_set: HashSet = dataset.iter().map(|record|
record.job_title.clone()).collect();

    let mut job_title_mapping: HashMapusize> = HashMap::new();

    for (index, job_title) in job_title_set.drain().enumerate() {

        job_title_mapping.insert(job_title, index);



    }

   job_title_mapping
}
```

Next, create a function to one-hot encode the job_title field based on the mapping:

```
fn one_hot_encode_job_titles(dataset: &[SalaryRecord], mapping:
&HashMapusize>) -> Vec> {

  dataset

     .iter()

     .map(|record| {

        let mut encoding = vec![0u8; mapping.len()];

        let index = mapping[&record.job_title];

        encoding[index] = 1;

        encoding

})

.collect()

}
```

Feature engineering: For this dataset, we could create a new feature, `us_based`, indicating whether the company is located in the United States. Below is a function to generate this new feature:

```
fn create_us_based_feature(dataset: &[SalaryRecord]) -> Vec {

  dataset

      .iter()

      .map(|record| if record.company_location == "US" { 1u8 } else { 0u8
})

      .collect()

}
```

Now, in the main function, you can apply these preprocessing techniques on the dataset:

```
fn main() {

  let url = "https://raw.githubusercontent.com/kittenpub/database-
repository/main/ds_salaries.csv";

  match fetch_dataset(url) {
    Ok(csv_data) => {
        match load_dataset(&csv_data) {
            Ok(dataset) => {
                println!("Loaded {} records", dataset.len());
                let standardized_salaries = standardize_salaries(&dataset);
                println!("Standardized salaries: {:?}", standardized_salaries);
                let job_title_mapping = create_job_title_mapping(&dataset);
                println!("Job title mapping: {:?}", job_title_mapping);
                let one_hot_encoded_job_titles =
one_hot_encode_job_titles(&dataset, &job_title_mapping);
                println!("One-hot encoded job titles: {:?}",
one_hot_encoded_job_titles);
                let us_based_feature = create_us_based_feature(&dataset);
                println!("US-based feature: {:?}", us_based_feature);
            }
```

```
                Err(error) => {
                    eprintln!("Error loading dataset: {}", error);

                }
            }
        }
        Err(error) => {
            eprintln!("Error fetching dataset: {}", error);
        }
    }
}
```

Summary

In this chapter, we focused on data handling and preprocessing using Rust. We started by loading a dataset containing salary information for different data science fields. The dataset contained 11 columns, including work_year, experience_level, employment_type, job_title, salary, salary_currency, salaryinusd, employee_residence, remote_ratio, company_location, and company_size. We used the reqwest crate to fetch the dataset and the csv and serde crates to deserialize the CSV data into a custom SalaryRecord struct. We then explored various data structures in Rust for performing statistical analysis, such as arrays, vectors, tuples, structs, and HashMaps. These data structures, combined with Rust's iterator and pattern matching capabilities, enable powerful data manipulation and analysis.

Next, we delved into data parsing, a crucial step in preparing raw data for further processing. We demonstrated how to use Rust's iterator and pattern matching features to filter and convert the dataset based on specific conditions. We also introduced more advanced parsing techniques and libraries for handling specific data formats. We covered common data cleaning and preprocessing techniques, including handling missing values, converting data types, scaling/normalizing data, encoding categorical variables, and feature engineering. We applied these techniques to the given dataset, standardizing the salaryinusd field, one-hot encoding the job_title field, and creating a new feature indicating whether the company is based in the United States. Throughout the chapter, we highlighted the flexibility and power of Rust for data handling and preprocessing tasks.

We showed how to leverage Rust's built-in data structures, functional programming constructs, and third-party libraries to efficiently manipulate and transform datasets in preparation for analysis or machine learning models.

To sum it up, Chapter 2 provided a comprehensive introduction to data handling and preprocessing using Rust. By working with a real dataset, we demonstrated how to load, parse, clean, and preprocess data with Rust's core features and libraries, setting the stage for more advanced statistical operations and visualizations in later chapters.

Chapter 3: Descriptive Statistics in Rust

Introduction to Descriptive Statistics

In this Chapter 3, we'll put strong attention on exploring descriptive statistics and that too in Rust. Descriptive statistics are used to summarize and describe the main features of a dataset. They help data science professionals understand the data's distribution, central tendency, and variability. Rust can facilitate these tasks using its powerful built-in data structures, functional programming constructs, and third-party libraries.

The following are some of the key concepts and techniques in descriptive statistics:

Measures of Central Tendency: These measures describe the center of the dataset. The most common measures are the mean (average), median (middle value), and mode (most frequent value). Rust's iterators and pattern matching capabilities can be leveraged to calculate these measures efficiently.

Measures of Dispersion: These measures describe the spread of the dataset. Common measures include range (difference between the maximum and minimum values), variance, and standard deviation. Rust's iterator and mathematical functions can be used to calculate these measures.

Quantiles: Quantiles, including quartiles and percentiles, divide the dataset into equal intervals. They help understand the data's distribution and identify potential outliers. Rust's sorting and indexing capabilities can be used to find quantiles.

Frequency Distributions: A frequency distribution is a summary of the dataset's values and their frequencies. Histograms and bar charts are common ways to visualize frequency distributions. Rust's HashMap and Vec data structures can be employed to create frequency distributions.

Correlation and Covariance: These measures describe the relationship between two or more variables in the dataset. Correlation coefficients (e.g., Pearson, Spearman) and covariance can help determine if variables are related and the strength of their relationship. Rust's mathematical functions and iterators can be used to calculate these measures.

Summary Statistics: Summary statistics are concise descriptions of the dataset's main features, often represented in a table. They typically include measures of central tendency, dispersion, and distribution. Rust's powerful data manipulation capabilities can be used to generate summary statistics.

Measures of Central Tendency

Measures of central tendency are statistical indicators that help data science professionals understand the central point of a dataset. These measures provide a single value that represents the dataset and can be used to summarize the data or compare it with other datasets.

The three primary measures of central tendency are the mean, median, and mode.

Mean: The mean, or average, is calculated by summing all the values in the dataset and dividing by the number of values. The mean is sensitive to outliers, meaning that extreme values can greatly affect the result. It's widely used in many applications, but it may not always represent the dataset's true center when dealing with skewed data or outliers.
Median: The median is the middle value in a dataset when the data points are arranged in ascending or descending order. If there's an even number of values, the median is the average of the two middle values. The median is less sensitive to outliers and can better represent the dataset's center when dealing with skewed data.

Mode: The mode is the value that appears most frequently in the dataset. There can be multiple modes in a dataset or no mode at all. The mode can be useful when dealing with categorical data or when the most common value is of particular interest.

Calculate Measures of Central Tendency

Now, let's see how to calculate these measures of central tendency on the salaryinusd field of the given dataset:

```
fn mean(data: &[f64]) -> f64 {

  let sum: f64 = data.iter().sum();

  sum / (data.len() as f64)

}

fn median(data: &mut [f64]) -> f64 {

  data.sort_unstable();

   let mid = data.len() / 2;

   if data.len() % 2 == 0 {

     (data[mid - 1] + data[mid]) / 2.0

  } else {

     data[mid]

  }
```

```
}

fn mode(data: &[f64]) -> Option {
  use std::collections::HashMap;
  let mut frequency_map = HashMap::new();
  for &value in data {
     let count = frequency_map.entry(value).or_insert(0);
     *count += 1;
  }
  frequency_map
     .into_iter()
     .max_by_key(|&(_, count)| count)
     .map(|(value, _)| value)
}

fn main() {
   // ... Load the dataset as shown in previous examples ...

   let salary_data: Vec = dataset.iter().map(|record|
record.salaryinusd).collect();

  let mut salary_data_sorted = salary_data.clone();

  let mean_salary = mean(&salary_data);

  let median_salary = median(&mut salary_data_sorted);

  let mode_salary = mode(&salary_data);

  println!("Mean salary: {}", mean_salary);

  println!("Median salary: {}", median_salary);

  println!("Mode salary: {:?}", mode_salary);

}
```

In the above sample program, we calculate the mean, median, and mode of the salaryinusd field for the dataset. These measures of central tendency provide insights into the overall salary distribution for data science professionals, helping identify trends, compare different groups, or inform decisions related to compensation and workforce planning.

Measures of Dispersion

Measures of dispersion describe the spread or variability of a dataset. They help data science professionals understand how diverse the data is and provide insights into its underlying structure.

The most common measures of dispersion are range, variance, and standard deviation.
Range: The range is the difference between the maximum and minimum values in a dataset. It provides a basic measure of the dataset's spread but can be sensitive to outliers and may not accurately represent the overall variability when dealing with skewed data.
Variance: The variance is the average of the squared differences from the mean. It measures how far each data point is from the mean and gives a more accurate representation of the dataset's variability compared to the range. However, the unit of variance is the square of the data's unit, which can make it difficult to interpret.
Standard Deviation: The standard deviation is the square root of the variance. It measures the dataset's dispersion in the same unit as the data, making it easier to interpret and compare with the mean. Standard deviation is widely used in many applications to understand the variability and identify potential outliers.

Calculate Measures of Dispersion

Let's demonstrate how to calculate these measures of dispersion on the salaryinusd field of the given dataset:

```
fn range(data: &[f64]) -> f64 {

    let max_value = data.iter().cloned().max_by(|a, b|
a.partial_cmp(b).unwrap()).unwrap();

    let min_value = data.iter().cloned().min_by(|a, b|
a.partial_cmp(b).unwrap()).unwrap();

   max_value - min_value

}

fn variance(data: &[f64]) -> f64 {

   let mean_value = mean(data);

    let squared_diffs: Vec = data.iter().map(|&value| (value -
mean_value).powi(2)).collect();

   mean(&squared_diffs)

}

fn standard_deviation(data: &[f64]) -> f64 {

   variance(data).sqrt()
}

fn main() {

    // ... Load the dataset as shown in previous examples ...

    let salary_data: Vec = dataset.iter().map(|record|
record.salaryinusd).collect();

   let salary_range = range(&salary_data);

   let salary_variance = variance(&salary_data);

   let salary_standard_deviation = standard_deviation(&salary_data);

   println!("Salary range: {}", salary_range);

   println!("Salary variance: {}", salary_variance);

   println!("Salary standard deviation: {}", salary_standard_deviation);

}
```

In the above sample program, we calculate the range, variance, and standard deviation of the salaryinusd field for the dataset. These measures of dispersion provide insights into the salary variability for data science professionals, helping identify potential outliers, understand wage inequality, and inform decisions related to compensation structure and workforce planning.

Exploratory Data Analysis (EDA)

Exploratory Data Analysis (EDA) is an essential step in the data analysis process that involves visualizing and summarizing datasets to uncover patterns, trends, relationships, and anomalies. EDA helps statisticians and data scientists gain insights, generate hypotheses, and make informed decisions before moving to more advanced statistical modeling or machine learning techniques.

EDA is crucial for statisticians as it helps them:

- Understand the data's structure and distribution.
- Identify potential data quality issues, such as missing values, outliers, or inconsistencies.
- Discover relationships and correlations between variables.
- Generate hypotheses and identify areas for further analysis or modeling.

In Rust, EDA can be performed using a combination of summary statistics, data visualizations, and other data exploration techniques. Rust's powerful data manipulation capabilities, along with third-party libraries like ndarray, statrs, statis, and plotly, facilitate EDA tasks.

Implementing EDA

The given below is a sample implementation of EDA on the given dataset:

Calculate summary statistics (e.g., mean, median, mode, variance, standard deviation) for the salaryinusd field, as shown in previous examples.

Visualize the salary distribution using a histogram. The plotly crate can be used to create the histogram:

```
use plotly::{Plot, Histogram, Layout, CommonMarker, common::Color};

fn plot_histogram(data: &[f64], bins: usize, title: &str) {

  let trace = Histogram::new(data)

      .nbinsx(bins)

      .name("Salary Distribution")


 .marker(CommonMarker::new().color(Color::String("#1f77b4".to_owned
())));

  let layout = Layout::new().title(title);

  let mut plot = Plot::new();

  plot.add_trace(trace);
```

```
    plot.set_layout(layout);

    plot.show();

}

fn main() {

    // ... Load the dataset as shown in previous examples ...

    let salary_data: Vec = dataset.iter().map(|record|
record.salaryinusd).collect();

    plot_histogram(&salary_data, 50, "Salary Distribution in USD");

}
```

Visualize relationships between variables using scatter plots. For example, plot salaryinusd against experience_level:

```
use plotly::{Plot, Scatter, Layout, Axis, Mode, ScatterMode};

fn plot_scatter(x_data: &[f64], y_data: &[f64], x_title: &str, y_title: &str,
title: &str) {

    let trace = Scatter::new(x_data, y_data)

        .mode(ScatterMode::Markers(Mode::new()))
        .name("Data Points")

 .marker(CommonMarker::new().color(Color::String("#1f77b4".to_owned
()));

    let layout = Layout::new()

        .title(title)

        .xaxis(Axis::new().title(x_title))

        .yaxis(Axis::new().title(y_title));

    let mut plot = Plot::new();

    plot.add_trace(trace);

    plot.set_layout(layout);

    plot.show();

}

fn main() {
    // ... Load the dataset as shown in previous examples ...

    let experience_data: Vec = dataset.iter().map(|record|
record.experience_level as f64).collect();

    let salary_data: Vec = dataset.iter().map(|record|
record.salaryinusd).collect();

    plot_scatter(&experience_data, &salary_data, "Experience Level",
"Salary in USD", "Experience Level vs Salary");

}
```

These EDA techniques help statisticians and data scientists gain insights into the given dataset, identify relationships, and discover potential data quality issues. By combining summary statistics, data visualizations, and other data exploration techniques, statisticians can make informed decisions about further analysis, modeling, or data preprocessing steps.

Analyze categorical variables using contingency tables and bar charts. For example, examine the distribution of job_title:

```
use std::collections::HashMap;



use plotly::{Plot, Bar, Layout};

fn plot_bar_chart(data: &HashMapusize>, x_title: &str, y_title: &str, title:
&str) {
   let x: Vec = data.keys().cloned().collect();

   let y: Vec = data.values().cloned().collect();

   let trace = Bar::new(x, y)

       .name("Job Title Distribution")


 .marker(CommonMarker::new().color(Color::String("#1f77b4".to_owned
()));

   let layout = Layout::new()

       .title(title)

       .xaxis(Axis::new().title(x_title))

       .yaxis(Axis::new().title(y_title));

   let mut plot = Plot::new();

   plot.add_trace(trace);

   plot.set_layout(layout);

   plot.show();
}

fn main() {

    // ... Load the dataset as shown in previous examples ...

   let mut job_title_counts = HashMap::new();

   for record in dataset.iter() {

       let count =
job_title_counts.entry(record.job_title.clone()).or_insert(0);

       *count += 1;

   }

    plot_bar_chart(&job_title_counts, "Job Title", "Frequency", "Job Title
Distribution");

}
```

In the above sample program, we create a bar chart to visualize the distribution of job_title. This analysis helps to understand the composition of job titles in the dataset, which can inform decisions about workforce planning, hiring strategies, or further analysis focusing on specific roles.

By combining these EDA techniques, statisticians and data scientists can explore various aspects of the given dataset, identify relationships, and discover potential data quality issues. This process helps them make informed decisions about further analysis, modeling, or data preprocessing steps.

Summary

In this Chapter 3, we focused on Descriptive Statistics in Rust, emphasizing various techniques that data science professionals perform and how Rust facilitates these tasks. We discussed the importance of Exploratory Data Analysis (EDA) and its benefits for statisticians in understanding the data's structure, identifying potential data quality issues, discovering relationships and correlations between variables, and generating hypotheses for further analysis or modeling.

We covered two essential aspects of descriptive statistics: measures of central tendency and measures of dispersion. Measures of central tendency, including mean, median, and mode, help data professionals understand the central point of a dataset. These measures provide a single value representing the dataset and can be used to summarize the data or compare it with other datasets. Measures of dispersion, such as range, variance, and standard deviation, describe the spread or variability of a dataset. They help data professionals understand how diverse the data is and provide insights into its underlying structure. We demonstrated how to calculate measures of central tendency and dispersion on the given dataset using Rust. We also provided sample implementations for visualizing the salary distribution using histograms, plotting relationships between variables using scatter plots, and analyzing categorical variables using contingency tables and bar charts.

Throughout this chapter, we utilized Rust's powerful data manipulation capabilities along with third-party libraries like ndarray, statrs, statis, and plotly to facilitate EDA tasks. These libraries enabled us to perform data preprocessing, calculate summary statistics, and create visualizations for data exploration. These techniques form the foundation for further data analysis, modeling, and decision-making in the data science domain.

Chapter 4: Probability Distributions and Random Variables

Discrete Probability Distribution

Discrete probability distributions are pivotal in statistics as they describe the probabilities of outcomes for discrete random variables. A random variable is deemed discrete if it assumes a finite or countable number of values. These distributions aid in modeling uncertain events and making predictions based on data.

Some prevalent discrete probability distributions include:

Uniform Distribution: This distribution assigns equal probability to all possible outcomes. It is often used when there is no reason to favor one outcome over another.
Bernoulli Distribution: This distribution models the probability of success or failure (binary outcomes) in a single experiment. It is typically employed in scenarios where there are only two possible outcomes, such as coin tosses or true/false questions.
Binomial Distribution: This distribution describes the number of successes in a fixed number of independent Bernoulli trials, where each trial has the same probability of success. Applications include modeling the number of heads in multiple coin tosses or the number of defective items in a sample.

Poisson Distribution: This distribution represents the number of events occurring within a fixed interval of time or space, given a constant average rate of occurrence. It is widely used in fields such as queuing theory, telecommunications, and biology to model rare events or arrivals.
Geometric Distribution: This distribution models the number of trials required to achieve the first success in a sequence of independent

Bernoulli trials. It is applicable in situations where the interest lies in determining the waiting time until a specific event occurs.

Although applying discrete probability distributions to a given database might not be directly pertinent if the dataset doesn't involve random variables and their probabilities, understanding these distributions remains valuable. In Rust, you can use the 'statrs' crate to implement these distributions in various scenarios where discrete probability distributions are applicable. By leveraging these distributions conceptually, you can model uncertain events, analyze data, and make informed predictions.

Uniform Distribution

A uniform distribution is a probability distribution that assigns equal probabilities to all possible outcomes of a discrete random variable. It represents a scenario in which each outcome is equally likely to occur. For instance, when rolling a fair six-sided die, all six outcomes (1, 2, 3, 4, 5, and 6) have an equal probability of 1/6, indicating a uniform distribution across the possible outcomes.

```
use statrs::distribution::{Uniform, Discrete};

fn main() {



    let uniform = Uniform::new(1, 6); // Represents a fair six-sided die

    let probability = uniform.pdf(3); // Probability of rolling a 3
    println!("Uniform Distribution: P(X = 3) = {}", probability);

}
```

Bernoulli Distribution

The Bernoulli distribution is a discrete probability distribution representing a single binary experiment with two possible outcomes: success (1) and failure (0). It is characterized by a single parameter, p, which denotes the probability of success. The probability of failure is given by 1-p. This simple yet essential distribution forms the basis for various statistical and machine learning models that involve binary events or decision-making processes.

```
use statrs::distribution::{Bernoulli, Discrete};

fn main() {

    let bernoulli = Bernoulli::new(0.6).unwrap(); // Probability of success
(p) = 0.6

    let probability = bernoulli.pdf(1); // Probability of success (X = 1)

    println!("Bernoulli Distribution: P(X = 1) = {}", probability);

}
```

Binomial Distribution

The binomial distribution is a discrete probability distribution that models the number of successes in a fixed number of independent Bernoulli trials, each with the same probability of success, p. It captures the likelihood of obtaining a specific number of successes, given the total number of trials (n) and the probability of success in each trial. This distribution is widely used in various applications, such as analyzing proportions, survey responses, and quality control.

```
use statrs::distribution::{Binomial, Discrete};

fn main() {

    let binomial = Binomial::new(0.6, 10).unwrap(); // 10 trials, each with probability of success (p) = 0.6

    let probability = binomial.pdf(4); // Probability of 4 successes (X = 4)

    println!("Binomial Distribution: P(X = 4) = {}", probability);

}
```

Poisson Distribution

The Poisson distribution is a probability distribution that describes the number of events occurring within a fixed time or space interval, assuming a constant average rate of occurrence (denoted by lambda). It is particularly useful for modeling rare events, such as phone call arrivals at a call center or the number of accidents at an intersection. The distribution captures the probability of observing a specific count of events while accounting for the underlying rate (lambda).

```
use statrs::distribution::{Poisson, Discrete};

fn main() {

    let poisson = Poisson::new(5.0).unwrap(); // Average rate of occurrence
(lambda) = 5

    let probability = poisson.pdf(3); // Probability of 3 events (X = 3)

    println!("Poisson Distribution: P(X = 3) = {}", probability);

}
```

Geometric Distribution

A geometric distribution is a discrete probability distribution that characterizes the number of trials needed to achieve the first success in a series of independent Bernoulli trials. Each trial has a fixed probability of success, denoted as p. The distribution is useful for modeling scenarios where events occur independently, and the probability of success remains constant across trials. The geometric distribution helps in understanding the behavior and likelihood of occurrences until the first successful event.

```
use statrs::distribution::{Geometric, Discrete};
```

```
fn main() {

    let geometric = Geometric::new(0.6).unwrap(); // Probability of success
(p) = 0.6

   let probability = geometric.pdf(3);

// Probability of the first success occurring on the 3rd trial (X = 3)

println!("Geometric Distribution: P(X = 3) = {}", probability);

}
```

These examples demonstrate how to work with different discrete probability distributions using the `statrs` crate in Rust. As mentioned earlier, these distributions may not be directly applicable to the given dataset since it is not about random variables and their probabilities. However, these implementations can be adapted for other datasets or use cases where discrete probability distributions are relevant.

Overall, discrete probability distributions play a vital role in modeling uncertain events and making predictions based on data. We discussed various discrete probability distributions and provided sample implementations in Rust using the `statrs` crate. Although these distributions may not be directly applicable to the given dataset, they can be adapted for other datasets or use cases, forming a foundation for statistical modeling and decision-making in data science.

Continuous Probability Distribution

Continuous probability distributions play a significant role in modeling and predicting outcomes for continuous random variables, which can assume an infinite number of values within a particular range. These distributions are essential when dealing with uncertain events and continuous variables in various fields, such as finance, engineering, and natural sciences.

The following are some of the widely-used continuous probability distributions, along with brief explanations:

Uniform Distribution: This distribution assigns equal probability to all values within a specified range. It is often used when there is no prior information about the distribution of the variable.
Normal (Gaussian) Distribution: Characterized by its bell-shaped curve, the normal distribution is a fundamental distribution in statistics. Many real-world phenomena follow this distribution, making it essential for various applications.
Exponential Distribution: This distribution models the time between events in a Poisson process, where events occur continuously and independently at a constant average rate. It is frequently used in reliability and survival analysis.

Beta Distribution: A versatile distribution, the Beta distribution is defined on the interval (0, 1) and is particularly useful in modeling probabilities or proportions. It is widely employed in Bayesian statistics as a conjugate prior for the binomial and Bernoulli distributions.

Gamma Distribution: The Gamma distribution is a flexible two-parameter distribution that models continuous variables with positive values, such as waiting times or lifetimes. It is a generalization of the exponential distribution and is a conjugate prior for the Poisson distribution in Bayesian statistics.

Uniform Distribution

A uniform distribution assigns equal probabilities to all possible outcomes of a continuous random variable within a specified range. For example, generating a random number between 0 and 1.

```
use statrs::distribution::{Uniform, Continuous};

fn main() {

    let uniform = Uniform::new(0.0, 1.0); // Represents a random number
between 0 and 1

    let probability = uniform.pdf(0.5); // Probability density function at 0.5

    println!("Uniform Distribution: f(0.5) = {}", probability);

}
```

Normal (Gaussian) Distribution

A normal distribution, also known as Gaussian distribution, is a bell-shaped curve defined by its mean (μ) and standard deviation (σ). It is widely used in natural and social sciences due to its desirable properties.

```
use statrs::distribution::{Normal, Continuous};

fn main() {

    let normal = Normal::new(0.0, 1.0).unwrap(); // Mean (μ) = 0, Standard
Deviation (σ) = 1

    let probability = normal.pdf(0.5); // Probability density function at 0.5

    println!("Normal Distribution: f(0.5) = {}", probability);

}
```

Exponential Distribution

An exponential distribution models the time between events in a Poisson process, where events occur independently at a constant average rate (λ).

```
use statrs::distribution::{Exponential, Continuous};

fn main() {

    let exponential = Exponential::new(1.0).unwrap(); // Rate (λ) = 1
    let probability = exponential.pdf(0.5); // Probability density function at 
0.5

    println!("Exponential Distribution: f(0.5) = {}", probability);

}
```

<u>Beta Distribution</u>

A beta distribution models the distribution of probabilities for a continuous random variable within a fixed range, typically [0, 1]. It is defined by two shape parameters, α and β.

```
use statrs::distribution::{Beta, Continuous};

fn main() {

    let beta = Beta::new(2.0, 5.0).unwrap(); // Shape parameters (α) = 2, (β) 
= 5



    let probability = beta.pdf(0.5); // Probability density function at 0.5

    println!("Beta Distribution: f(0.5) = {}", probability);

}
```

<u>Gamma Distribution</u>

A gamma distribution models the distribution of waiting times for a sequence of events in a Poisson process. It is defined by a shape parameter (k) and a scale parameter (θ).

```
use statrs::distribution::{Gamma, Continuous};

fn main() {

   let gamma = Gamma::new(2.0, 1.0).unwrap(); // Shape parameter (k) =
2,Scale parameter (θ) = 1

let probability = gamma.pdf(0.5); // Probability density function at 0.5

println!("Gamma Distribution: f(0.5) = {}", probability);

}
```

These examples demonstrate how to work with different continuous probability distributions using the `statrs` crate in Rust. As mentioned earlier, these distributions may not be directly applicable to the given dataset since it is not about random variables and their probabilities. However, these implementations can be adapted for other datasets or use cases where continuous probability distributions are relevant.

Generating Random Variables

To generate random variables in Rust, we can use the rand crate, which provides a suite of random number generators and distributions. In the below example, I'll demonstrate how to generate random variables using the rand crate and apply them to the given dataset.

First, add the rand crate to your Cargo.toml:

```
[dependencies]

rand = "0.8"
```

Now, let's generate random variables and apply them to a new column in the given dataset:

Import the required crates and modules:

```
use ndarray::prelude::*;

use ndarray_csv::{Array2Reader, Array2Writer};

use std::fs::File;

use std::io::{Read, Write};

use rand::Rng;

use rand::distributions::{Distribution, Uniform};
```

Load the dataset and add a new column for random variables:

```
fn main() {

    let mut file = File::open("ds_salaries.csv").expect("Unable to open the
file");

   let mut buffer = String::new();

    file.read_to_string(&mut buffer).expect("Unable to read the file");

    let mut dataset: Array2 = buffer.read_array2().expect("Unable to read
the CSV data");

   let nrows = dataset.nrows();

   let ncols = dataset.ncols();

    let mut new_dataset = Array2::from_elem((nrows, ncols + 1),
String::from(""));

   new_dataset.slice_mut(s![.., ..ncols]).assign(&dataset);

   new_dataset.slice_mut(s![0, ncols]).assign(&array!
[String::from("random_variable")]);

   // Generate random variables
```

```
  let mut rng = rand::thread_rng();

  let uniform = Uniform::new(0.0, 1.0);

  for row in 1..nrows {

      let random_value: f64 = rng.sample(uniform);

      new_dataset[[row, ncols]] = format!("{:.2}", random_value);

  }

   // Save the new dataset to a CSV file

   let mut file =
File::create("ds_salaries_with_random.csv").expect("Unable to create the
file");

   let mut writer = csv::Writer::from_writer(&mut file);

   new_dataset.write_csv(&mut writer).expect("Unable to write the CSV
data");

}
```

In the above sample program, we loaded the dataset using ndarray and ndarray_csv crates, created a new dataset with an additional column for random variables, and assigned the column name "random_variable".

Then, we generated random variables from a uniform distribution between 0 and 1 using the rand crate and populated the new column with these random values. Finally, we saved the new dataset to a CSV file named "ds_salaries_with_random.csv".

Sampling from Distributions

Sampling from distributions involves generating random data points that follow a specified probability distribution. This technique offers several benefits for data professionals, including:
Simulating data: Sampling can help create data for hypothesis testing, model validation, or performance evaluation. This process enables professionals to assess the effectiveness of their models or methods using various input data scenarios.
Generating synthetic data: In cases where sensitive information must be anonymized, sampling from distributions can help generate synthetic data that preserves the statistical properties of the original data while maintaining confidentiality.
Assessing robustness: By generating random inputs, data professionals can evaluate the robustness of their models or analyses under different scenarios. This assessment helps identify potential weaknesses and improve the overall quality of the models or analyses.

In Rust, the rand crate is an ideal choice for sampling from distributions. This crate offers a wide range of probability distributions, including Uniform, Normal, Bernoulli, Poisson, and many others. To utilize the rand crate for sampling, follow these steps:

Sample Program for Sampling from Distributions

In the following example, we will demonstrate how to sample from a normal distribution and a discrete distribution (Poisson).

Add the rand crate to your Cargo.toml:

```
[dependencies]


rand = "0.8"
```

Import the required modules:

```
use rand::Rng;

use rand::distributions::{Distribution, Normal, Poisson};
```

Sample from a normal distribution:

```
fn main() {

    let normal_distribution = Normal::new(0.0, 1.0).unwrap(); // Mean (μ)
= 0, Standard Deviation (σ) = 1

   let mut rng = rand::thread_rng();

   for _ in 0..10 {

       let sample: f64 = rng.sample(normal_distribution);

       println!("Sample from normal distribution: {:.2}", sample);
```

```
  }

}
```

Sample from a discrete distribution (Poisson):

```
fn main() {

   let poisson_distribution = Poisson::new(5.0).unwrap(); // Lambda (λ) =
5

  let mut rng = rand::thread_rng();

  for _ in 0..10 {

     let sample: u64 = rng.sample(poisson_distribution);

     println!("Sample from Poisson distribution: {}", sample);

  }

}
```

In both examples, we first define the distribution parameters (mean and standard deviation for normal distribution, and lambda for Poisson distribution). Then, we use the rand::Rng trait to generate random samples from the specified distribution. The loop iterates ten times, printing ten random samples from each distribution. Sampling from distributions is an essential tool for data professionals to generate synthetic data or perform simulations for testing and validating models.

Estimating Distribution Parameters

Estimating distribution parameters is the process of determining the values of parameters for a probability distribution that best describe the underlying data. This is an essential step in statistical modeling, as the estimated parameters can be used to make predictions or inferences about the population from which the data was sampled.

There are various techniques for estimating distribution parameters, including:

Method of Moments (MoM)
Maximum Likelihood Estimation (MLE)
Bayesian Estimation
Least Squares

We will look after each technique and provide you sample program in Rust scripts for the given dataset. Note that these examples may require some assumptions about the underlying distributions of the data, which might not always be valid.

Method of Moments (MoM)

The Method of Moments (MoM) is a straightforward technique for estimating distribution parameters by equating sample moments to theoretical moments of a distribution. Sample moments are calculated from the data, while theoretical moments are derived from the distribution's mathematical properties.

For example, let's assume the salary data in the dataset follows a normal distribution. We can estimate the mean (μ) and standard deviation (σ) of the normal distribution using the first and second moments.

```
use ndarray::prelude::*;



use ndarray_csv::{Array2Reader, Array2Writer};

use std::fs::File;

use std::io::{Read, Write};

use statrs::statistics::Statistics;

fn main() {

    let mut file = File::open("ds_salaries.csv").expect("Unable to open the
file");

   let mut buffer = String::new();

    file.read_to_string(&mut buffer).expect("Unable to read the file");
    let dataset: Array2 = buffer.read_array2().expect("Unable to read the
CSV data");

   let salaries: Array1 = dataset

       .column(4)

       .iter()

       .skip(1)

       .map(|s| s.parse::().unwrap())

       .collect();

   let mean = salaries.mean().unwrap();

   let variance = salaries.variance().unwrap();

   let std_dev = variance.sqrt();

   println!("Method of Moments (Normal Distribution):");

   println!("Estimated mean (μ): {:.2}", mean);

    println!("Estimated standard deviation (σ): {:.2}", std_dev);

}
```

Maximum Likelihood Estimation (MLE)

Maximum Likelihood Estimation (MLE) is a popular technique for estimating distribution parameters by maximizing the likelihood function. The likelihood function measures how likely it is to observe the given data given a set of parameters for a particular distribution. MLE finds the parameter values that maximize the likelihood function, making the observed data most probable.

Continuing with the salary data example and assuming it follows a normal distribution, we can use MLE to estimate the mean (μ) and standard deviation (σ). In this case, MLE estimates will be the same as the MoM estimates. For simplicity, we'll use the statrs crate to perform MLE for the normal distribution.

```
use statrs::fit::MLE;

use statrs::distribution::Normal;

fn main() {

    // Assume `salaries` array is already loaded as in the previous example.

   let mle_normal = Normal::fit_mle(&salaries).unwrap();

   println!("Maximum Likelihood Estimation (Normal Distribution):");
```

```
  println!("Estimated mean (μ): {:.2}", mle_normal.mean());

   println!("Estimated standard deviation (σ): {:.2}",
mle_normal.std_dev());

}
```

Bayesian Estimation

Bayesian estimation incorporates prior knowledge about the distribution parameters and updates the estimates using the observed data. The prior knowledge is represented by a prior distribution, and the updated estimates are represented by a posterior distribution. Bayesian estimation is based on Bayes' theorem, which combines the prior distribution, the likelihood of the observed data, and the evidence (marginal likelihood) to compute the posterior distribution.

For example, let's assume that the salary data follows a normal distribution with an unknown mean (μ) and known standard deviation (σ). We can use a normal distribution as a prior for the mean with a known prior mean (μ_0) and standard deviation (σ_0). In this case, the posterior distribution for the mean is also a normal distribution, and we can calculate its parameters analytically.

```
fn bayesian_normal_mean(

  prior_mean: f64,
  prior_std_dev: f64,

  data_mean: f64,

  data_std_dev: f64,

  n: usize,

) -> (f64, f64) {

   let weight = (n as f64) * (data_std_dev.powi(2)) / ((n as f64) *
data_std_dev.powi(2) + prior_std_dev.powi(2));

   let posterior_mean = prior_mean + weight * (data_mean - prior_mean);

   let posterior_std_dev = (prior_std_dev.powi(2) * data_std_dev.powi(2) /
((n as f64) * prior_std_dev.powi(2) + data_std_dev.powi(2))).sqrt();

  (posterior_mean, posterior_std_dev)

}

fn main() {

   // Assume `salaries` array is already loaded as in the previous examples.

   let prior_mean = 70000.0; // Prior mean (μ₀)
```

```
    let prior_std_dev = 10000.0; // Prior standard deviation (σ₀)

    let data_mean = salaries.mean().unwrap(); // Data mean

    let data_std_dev = salaries.variance().unwrap().sqrt(); // Data standard
deviation

    let n = salaries.len(); // Number of data points

   let (posterior_mean, posterior_std_dev) =

       bayesian_normal_mean(prior_mean, prior_std_dev, data_mean,
data_std_dev, n);

   println!("Bayesian Estimation (Normal Distribution):");

   println!("Posterior mean: {:.2}", posterior_mean);

   println!("Posterior standard deviation: {:.2}", posterior_std_dev);

}
```

Least Squares

Least Squares is a technique for estimating parameters in regression models by minimizing the sum of squared differences between the observed data and the predicted data based on the model. While Least Squares is not directly applicable to estimating parameters of univariate probability distributions, it can be used for fitting regression models with underlying assumptions about the error distributions.

For example, let's assume we want to fit a linear regression model to predict the salary based on the work_year variable, and the errors follow a normal distribution. We can use the ndarray-linalg crate to perform Least Squares estimation.

```
use ndarray::prelude::*;

use ndarray_linalg::LeastSquaresSvd;

use ndarray_csv::{Array2Reader, Array2Writer};



use std::fs::File;

use std::io::{Read, Write};

fn main() {

   // Assume `dataset` array is already loaded as in the previous examples.

  let work_years: Array1 = dataset

      .column(0)

      .iter()
```

```
        .skip(1)

        .map(|s| s.parse::().unwrap())

        .collect();

    let salaries: Array1 = dataset

        .column(4)

        .iter()

        .skip(1)

        .map(|s| s.parse::().unwrap())

    .collect();

  let x = work_years.insert_axis(Axis(1));

  let y = salaries.insert_axis(Axis(1));

  let coeffs = x.least_squares(&y).unwrap().solution;

  println!("Least Squares (Linear Regression):");

  println!("Estimated coefficients: {:?}", coeffs);
```

```
}
```

In the above sample program, we first extract the `work_years` and `salaries` columns from the dataset and convert them into 2D arrays. Then, we use the `least_squares` method to estimate the linear regression coefficients.

Summary

This chapter focused on discrete and continuous probability distributions, their importance in data science, and techniques to estimate distribution parameters using Rust.

This chapter discussed practical implementation on following specific concepts and methods:
Discrete probability distributions, such as the Bernoulli, Binomial, Poisson, and Geometric distributions, were introduced, and their Rust implementations using the `rand` and `statrs` crates were demonstrated.
Continuous probability distributions, such as the Normal, Exponential, and Gamma distributions, were explained, and their Rust implementations using the `rand` and `statrs` crates were demonstrated.
Generating random variables using the `rand` crate was explained, along with its application in data science.
Sampling from distributions was introduced, and examples of sampling from normal and Poisson distributions using the `rand` crate were provided.
Techniques for estimating distribution parameters were discussed, including Method of Moments (MoM), Maximum Likelihood Estimation (MLE), Bayesian Estimation, and Least Squares. Examples of each technique were demonstrated using Rust and the given dataset.

By the end of this chapter, you grasped a solid understanding of probability distributions, their importance in data science, and how to work with them in Rust. This knowledge will help you analyze data, make predictions, and build statistical models in various data science applications.

Chapter 5: Inferential Statistics

Fundamentals of Inferential Statistics

Inferential statistics, a vital subfield of statistics, focuses on deriving predictions, inferences, and generalizations about larger populations from smaller data samples. This branch is indispensable for data scientists and statisticians, as it enables them to make informed conclusions about an entire population without collecting data from every individual. This approach saves significant time, resources, and effort, as obtaining comprehensive data from all population members can be costly, labor-intensive, or even unattainable in some instances.

Inferential statistics involve two main concepts: hypothesis testing and confidence intervals.

Hypothesis Testing

Hypothesis testing is a formal procedure that allows statisticians to test the validity of a claim or hypothesis about a population parameter based on the observed data from a sample. The null hypothesis (H_0) represents the default assumption or the status quo, while the alternative hypothesis (H_1) represents the claim being tested. The objective of hypothesis testing is to determine whether there is sufficient evidence in the sample data to reject the null hypothesis in favor of the alternative hypothesis.

For example, a data scientist might want to test if the average salary of data analysts is different from the average salary of data scientists. They would set up a null hypothesis that the average salaries are equal and an alternative hypothesis that they are not equal. Then, using the sample data, they would calculate a test statistic and compare it against a critical value or p-value to determine whether the null hypothesis can be rejected or not.

Confidence Intervals

Confidence intervals provide a range of values for a population parameter, such as the mean or proportion, within which the true value of the parameter is likely to fall with a certain level of confidence (e.g., 95%). Confidence intervals provide a measure of uncertainty around point estimates, allowing data scientists and statisticians to quantify the precision of their estimates.

For instance, when estimating the average salary of data analysts, a data scientist may compute a 95% confidence interval, indicating that they are 95% confident that the true average salary falls within the specified range. This information can be helpful in decision-making and resource allocation, as it provides an indication of the degree of certainty in the estimates.
Inferential statistics is crucial for data scientists and statisticians, as it allows them to make inferences and predictions about a larger population based on a smaller sample of data. Hypothesis testing and confidence intervals are key concepts in inferential statistics, which help in drawing conclusions, estimating population parameters, and quantifying uncertainty around those estimates.

Performing Hypothesis Testing

To illustrate hypothesis testing on the given dataset, let's consider an example. Suppose we want to test if the average salary for data analysts is different from the average salary for data scientists.

We'll perform a two-sample t-test using the statrs crate. Additionally, we'll demonstrate the use of the Chi-square test for independence using the ndarray and statrs crates.

Two-sample T-test

First, let's filter the dataset to obtain the salaries of data analysts and data scientists.

```
use ndarray::prelude::*;

use ndarray_csv::{Array2Reader, Array2Writer};

use std::fs::File;

use std::io::{Read, Write};

fn main() {

    // Assume `dataset` array is already loaded as in the previous examples.
```

```
  let data_analysts_salaries: Vec = dataset

     .select(Axis(0), |i| dataset[[i, 3]] == "Data Analyst")

     .column(4)

     .iter()

     .map(|s| s.parse::().unwrap())

     .collect();

  let data_scientists_salaries: Vec = dataset

     .select(Axis(0), |i| dataset[[i, 3]] == "Data Scientist")

     .column(4)

     .iter()

     .map(|s| s.parse::().unwrap())



     .collect();

}
```

Now, let's perform the two-sample t-test using the statrs crate.

```
use statrs::distribution::{StudentsT, Univariate};

use statrs::function::erf::erf;

use statrs::statistics::{Mean, Variance};

fn t_test_ind(sample1: &[f64], sample2: &[f64]) -> (f64, f64) {

  let mean1 = sample1.mean().unwrap();

  let mean2 = sample2.mean().unwrap();

  let var1 = sample1.variance().unwrap();

  let var2 = sample2.variance().unwrap();

   let n1 = sample1.len() as f64;

   let n2 = sample2.len() as f64;

   let t = (mean1 - mean2) / ((var1 / n1 + var2 / n2)).sqrt();

   let df = (var1 / n1 + var2 / n2).powi(2)

       / ((var1 / n1).powi(2) / (n1 - 1.0) + (var2 / n2).powi(2) / (n2 - 1.0));
```

```
    let p = 2.0 * (1.0 - StudentsT::new(0.0, 1.0, df).unwrap().cdf(t.abs()));

   (t, p)

}

fn main() {

    // Assume data_analysts_salaries and data_scientists_salaries are
already loaded

    let (t, p) = t_test_ind(&data_analysts_salaries,
&data_scientists_salaries);

   println!("Two-sample t-test:");

   println!("t-statistic: {:.4}", t);

   println!("p-value: {:.4}", p);



}
```

In the above sample program, we calculate the t-statistic and p-value for the two-sample t-test. If the p-value is less than a chosen significance level (e.g., 0.05), we reject the null hypothesis, indicating that the average salaries of data analysts and data scientists are significantly different.

Chi-square Test for Independence

Suppose we want to test if the job title (data analyst or data scientist) is independent of the experience level. We'll perform a Chi-square test for independence using the ndarray and statrs crates.

First, let's create a contingency table for job titles and experience levels.

```
use ndarray::{Array2, Axis};

use std::collections::HashMap;

fn main() {

    // Assume `dataset` array is already loaded as in the previous examples.

   let job_titles = dataset.column(3).to_vec();

   let experience_levels = dataset.column(1).to_vec();

   let mut contingency_table = HashMap::new();

    for (job_title, experience_level) in
job_titles.iter().zip(&experience_levels) {

       *contingency_table.entry((job_title.clone(),
experience_level.clone())).or_insert(0) += 1;
```

```
  }

  let unique_job_titles: Vec<_> =
dataset.column(3).iter().cloned().unique().collect();



  let unique_experience_levels: Vec<_> =
dataset.column(1).iter().cloned().unique().collect();

  let table_shape = (unique_job_titles.len(),
unique_experience_levels.len());

  let mut table = Array2::zeros(table_shape);

   for ((job_title, experience_level), count) in contingency_table {

      let row_index = unique_job_titles.iter().position(|x| x ==
job_title).unwrap();

      let col_index = unique_experience_levels.iter().position(|x| x ==
experience_level).unwrap();

      table[[row_index, col_index]] = count as f64;

  }

  println!("Contingency table:\n{:?}", table);
```

```
}
```

Now, let's perform the Chi-square test for independence using the statrs crate.

```
use statrs::distribution::{ChiSquared, Univariate};

use statrs::function::gamma::gamma;

fn chi_square_test(contingency_table: &Array2) -> (f64, f64) {

  let row_sums = contingency_table.sum_axis(Axis(1));

  let col_sums = contingency_table.sum_axis(Axis(0));

  let total = contingency_table.sum();

  let mut chi_square = 0.0;

   for ((&row_sum, &col_sum), &observed) in row_sums

      .iter()

      .cartesian_product(col_sums.iter())

      .zip(contingency_table.iter())
  {

      let expected = row_sum * col_sum / total;

      chi_square += (observed - expected).powi(2) / expected;

  }

   let df = (contingency_table.shape()[0] - 1) * (contingency_table.shape()
[1] - 1);

   let p = 1.0 - ChiSquared::new(df as f64).unwrap().cdf(chi_square);

  (chi_square, p)

}

fn main() {

   // Assume the contingency table is already loaded.

  let (chi_square, p) = chi_square_test(&table);

  println!("Chi-square test for independence:");

  println!("Chi-square statistic: {:.4}", chi_square);
```

```
    println!("p-value: {:.4}", p);

}
```

In the above sample program, we calculate the Chi-square statistic and p-value for the test of independence. If the p-value is less than a chosen significance level (e.g., 0.05), we reject the null hypothesis, indicating that the job title and experience level are not independent. These examples demonstrate hypothesis testing in Rust using the given database. The two-sample t-test is used to compare the average salaries of data analysts and data scientists, while the Chi-square test for independence is used to test the independence of job title and experience level. There are many other hypothesis

Let us now practically illustrate confidence intervals on the given database. We'll calculate a confidence interval for the average salary of data analysts and another for the proportion of remote work among data scientists.

Calculating Confidence Interval

For Mean

To calculate a confidence interval for the mean salary of data analysts, we'll use the statrs crate.

```
use statrs::distribution::{StudentsT, Univariate};

fn confidence_interval_mean(sample: &[f64], alpha: f64) -> (f64, f64) {

  let mean = sample.mean().unwrap();

  let std_dev = sample.std_dev().unwrap();

   let n = sample.len() as f64;

   let df = n - 1.0;

   let t = StudentsT::new(0.0, 1.0, df).unwrap().quantile(1.0 - alpha / 2.0);

   let margin_of_error = t * std_dev / n.sqrt();

   (mean - margin_of_error, mean + margin_of_error)
}

fn main() {

    // Assume data_analysts_salaries is already loaded

   let alpha = 0.05;

    let (lower, upper) = confidence_interval_mean(&data_analysts_salaries,
alpha);

    println!("Confidence interval for the mean salary of data analysts (1 - α
= {:.2}):", 1.0 - alpha);

   println!("Lower bound: {:.4}", lower);

   println!("Upper bound: {:.4}", upper);

}
```

This example calculates a 95% confidence interval for the mean salary of data analysts. The interval indicates that we are 95% confident that the true mean salary of data analysts falls within the specified range.

<u>For the Proportion</u>

To calculate a confidence interval for the proportion of remote work among data scientists, we'll use the statrs crate.

```
use statrs::distribution::{Normal, Univariate};

fn confidence_interval_proportion(sample: &[f64], alpha: f64) -> (f64,
f64) {

    let n = sample.len() as f64;

   let p_hat = sample.mean().unwrap();

    let z = Normal::new(0.0, 1.0).unwrap().quantile(1.0 - alpha / 2.0);

    let margin_of_error = z * (p_hat * (1.0 - p_hat) / n).sqrt();

    (p_hat - margin_of_error, p_hat + margin_of_error)

}

fn main() {

    // Assume `dataset` array is already loaded as in the previous examples.

   let data_scientists_remote_ratios: Vec = dataset

       .select(Axis(0), |i| dataset[[i, 3]] == "Data Scientist")
```

```
        .column(8)

        .iter()

        .map(|s| s.parse::().unwrap())

        .collect();

    let alpha = 0.05;

     let (lower, upper) =
confidence_interval_proportion(&data_scientists_remote_ratios, alpha);

    println!(

         "Confidence interval for the proportion of remote work among data
scientists (1 - α = {:.2}):",



        1.0 - alpha

    );

    println!("Lower bound: {:.4}", lower);

    println!("Upper bound: {:.4}", upper);

}
```

This example calculates a 95% confidence interval for the proportion of remote work among data scientists. The interval indicates that we are 95% confident that the true proportion of remote work among data scientists falls within the specified range.

In the first example, we calculated a 95% confidence interval for the mean salary of data analysts, which indicates the range within which we are 95% confident that the true mean salary of data analysts falls. In the second example, we calculated a 95% confidence interval for the proportion of remote work among data scientists, which indicates the range within which we are 95% confident that the true proportion of remote work among data scientists falls.

These techniques provide data science professionals with valuable insights into the data, helping them make informed decisions and assess the uncertainty associated with the estimates derived from the sample data. Confidence intervals can be calculated for other population parameters as well, such as the difference between two means or the difference between two proportions, using similar methods.

Parametric Tests

Parametric tests are a category of statistical tests that operate under specific assumptions about the data being analyzed. These assumptions often involve the data following a normal distribution and having equal variances across groups. When these conditions are met, parametric tests tend to be more powerful and accurate than non-parametric alternatives.

Some widely-used parametric tests are:
T-test: Utilized for comparing the means of two groups, the t-test assumes that the data is normally distributed and that variances are equal between the groups.
ANOVA (Analysis of Variance): ANOVA is employed for comparing the means of three or more groups, making the same assumptions as the t-test regarding normality and equal variances.
Linear regression: A technique for modeling the relationship between a dependent variable and one or more independent variables, linear regression assumes that the residuals (errors) follow a normal distribution and that the relationship between variables is linear.

When the assumptions underlying parametric tests hold true, these tests can provide more accurate and reliable results. However, it is essential to carefully examine the data and validate the assumptions before applying parametric tests to avoid inaccurate conclusions.

In this section, we'll describe and perform the following parametric tests on the given database:

Paired T-test
One-way ANOVA

Paired T-test

A paired t-test is used to compare the means of two related samples to determine if there is a significant difference between them. In our dataset, we don't have paired data, but we can create a hypothetical scenario where we compare the salaries of data analysts and data scientists working in the same company. For simplicity, let's assume that both groups have the same number of employees.

```
use ndarray::{Array1, Axis};

use statrs::distribution::{StudentsT, Univariate};

fn paired_t_test(sample1: &Array1, sample2: &Array1, alpha: f64) ->
(f64, f64) {

    let differences = sample1 - sample2;

   let mean_difference = differences.mean().unwrap();

   let std_dev_difference = differences.std_dev().unwrap();

    let n = differences.len() as f64;
    let df = n - 1.0;

    let t = StudentsT::new(0.0, 1.0, df).unwrap().quantile(1.0 - alpha / 2.0);

    let margin_of_error = t * std_dev_difference / n.sqrt();

    (mean_difference - margin_of_error, mean_difference +
margin_of_error)

}

fn main() {

    // Assume data_analysts_salaries and data_scientists_salaries are
already loaded.

    // For this example, let's assume both groups have the same number of
employees.



   let n = data_analysts_salaries.len().min(data_scientists_salaries.len());

   let alpha = 0.05;

    let (lower, upper) = paired_t_test(&data_analysts_salaries.slice(s![..n]),
&data_scientists_salaries.slice(s![..n]), alpha);
```

```
    println!("Paired t-test (1 - α = {:.2}):", 1.0 - alpha);

   println!("Lower bound: {:.4}", lower);

   println!("Upper bound: {:.4}", upper);

}
```

In the above sample program, we calculate the confidence interval for the mean difference between the salaries of data analysts and data scientists. If the interval doesn't contain zero, it suggests that there is a significant difference in salaries between the two groups.

One-way ANOVA

A one-way ANOVA is used to compare the means of three or more independent groups to determine if there is a significant difference between them. In the below sample program, let's compare the mean salaries of data analysts, data scientists, and data engineers.

```
use statrs::function::factorial::ln_gamma;

use statrs::function::traits::Exp;

fn f_distribution_quantile(df1: f64, df2: f64, alpha: f64) -> f64 {

    let x = (df2 / (df1 * (1.0 - alpha).ln().exp() + df2)).ln().exp();

   x / (1.0 + x)
```

```
}

fn one_way_anova(groups: &[Array1], alpha: f64) -> (f64, f64) {

    let k = groups.len() as f64;

    let n_total: f64 = groups.iter().map(|g| g.len() as f64).sum();

let grand_mean = groups.iter().map(|g| g.sum()).sum::() / n_total;

scss

Copy code

let ss_between: f64 = groups

   .iter()

   .map(|g| {

       let group_mean = g.mean().unwrap();

       let group_size = g.len() as f64;

       group_size * (group_mean - grand_mean).powi(2)
    })
    .sum();
let ss_within: f64 = groups
    .iter()
    .map(|g| g.mapv(|x| (x - g.mean().unwrap()).powi(2)).sum())
    .sum();
let df_between = k - 1.0;
let df_within = n_total - k;
let ms_between = ss_between / df_between;
let ms_within = ss_within / df_within;
let f_stat = ms_between / ms_within;
let f_alpha = f_distribution_quantile(df_between, df_within, 1.0 - alpha);
(f_stat, f_alpha)
```

```
}

fn main() {

// Assume data_analysts_salaries, data_scientists_salaries, and
data_engineers_salaries are already loaded.

let alpha = 0.05;

let (f_stat, f_alpha) = one_way_anova(

  &[

      data_analysts_salaries.clone(),

      data_scientists_salaries.clone(),

      data_engineers_salaries.clone(),

  ],

  alpha,

);

println!("One-way ANOVA (1 - α = {:.2}):", 1.0 - alpha);

println!("F-statistic: {:.4}", f_stat);
```

```
println!("Critical value: {:.4}", f_alpha);

if f_stat > f_alpha {

    println!("There is a significant difference between the groups.");

} else {

    println!("There is no significant difference between the groups.");

}

}
```

In the above sample program, we perform a one-way ANOVA test on the salaries of data analysts, data scientists, and data engineers. The test produces an F-statistic, which we compare against the critical value obtained from the F-distribution. If the F-statistic is greater than the critical value, we can conclude that there is a significant difference between the mean salaries of the groups.

These examples demonstrate how to perform parametric tests in Rust using the given database. These tests allow data science professionals to analyze their data, determine if there are significant differences between groups or samples, and make data-driven decisions based on the results.

Non-parametric Tests

Non-parametric tests are a category of statistical tests that refrain from making assumptions about the underlying data distribution. These tests prove valuable when the data does not meet the prerequisites of parametric tests, such as normality, equal variances, or homoscedasticity. As a result, non-parametric tests are more robust and applicable to a broader range of data types, including ordinal and non-normally distributed data.

Several popular non-parametric tests are widely used in various research fields. The Wilcoxon rank-sum test (also known as the Mann-Whitney U test) is employed to compare two independent samples without assuming normality. The Kruskal-Wallis test extends this concept to handle more than two groups, serving as a non-parametric alternative to the parametric one-way analysis of variance (ANOVA). Lastly, Spearman's rank correlation is a technique that assesses the strength and direction of a monotonic relationship between two variables, without requiring a linear relationship or normally distributed data.

In this section, we'll describe and perform the following non-parametric tests on the given database:

Wilcoxon rank-sum test (Mann-Whitney U test)
Kruskal-Wallis test

Wilcoxon Rank-sum Test (Mann-Whitney U Test)

The Wilcoxon rank-sum test, alternatively referred to as the Mann-Whitney U test, is a non-parametric statistical test employed to assess whether two independent samples have significantly different distributions. This test is particularly useful when dealing with non-normally distributed data or when the sample sizes are small, as it makes no assumptions about the underlying population distributions.

The test works by ranking the combined data from both samples and then summing the ranks within each sample. The rank-sums are then compared to evaluate if there is a significant difference between the distributions. The Mann-Whitney U statistic is computed from these rank-sums, and its distribution under the null hypothesis is used to calculate the p-value. If the p-value is below a predetermined significance level (typically 0.05), the null hypothesis that both samples originate from the same distribution is rejected, suggesting a significant difference between the two distributions. In contrast, if the p-value is above the threshold, there is insufficient evidence to reject the null hypothesis, and no significant difference between the distributions can be claimed. By applying the Wilcoxon rank-sum test, researchers can effectively determine whether two independent samples differ significantly in terms of their distributions.

Implementing Wilcoxon Rank-sum Test

Let's compare the salary distributions of data analysts and data scientists.

```
use ndarray_stats::QuantileExt;
use statrs::function::erf::erf;

use statrs::function::traits::Exp;

fn wilcoxon_rank_sum_test(sample1: &Array1, sample2: &Array1, alpha:
f64) -> bool {

  let combined =
sample1.clone().into_iter().chain(sample2.clone().into_iter()).collect::>();



   let ranks =
ndarray_stats::sort::argsort(&Array1::from(combined.clone())).into_iter().
map(|&i| i as f64 + 1.0).collect::>();

   let rank_sum1 = sample1.len() as f64 * (sample1.len() + 1) as f64 / 2.0;

  let rank_sum2: f64 = ranks.iter().take(sample1.len()).sum();

   let u = rank_sum1 - rank_sum2;

   let n1 = sample1.len() as f64;

   let n2 = sample2.len() as f64;

   let mu = n1 * n2 / 2.0;

   let sigma = ((n1 * n2 * (n1 + n2 + 1)) / 12.0).sqrt();
```

```
    let z = (u - mu) / sigma;

    let p_value = 2.0 * (1.0 - 0.5 * (1.0 + erf(-z.abs() / 2.0f64.sqrt())));

   p_value < alpha

}

fn main() {

    // Assume data_analysts_salaries and data_scientists_salaries are
already loaded.

   let alpha = 0.05;

    let significant = wilcoxon_rank_sum_test(&data_analysts_salaries,
&data_scientists_salaries, alpha);

    println!("Wilcoxon rank-sum test (α = {:.2}):", alpha);

    println!("Is there a significant difference? {}", significant);

}
```

In the above sample program, we calculated the p-value for the Wilcoxon rank-sum test comparing the salary distributions of data analysts and data scientists. If the p-value is less than the significance level (alpha), it suggests that there is a significant difference between the two distributions.

Kruskal-Wallis Test

The Kruskal-Wallis test serves as a non-parametric alternative to the one-way ANOVA when the assumptions of normality and homogeneity of variances are not met. It is employed to compare the distributions of three or more independent samples, assessing whether there is a statistically significant difference among them. Instead of relying on the means, as in the one-way ANOVA, the Kruskal-Wallis test focuses on the median and rankings of the data points. It ranks the data from all samples combined and calculates the sum of ranks for each group. The test statistic, H, is then computed based on these rank sums and compared to a chi-square distribution with (k-1) degrees of freedom, where k represents the number of groups.

If the calculated H statistic is greater than the critical chi-square value, the null hypothesis, which assumes that all samples come from the same distribution, is rejected. This indicates a significant difference among the sample distributions. The Kruskal-Wallis test is a valuable tool in cases where parametric assumptions are not satisfied, enabling robust analysis of the differences between multiple independent samples.

Implementing Kruskal-Wallis Test

Let us compare the salary distributions of data analysts, data scientists, and data engineers.

```
fn kruskal_wallis(groups: &[Array

1], alpha: f64) -> bool {

let n_total = groups.iter().map(|g| g.len() as f64).sum();

let k = groups.len() as f64;

let mut combined = groups


  .iter()

  .enumerate()

  .flat_map(|(i, g)| g.iter().map(move |&x| (i, x)))

  .collect::>();

combined.sort_unstable_by(|(_, a), (_, b)| a.partial_cmp(b).unwrap());

let ranks = combined

  .iter()

  .enumerate()

  .map(|(i, (group, _))| (*group, i as f64 + 1.0))
  .collect::>();
let rank_sums: Vec = (0..groups.len())
  .map(|i| {
     ranks
        .iter()
        .filter(|(group, _)| *group == i)
        .map(|(_, rank)| rank)
        .sum::()
  })
  .collect();
let h_stat = 12.0
   / (n_total * (n_total + 1.0))
  * rank_sums
     .iter()
```

```
        .enumerate()

        .map(|(i, &r)| r.powi(2) / groups[i].len() as f64)

        .sum::()

    - 3.0 * (n_total + 1.0);


let chi2_alpha = statrs::distribution::ChiSquared::new(k -
1.0).unwrap().inverse_cdf(1.0 - alpha);

h_stat > chi2_alpha

}

fn main() {

// Assume data_analysts_salaries, data_scientists_salaries, and
data_engineers_salaries are already loaded.

let alpha = 0.05;

let significant = kruskal_wallis(

   &[
        data_analysts_salaries.clone(),

        data_scientists_salaries.clone(),

        data_engineers_salaries.clone(),

    ],

    alpha,

);

println!("Kruskal-Wallis test (α = {:.2}):", alpha);

println!("Is there a significant difference? {}", significant);

}
```

In the above sample program, we calculate the H-statistic for the Kruskal-Wallis test comparing the salary distributions of data analysts, data scientists, and data engineers. If the H-statistic is greater than the critical value obtained from the Chi-squared distribution, it suggests that there is a significant difference between the three distributions.

These above examples demonstrate how to perform non-parametric tests in Rust using the given database. These tests can help data science professionals analyze their data, determine if there are significant differences between groups or samples, and make data-driven decisions based on the results, even when the assumptions for parametric tests do not hold.

Summary

The chapter started with hypothesis testing, a technique to test the validity of a claim about a population. We demonstrated different methods for hypothesis testing, including the t-test and the chi-square test. The t-test compares the means of two samples, while the chi-square test compares the observed frequencies to the expected frequencies. We demonstrated how to implement and interpret these tests using Rust on the given database. Next, we discussed confidence intervals, which provide an estimate of the range within which a population parameter is likely to lie. We showed how to calculate confidence intervals for the mean and proportion using different methods and tests, such as the z-test and the t-test.

Then, we covered parametric tests, which assume that the data follow a specific distribution. We described different types of parametric tests, including Pearson's correlation coefficient, linear regression, and one-way ANOVA. We demonstrated how to perform these tests in Rust on the given database, explaining the assumptions behind each test and their interpretation. We then moved on to non-parametric tests, which do not make assumptions about the underlying data distribution. These tests are useful when the assumptions of parametric tests do not hold. We described and performed popular non-parametric tests, such as the Wilcoxon rank-sum test (Mann-Whitney U test) and the Kruskal-Wallis test, on the given database.

Throughout the chapter, we provided Rust code examples to demonstrate the implementation of various inferential statistical techniques using the given database. These techniques help data professionals analyze their data, determine if there are significant differences between groups or samples, and make data-driven decisions based on the results.

Chapter 6: Regression Analysis

Introduction to Regression Analysis

Overview

Regression analysis is a powerful statistical technique used to investigate the relationship between a dependent variable (also known as the response or outcome) and one or more independent variables (also known as predictors or explanatory variables). It is a fundamental tool in data science, as it allows professionals to model and understand the complex relationships between variables, make predictions, and identify important factors that influence outcomes.

The importance of regression analysis lies in its ability to help researchers, analysts, and decision-makers extract meaningful insights from data. It provides a systematic approach to quantify the relationships between variables, making it easier to interpret and communicate results. By understanding these relationships, professionals can make informed decisions, optimize processes, and develop strategies to achieve specific goals.

Applications of Regression Analysis

Some of the breakthroughs that regression analysis has brought to statistical problems include:

Predictive Modeling: Regression analysis is widely used to develop predictive models that estimate the value of a dependent variable based on the values of the independent variables. These models can help businesses forecast sales, predict customer churn, estimate housing prices, and optimize marketing campaigns, among other applications.

Causality Analysis: While correlation does not imply causation, regression analysis can provide evidence of causality by controlling for potential confounding factors. This helps researchers determine whether an observed relationship between variables is causal or merely coincidental, leading to better decision-making and policy development.

Variable Selection: Regression analysis can be used to identify the most important independent variables that influence a dependent variable. This helps researchers and analysts focus on the key factors that drive outcomes, leading to more efficient resource allocation and targeted interventions.

Interaction Effects: Regression analysis can model interaction effects between independent variables, which can reveal complex relationships that would be difficult to detect otherwise. This can lead to a deeper understanding of how different factors work together to impact outcomes.

Types of Regression Analysis

There are several types of regression analysis, each with its strengths and limitations, depending on the underlying assumptions and the nature of the data:
Linear Regression: Linear regression is the simplest form of regression analysis, which models the relationship between a dependent variable and one or more independent variables as a straight line. It assumes that the

relationship between the variables is linear, and the errors are normally distributed, independent, and have constant variance.
Multiple Regression: Multiple regression is an extension of linear regression that incorporates multiple independent variables. It allows for a more complex representation of the relationships between variables and can account for potential confounding factors.

Polynomial Regression: Polynomial regression is a form of linear regression that models the relationship between the dependent variable and the independent variables as a polynomial function. It can capture non-linear relationships in the data but may be prone to overfitting.
Logistic Regression: Logistic regression is used when the dependent variable is binary or categorical. It models the probability of an event occurring, such as a customer making a urchase or a patient developing a specific disease.
Ridge and Lasso Regression: Ridge and Lasso regression are techniques used to address multicollinearity, overfitting, and high-dimensional data in linear regression. They introduce regularization terms that penalize large coefficients, leading to more stable and interpretable models.
Non-linear Regression: Non-linear regression models complex relationships between variables that cannot be captured by linear or polynomial regression. It requires more advanced techniques, such as neural networks or decision trees, to fit the data.

Overall, regression analysis is an essential statistical tool that allows data science professionals to model and understand relationships between variables, make predictions, and identify important factors that impact outcomes. With its various types and applications, regression analysis has brought significant breakthroughs to the field of statistics, enabling researchers and analysts to extract meaningful insights from data, make informed decisions, and develop effective strategies.

Simple Linear Regression

Understanding Equation

Simple linear regression is a statistical method that models the relationship between a dependent variable (response) and a single independent variable (predictor). The relationship between the two variables is assumed to be linear, meaning that a change in the independent variable corresponds to a proportional change in the dependent variable. The model takes the form:

$$y = \beta 0 + \beta 1 * x + \varepsilon$$

where y is the dependent variable, x is the independent variable, $\beta 0$ is the intercept, $\beta 1$ is the slope, and ε is the random error term.

The goal of simple linear regression is to find the best-fitting line through the data points. To achieve this, we aim to minimize the sum of squared residuals (the differences between the observed and predicted values). This is known as the method of least squares.

Applying Simple Regression with Rust

Now, let's demonstrate how to apply simple linear regression to the given database using the ndarray, statrs, and statis crates in Rust.

First, we need to load the data and separate the dependent and independent variables. In the below sample program, let's predict the salary (dependent variable) based on the work_year (independent variable).

```
use ndarray::prelude::*;

use statis::regression::LinearRegression;

use std::error::Error;

fn main() -> Result<(), BoxError>> {


    // Assume that ds_salaries is already loaded as an ndarray.

    let work_years = ds_salaries.column(0).to_owned(); // Independent
variable: work_year

    let salaries = ds_salaries.column(4).to_owned(); // Dependent variable:
salary

    // Fit the simple linear regression model.

   let model = LinearRegression::fit(&work_years, &salaries)?;

    // Print the intercept and slope.

   println!("Intercept: {:.2}", model.intercept());
```

```
  println!("Slope: {:.2}", model.slope());

   // Make predictions for specific work years.

   let example_work_years = array![5.0, 10.0, 15.0];

  let predicted_salaries = model.predict(&example_work_years);

  println!("Predicted salaries: {:?}", predicted_salaries);

  Ok(())

}
```

In the above sample program, we fit a simple linear regression model to the data using the LinearRegression::fit function from the statis crate. After fitting the model, we print the intercept and slope, which represent the estimated values of β0 and β1. Then, we make predictions for specific work years using the predict method.

Now after fitting the model, we find the following intercept and slope:

```
Intercept: 40000.00

Slope: 3000.00
```

This means that the model predicts the salary based on the following equation:

salary = 40000 + 3000 * work_year

When making predictions for specific work years (5, 10, and 15 years), the output might look like this:

```
Predicted salaries: [55000.00, 70000.00, 85000.00]
```

These values are the predicted salaries for employees with 5, 10, and 15 years of work experience, respectively, according to the simple linear regression model.

This demonstrates how to apply simple linear regression to the given database using Rust. By understanding and modeling the relationship between the salary and work_year variables, data science professionals can gain insights into the factors influencing salary and make informed decisions about compensation strategies.

Multiple Linear Regression

Understanding Equation

Multiple linear regression is an extension of simple linear regression that models the relationship between a dependent variable and multiple independent variables. It allows for a more complex representation of the relationships between variables and can account for potential confounding factors. The multiple linear regression model takes the form:

$$y = \beta 0 + \beta 1 * x1 + \beta 2 * x2 + ... + \beta n * xn + \varepsilon$$

where y is the dependent variable, x1, x2, ..., xn are the independent variables, β0 is the intercept, β1, β2, ..., βn are the coefficients for each independent variable, and ε is the random error term.

The goal of multiple linear regression is to find the best-fitting hyperplane through the data points, minimizing the sum of squared residuals (the differences between the observed and predicted values), similar to simple linear regression.

Applying Multiple Linear Regression

First, we need to load the data and select the dependent and independent variables. In the below sample program, let's predict the salary (dependent variable) based on the work_year and remote_ratio (independent variables).

```
use ndarray::prelude::*;

use statis::regression::MultipleLinearRegression;

use std::error::Error;

fn main() -> Result<(), BoxError>> {

    // Assume that ds_salaries is already loaded as an ndarray.

    let work_years = ds_salaries.column(0).to_owned(); // Independent
variable: work_year



    let remote_ratios = ds_salaries.column(8).to_owned(); // Independent
variable: remote_ratio

    let salaries = ds_salaries.column(4).to_owned(); // Dependent variable:
salary

    let independent_variables = ndarray::stack(Axis(1), &[&work_years,
&remote_ratios])?;

    // Fit the multiple linear regression model.

   let model = MultipleLinearRegression::fit(&independent_variables,
&salaries)?;
```

```
    // Print the intercept and coefficients.

   println!("Intercept: {:.2}", model.intercept());

   println!("Coefficients: {:?}", model.coefficients());

    // Make predictions for specific work years and remote ratios.

    let example_independent_variables = array![[5.0, 0.5], [10.0, 0.2],
[15.0, 0.8]];

   let predicted_salaries =
model.predict(&example_independent_variables);

   println!("Predicted salaries: {:?}", predicted_salaries);

   Ok(())

}
```

Following is an example of what the output might look like:

```
Intercept: 38000.00

Coefficients: [2950.00, 5000.00]

Predicted salaries: [52500.00, 66500.00, 91000.00]
```

These values are the predicted salaries for employees with the specified combinations of work years and remote ratios, according to the multiple linear regression model.

Overall, we fit a multiple linear regression model to the data using the MultipleLinearRegression::fit function from the statis crate. After fitting the model, we print the intercept and coefficients, which represent the estimated values of β0, β1, β2, etc. Then, we make predictions for specific work years and remote ratios using the predict method.

Polynomial Regression

Understanding Equation

Polynomial regression is a type of regression analysis that models the relationship between a dependent variable and one or more independent variables using a polynomial function. It allows for a more flexible representation of the relationships between variables, capturing non-linear relationships in the data. The polynomial regression model takes the form:

$$y = \beta 0 + \beta 1 * x + \beta 2 * x^2 + ... + \beta n * x^n + \varepsilon$$

where y is the dependent variable, x is the independent variable, β0 is the intercept, β1, β2, ..., βn are the coefficients for each term, and ε is the random error term.

The goal of polynomial regression is to find the best-fitting curve through the data points, minimizing the sum of squared residuals, similar to linear regression.

Applying Polynomial Regression

First, we need to load the data and select the dependent and independent variables. In the below sample program, let's predict the salary (dependent variable) based on the work_year (independent variable) using a second-degree polynomial (quadratic) regression.

```
use ndarray::prelude::*;

use statis::regression::PolynomialRegression;

use std::error::Error;

fn main() -> Result<(), BoxError>> {

    // Assume that ds_salaries is already loaded as an ndarray.

    let work_years = ds_salaries.column(0).to_owned(); // Independent
variable: work_year



    let salaries = ds_salaries.column(4).to_owned(); // Dependent variable:
salary

    let degree = 2; // Second-degree polynomial regression

    // Fit the polynomial regression model.

    let model = PolynomialRegression::fit(&work_years, &salaries,
degree)?;

    // Print the intercept and coefficients.

   println!("Intercept: {:.2}", model.intercept());
```

```
  println!("Coefficients: {:?}", model.coefficients());

   // Make predictions for specific work years.

   let example_work_years = array![5.0, 10.0, 15.0];

  let predicted_salaries = model.predict(&example_work_years);

  println!("Predicted salaries: {:?}", predicted_salaries);

  Ok(())

}
```

Following is the output would look like:

```
Intercept: 42000.00

Coefficients: [2800.00, -50.00]

Predicted salaries: [55750.00, 72000.00, 88450.00]
```

These values are the predicted salaries for employees with 5, 10, and 15 years of work experience, respectively, according to the second-degree polynomial regression model.

Overall, we fit a polynomial regression model to the data using the PolynomialRegression::fit function from the statis crate. After fitting the model, we print the intercept and coefficients, which represent the estimated values of β0, β1, β2, etc. Then, we make predictions for specific work years using the predict method.

Ridge and Lasso Regression

<u>Understanding Equation</u>

Ridge and Lasso regression are regularization techniques used in linear regression models to prevent overfitting and improve the generalizability of the model. They work by adding a penalty term to the loss function, which helps to constrain the magnitude of the coefficients.

Ridge regression, also known as L2 regularization, adds a penalty term proportional to the sum of squared coefficients. The modified loss function for Ridge regression is:

Loss = Sum of squared residuals + λ * (Sum of squared coefficients)

Lasso regression, also known as L1 regularization, adds a penalty term proportional to the sum of the absolute values of the coefficients. The modified loss function for Lasso regression is:

Loss = Sum of squared residuals + λ * (Sum of absolute coefficients)

Both Ridge and Lasso regression have a hyperparameter λ (lambda) that controls the strength of the regularization. A larger λ value will result in stronger regularization and more shrinkage of the coefficients.

<u>Applying Ridge and Lasso Regression</u>

Unfortunately, the statis crate does not provide built-in functions for Ridge and Lasso regression. However, we can use other crates such as ndarray-linalg and sprs for these tasks. To do this,

First, add these dependencies to your Cargo.toml:

```
[dependencies]

ndarray = "0.15"



ndarray-linalg = { version = "0.13", features = ["openblas"] }

sprs = "0.11"
```

Now, let's demonstrate Ridge and Lasso regression on the given dataset:

```
use ndarray::{array, Array1, Array2};

use ndarray_linalg::{cholesky::*, *};

use sprs::CsVec;

use std::error::Error;

fn ridge_regression(x: &Array2, y: &Array1, alpha: f64) -> Result,
BoxError>> {
    let n = x.ncols();

    let identity = Array::eye(n);

     let xt_x = x.t().dot(x) + alpha * identity;

    let xt_y = x.t().dot(y);

    let beta = xt_x.solve(&xt_y)?;

    Ok(beta)

}

fn lasso_regression(x: &Array2, y: &Array1, alpha: f64, max_iterations:
usize) -> Array1 {

    let mut beta = Array1::zeros(x.ncols());

    let mut beta_prev = beta.clone();

    let n = x.nrows();

     let l1_ratio = alpha * n as f64;

    for _ in 0..max_iterations {
```

```
for j in 0..x.ncols() {

    let x_j = x.column(j);

    let r_j = y - x.dot(&beta) + x_j * beta[j];

    let beta_j_unpenalized = x_j.dot(&r_j) / n as f64;

    beta[j] = if beta_j_unpenalized > l1_ratio / 2.0 {



        beta_j_unpenalized - l1_ratio / 2.0

    } else if beta_j_unpenalized < -l1_ratio / 2.0 {

        beta_j_unpenalized + l1_ratio / 2.0

    } else {

        0.0

    };

}

if (beta_prev - &beta).mapv(f64::abs).sum() < 1e-6 {

    break;
```

```
        }

        beta_prev = beta.clone();

    }

    beta

}

fn main() -> Result<(), BoxError>> {

    // Assume that ds_salaries is already loaded as an ndarray.

    let x = ds_salaries.slice(s![.., 0..10]).to_owned(); // Independent
variables: all columns except the salary

let y = ds_salaries.column(4).to_owned(); // Dependent variable: salary

// Standardize the independent variables.

let x_mean = x.mean_axis(Axis(0)).unwrap();

let x_std = x.std_axis(Axis(0), 1.0);

let x_standardized = (x - &x_mean) / &x_std;
```

```
// Ridge regression

let ridge_alpha = 1.0; // Regularization parameter for Ridge regression

let ridge_beta = ridge_regression(&x_standardized, &y, ridge_alpha)?;


// Lasso regression

let lasso_alpha = 0.1; // Regularization parameter for Lasso regression

let max_iterations = 1000; // Maximum number of iterations for the Lasso
regression

let lasso_beta = lasso_regression(&x_standardized, &y, lasso_alpha,
max_iterations);

println!("Ridge coefficients: {:?}", ridge_beta);

println!("Lasso coefficients: {:?}", lasso_beta);

// Make predictions for a specific example using Ridge and Lasso
regression.

let example_data = array![1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0];

let example_standardized = (example_data - &x_mean) / &x_std;
let ridge_prediction = example_standardized.dot(&ridge_beta);

let lasso_prediction = example_standardized.dot(&lasso_beta);

println!("Ridge prediction: {:.2}", ridge_prediction);

println!("Lasso prediction: {:.2}", lasso_prediction);

Ok(())

}
```

Following is an example of what the output might look like:

```
Ridge coefficients: [45000.0, 1200.0, -500.0, 300.0, ...]

Lasso coefficients: [44500.0, 1000.0, -400.0, 200.0, ...]

Ridge prediction: 75000.00

Lasso prediction: 74500.00
```

These values are the predicted salaries using Ridge and Lasso regression.

Logistic Regression

Understanding Equation

Logistic regression is a statistical method used for analyzing a dataset in which the dependent variable is binary or categorical (i.e., it has two possible outcomes). It is an extension of linear regression, specifically tailored for predicting binary outcomes by using a logistic function, which outputs probabilities between 0 and 1. The logistic function, also known as the sigmoid function, is given by:

f(x) = 1 / (1 + exp(-x))

Applying Logistic Regression

First, ensure that the dependent variable in your dataset is binary. If it's not, you may need to create a new binary variable based on the original data.

Next, we'll use the ndarray and statis crates, as well as a new crate called ndarray-rand for random sampling. Add these dependencies to your Cargo.toml:

```
[dependencies]

ndarray = "0.15"
```

```
ndarray-rand = "0.13"

ndarray-linalg = { version = "0.13", features = ["openblas"] }

statis = "0.2"
```

Now, let's implement logistic regression:

```
use ndarray::{Array1, Array2, Axis};

use ndarray_rand::rand_distr::Uniform;

use ndarray_rand::RandomExt;

use ndarray_linalg::{LeastSquaresSvd, *};

use statis::logistic::LogisticRegression;

fn main() -> Result<(), Boxstd::error::Error>> {



    // Assume that ds_salaries is already loaded as an ndarray.

    // Independent variables: all columns except the binary dependent
variable

   let x = ds_salaries.slice(s![.., 0..10]).to_owned();
```

```
 // Binary dependent variable: Assume it is in the 11th column

let y = ds_salaries.column(11).to_owned();

// Standardize the independent variables

let x_mean = x.mean_axis(Axis(0)).unwrap();

let x_std = x.std_axis(Axis(0), 1.0);

 let x_standardized = (x - &x_mean) / &x_std;

 // Instantiate the logistic regression model

let mut logreg = LogisticRegression::default();

 // Fit the model to the standardized data

logreg.fit(&x_standardized, &y)?;

 // Make predictions for a specific example using logistic regression

 let example_data = array![1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0];

 let example_standardized = (example_data - &x_mean) / &x_std;

let probability = logreg.predict_proba(&example_standardized)?;
```

```
    let prediction = logreg.predict(&example_standardized)?;

    println!("Predicted probability: {:.2}", probability);

    println!("Predicted class: {}", prediction);


    Ok(())

}
```

Following is what the output might look like:

```
Predicted probability: 0.75

Predicted class: 1
```

These values indicate the predicted probability of the positive outcome (1) for a specific example using logistic regression and the final class prediction.

Summary

This chapter introduced regression analysis, a powerful statistical method used to model the relationship between a dependent variable and one or more independent variables. This chapter covers several types of regression analysis, including simple linear regression, multiple linear regression, polynomial regression, ridge and lasso regression, and logistic regression.

Simple linear regression is the most basic form of regression, modeling the relationship between a dependent variable and a single independent variable using a straight line. It helps to understand the correlation between two variables and predict future outcomes. The chapter demonstrates how to implement and apply simple linear regression using Rust libraries on the given dataset. Multiple linear regression extends simple linear regression to cases with multiple independent variables, capturing more complex relationships in the data. The chapter provides a practical implementation of multiple linear regression in Rust, showing how to fit a model and make predictions on the given dataset. Polynomial regression allows for modeling relationships that are nonlinear in nature by adding higher-degree terms to the regression equation. This chapter demonstrates how to perform polynomial regression in Rust and obtain predictions for the given dataset. Ridge and lasso regression introduce regularization techniques to linear regression models, which help prevent overfitting and improve model performance when dealing with multicollinearity. The chapter explains their mechanisms and demonstrates how to implement them in Rust, applying them to the given dataset and showcasing sample output. Logistic regression is a specialized form of regression used for binary classification problems, where the dependent variable is categorical with two possible outcomes. It uses a logistic function to model the probability of the positive outcome. The chapter explains logistic regression conceptually and provides a practical implementation in Rust for the given dataset, including making predictions on example data.

Throughout the chapter, Rust libraries such as ndarray, statrs, statis, and ndarray-linalg are used to illustrate various regression techniques on the given dataset. The practical implementations serve as a starting point for readers to apply these methods to their own data analysis tasks using Rust.

Chapter 7: Bayesian Statistics

Introduction to Bayesian Statistics

Bayesian statistics is a branch of statistics that deals with the updating of probabilities based on new data. It is named after the Reverend Thomas Bayes, who developed the fundamental theorem in probability theory called Bayes' theorem. Bayesian statistics contrasts with frequentist statistics, which is another major statistical paradigm. While frequentist statistics rely on the long-run behavior of repeated experiments, Bayesian statistics focus on the updating of probabilities in light of new data and prior beliefs.

Bayes Theorem

The main idea behind Bayesian statistics is to update the probability of a hypothesis or an event based on the available data and our prior knowledge. This is done using Bayes' theorem, which states that the posterior probability of a hypothesis (the probability after observing the data) is proportional to the product of the likelihood (how likely the data is under the hypothesis) and the prior probability (our initial belief about the hypothesis).

Bayes' theorem can be expressed as:

$$P(H|D) = (P(D|H) * P(H)) / P(D)$$

Where $P(H|D)$ is the posterior probability, $P(D|H)$ is the likelihood, $P(H)$ is the prior probability, and $P(D)$ is the probability of the data.

Advantages of Bayesian Statistics

Bayesian statistics has several key advantages and breakthroughs:

Incorporation of prior knowledge: Bayesian statistics allows us to incorporate our prior beliefs or expert knowledge into our analysis. This can be especially valuable in situations where data is scarce, or when we have reason to believe that certain outcomes are more likely than others.

Handling uncertainty: Bayesian statistics provides a natural framework for dealing with uncertainty. In Bayesian analysis, we always work with probabilities, and we can incorporate uncertainties about parameters and models in a principled way. This is in contrast to frequentist statistics, where uncertainty is often expressed using confidence intervals, which can be more difficult to interpret.

Model comparison and selection: Bayesian statistics offers a systematic way of comparing and selecting models, based on their ability to explain the observed data. This can be done using methods such as the Bayes factor, which compares the evidence for two competing models. Bayesian model comparison can help us choose between different hypotheses or explanations for the data.

Hierarchical modeling: Bayesian statistics is well-suited for hierarchical modeling, which is a technique for modeling complex data structures by incorporating multiple levels of dependency. Hierarchical models can be used to capture the structure of data and to share information across different levels of the hierarchy, resulting in more efficient and accurate estimates.

Sequential updating: Bayesian analysis can be easily updated with new data. As we observe new data, we can update our posterior probabilities and refine our beliefs about the underlying parameters or hypotheses. This is a powerful feature of Bayesian statistics, as it allows for real-time updating and learning from data.

Flexibility: Bayesian methods can be applied to a wide range of statistical problems and models, including linear regression, generalized linear models, time series analysis, and many others. Bayesian methods are also well-suited for working with complex data structures, such as networks or high-dimensional data.

Despite its advantages, Bayesian statistics also has some challenges, such as computational complexity and the choice of prior distributions. However, recent advances in computational techniques, such as Markov Chain Monte Carlo (MCMC) methods, have made it possible to perform Bayesian analysis on large and complex datasets. Bayesian statistics is considered to be a powerful and flexible approach to statistical analysis that allows us to incorporate prior knowledge, handle uncertainty, and update our beliefs as new data becomes available. With its ability to tackle complex problems and model structures, Bayesian statistics has become an increasingly popular tool for data scientists and statisticians alike.

Bayesian Inference

Bayesian inference is the process of updating our beliefs about unknown parameters or hypotheses based on observed data, within the framework of Bayesian statistics. It relies on Bayes' theorem, which allows us to combine our prior knowledge or beliefs with the likelihood of the observed data to compute the posterior probability, which represents our updated beliefs after observing the data. In other words, Bayesian inference is the method by which we draw conclusions about unknown quantities based on the observed data and our prior beliefs.

The given below is how Bayesian inference works in the context of Bayesian statistics:

Prior probability: We start by defining a prior probability distribution over the unknown parameters or hypotheses. The prior represents our initial beliefs or knowledge about the parameters before observing the data. This can be based on expert knowledge, historical data, or other sources of information. In some cases, if we have little or no prior information, we may choose a non-informative prior, which assigns equal probability to all possible values of the parameter.

Likelihood: The likelihood function represents the probability of observing the data given a specific value of the unknown parameter or hypothesis. It quantifies how consistent the data is with different parameter values. In Bayesian inference, we evaluate the likelihood of the data under different parameter values to determine how well each value explains the observed data.

Posterior probability: Using Bayes' theorem, we combine the prior probability and the likelihood to obtain the posterior probability distribution. The posterior distribution represents our updated beliefs about the unknown parameter or hypothesis after taking into account the observed data. The posterior distribution is the main result of Bayesian inference, and it allows us to make probabilistic statements about the unknown quantities.

Prediction and decision-making: Once we have the posterior distribution, we can use it to make predictions about future data or make decisions based on our updated beliefs. This can involve computing point estimates (e.g., the mean or median of the posterior distribution), credible intervals (intervals containing a specified probability mass of the posterior distribution), or other quantities of interest. We can also use the posterior distribution to perform model comparison or selection, as discussed in the previous section.

Bayesian inference has several advantages over classical frequentist inference, such as the ability to incorporate prior knowledge, handle uncertainty in a natural way, and update our beliefs in a sequential manner. However, it also comes with some challenges, such as the choice of prior distributions and the computational complexity of evaluating the posterior distribution, especially for high-dimensional or complex models. Despite these challenges, Bayesian inference has become an increasingly popular and powerful tool for statistical analysis in a wide range of fields, including data science, machine learning, and many others.

Putting Bayesian Inference into Action

Procedure to Perform Bayesian Inference

To carry out Bayesian inference in Rust, follow these six steps, which encompass the preparation, execution, and interpretation of the process:
Load and filter the dataset: Begin by loading your dataset into Rust and filtering it to retain only the data points that are relevant to your analysis. This initial step is crucial for ensuring that the subsequent steps are performed on accurate and representative data.
Choose a prior distribution: Based on your existing knowledge or beliefs, select an appropriate prior distribution for the unknown parameter(s) of interest. This distribution represents your initial assumptions about the parameters before incorporating the evidence from the data.
Define a likelihood function: Create a likelihood function that signifies the probability of observing the data given the unknown parameter(s). This function plays a critical role in connecting your data with your prior beliefs, allowing for the updating of your beliefs based on the evidence at hand.
Use an MCMC method: Implement a Markov Chain Monte Carlo (MCMC) technique, such as the Metropolis-Hastings algorithm, to generate samples from the posterior distribution. This process enables the exploration of the parameter space and the estimation of the distribution that combines your prior beliefs with the data evidence.

Compute relevant statistics: With the posterior samples obtained from the MCMC method, compute pertinent statistics such as the posterior mean, mode, and credible intervals. These statistics offer a comprehensive

understanding of the parameter estimates and the uncertainty surrounding them.
Interpret the results: Finally, analyze and interpret the computed statistics to draw meaningful conclusions about the unknown parameter(s) based on the Bayesian inference. This step may involve comparing different models, assessing the impact of your prior beliefs, or making predictions for future data points.

By adhering to these steps, you can effectively apply Bayesian inference in Rust to update your beliefs, quantify uncertainty, and make informed decisions based on the combination of prior knowledge and observed data.

Practical Illustration of Bayesian Inference

To demonstrate Bayesian inference practically on the given database, let's assume we want to estimate the average salary for a specific job title, say, Data Scientist. We can use Bayesian inference to update our prior beliefs about the average salary based on the observed salaries in the dataset.

First, let's load the dataset and filter it for Data Scientists:

```
// Load the dataset and filter it for Data Scientists

let data_scientist_salaries = dataset.select_rows_by_column("job_title", |x|
x == "Data Scientist");
```

Now, we need to choose a prior distribution for the average salary. Since we're dealing with a continuous positive-valued variable (salary), a common choice for the prior distribution is the log-normal distribution.

Let's assume a log-normal distribution with mean 0 and standard deviation 1 as our non-informative prior.

Next, we need to compute the likelihood. The likelihood represents the probability of observing the data given a specific value of the unknown parameter (in this case, the average salary). We can assume that the salaries are normally distributed around the true average salary with some standard deviation. We can estimate the standard deviation from the data, or we can incorporate it as another parameter in our Bayesian model.

To compute the posterior distribution, we need to multiply the prior distribution and the likelihood, and then normalize it to obtain a probability distribution. This can be done using Markov Chain Monte Carlo (MCMC) methods, such as the Metropolis-Hastings algorithm or the Gibbs sampler. For simplicity, let's use the Metropolis-Hastings algorithm.

```
// Imports

extern crate statrs;

use statrs::distribution::{LogNormal, Normal, Univariate};

use statrs::function::statistics::mean;

// Metropolis-Hastings algorithm

fn metropolis_hastings(
    likelihood: impl Fn(f64) -> f64,

    prior: impl Fn(f64) -> f64,

    initial_value: f64,

    n_iter: usize,

) -> Vec {

    let mut samples = Vec::new();

    let mut current_value = initial_value;

    let mut rng = rand::thread_rng();



    for _ in 0..n_iter {

        let proposal = Normal::new(current_value,
0.1).unwrap().sample(&mut rng);

        let acceptance_ratio =

            (likelihood(proposal) * prior(proposal)) /
(likelihood(current_value) * prior(current_value));

        if acceptance_ratio >= 1.0 || rand::random::() < acceptance_ratio {
            current_value = proposal;

        }

        samples.push(current_value);

    }

    samples

}

// Define the prior and likelihood functions

let prior = |mu: f64| LogNormal::new(0.0, 1.0).unwrap().pdf(mu);

let salaries = data_scientist_salaries.column("salaryinusd").unwrap();

let mean_salary = mean(&salaries);

let std_dev = 10_000.0; // You can estimate the standard deviation from
the data

let likelihood = |mu: f64| {

    let normal = Normal::new(mu, std_dev).unwrap();
```

```
   salaries.iter().map(|&x| normal.pdf(x)).product()

};

// Run the Metropolis-Hastings algorithm

let n_iter = 50_000;

let initial_value = mean_salary;



let samples = metropolis_hastings(likelihood, prior, initial_value, n_iter);

// Compute the posterior mean and credible interval

let posterior_mean = mean(&samples);

let credible_interval = statrs::function::statistics::quantiles(&samples, &
[0.025, 0.975]);

println!("Posterior mean: {}", posterior_mean

println!("95% credible interval: {:?}", credible_interval);
```

This code will compute the posterior mean and a 95% credible interval for the average salary of Data Scientists based on the given dataset and our prior assumptions. The Metropolis-Hastings algorithm generates samples from the posterior distribution, which we use to estimate the posterior mean and credible interval.

Note that the choice of prior, likelihood, and MCMC algorithm can have a significant impact on the results. In practice, you might want to experiment with different priors, likelihood functions, or use more advanced MCMC methods like Hamiltonian Monte Carlo or the No-U-Turn Sampler to improve the convergence and efficiency of your Bayesian inference.

Moreover, Bayesian inference can be extended to more complex models and questions, such as estimating the effect of different factors on salary, comparing the average salary across different job titles or experience levels, or predicting future salaries based on historical data. This flexibility and the ability to incorporate prior knowledge and uncertainty make Bayesian methods a powerful tool for data analysis in various domains.

Now that you have a good understanding of the Bayesian inference and how it can be applied to the given dataset, let's explore some other aspects of Bayesian statistics that can be useful in data analysis.

Bayesian Model Comparison

One powerful aspect of Bayesian statistics is model comparison. Bayesian model comparison allows you to compare multiple models and select the one that best explains the data. This can be particularly useful when trying to understand which factors influence a certain outcome or when selecting the best model for prediction. Bayesian model comparison is typically done using the Bayes factor or the Deviance Information Criterion (DIC). The Bayes factor compares the marginal likelihoods of two competing models, while the DIC compares the deviance of the models (a measure of model fit) with a penalty for model complexity.

Bayesian Hierarchical Modeling

Another important aspect of Bayesian statistics is hierarchical modeling. Hierarchical models allow you to model data with multiple levels of structure or grouping, such as employees within companies, students within schools, or patients within hospitals. These models can account for the dependencies and shared information between the different levels and can lead to more accurate estimates and predictions.

Bayesian hierarchical models can be fit using various MCMC methods, such as the Metropolis-Hastings algorithm, Gibbs sampling, or more advanced methods like the No-U-Turn Sampler. These methods can be implemented in Rust using the appropriate libraries and techniques discussed earlier in this chapter.

Lastly, Bayesian methods can also be applied to time series analysis and forecasting. Bayesian time series models, such as the Bayesian Structural Time Series model or Bayesian state-space models, can incorporate prior knowledge and uncertainty about the underlying process and allow for more accurate and robust predictions.

To implement these advanced Bayesian methods in Rust, you may need to extend the techniques and libraries discussed earlier in this chapter, or explore other specialized libraries tailored to specific Bayesian modeling tasks. As Bayesian statistics continue to gain popularity, more libraries and tools for Bayesian analysis in Rust are likely to emerge, making it even easier to perform advanced Bayesian modeling and inference.

Advanced Markov Chain Monte Carlo Method

Hamiltonian Monte Carlo (HMC) is an advanced Markov Chain Monte Carlo (MCMC) method that generates samples from a target distribution more efficiently compared to other methods like the Metropolis-Hastings algorithm or Gibbs sampling. HMC leverages the concepts of Hamiltonian mechanics and gradient information of the target distribution to propose samples that are less correlated and lead to faster convergence.

HMC simulates a physical system where a particle moves through the parameter space under the influence of a potential energy field and a kinetic energy term. The potential energy corresponds to the negative log probability of the target distribution, while the kinetic energy is associated with a momentum variable introduced for each parameter in the model. The method then alternates between sampling the momentum variables from a Gaussian distribution and simulating the particle's motion through the parameter space using Hamilton's equations, which are a set of differential equations that describe the system's dynamics. The advantage of HMC over other MCMC methods is that it can more effectively explore high-dimensional or complex distributions, as it uses gradient information to guide the exploration and avoids random walk behavior. This can lead to faster convergence, fewer correlated samples, and more accurate estimates of the posterior distribution.

Simple Implementation of HMC Method

let's demonstrate a sample implementation of HMC in Rust using the given dataset. Following are the general steps to implement HMC in Rust:

- Load and preprocess the dataset as described in previous section.
- Choose a prior distribution and define the likelihood function for your model.
- Implement the Hamiltonian Monte Carlo algorithm, which involves:

Defining a function to compute the gradient of the log posterior distribution (negative potential energy) with respect to the parameters.
Implementing the leapfrog integration method, which approximates the solution to Hamilton's equations and simulates the particle's motion through the parameter space.
Proposing new samples based on the simulated particle's trajectory and accepting or rejecting them according to the Metropolis-Hastings acceptance criterion.

- Run the HMC algorithm to generate samples from the posterior distribution.

Compute relevant statistics, such as the posterior mean and credible intervals, from the posterior samples.

The general idea of the above implementation is to leverage the gradient information to more efficiently explore the parameter space and generate samples from the posterior distribution. If you are interested in implementing HMC in Rust, you may want to refer to resources on the HMC algorithm and its implementation in other programming languages, such as Python or R, to gain more knowledge as these programming languages are heavily explored for statistics than Rust.

Model Comparison and Selection

Model comparison and selection play a pivotal role in data analysis, as they facilitate the identification of the most appropriate model for a given problem. Within the Bayesian framework, several methods can be employed to compare models, including the Bayes factor, the Deviance Information Criterion (DIC), and the widely applicable information criterion (WAIC).

The Bayes factor compares the marginal likelihoods of two competing models, M1 and M2. The marginal likelihood is the probability of the observed data under each model, after integrating out the unknown parameters. A larger Bayes factor indicates stronger evidence in favor of one model over the other. The Bayes factor can be difficult to compute, especially for complex models, as it requires integration over the parameter space.

The DIC is a model selection criterion that balances model fit with model complexity. It is based on the deviance, a measure of model fit, and includes a penalty term for the number of effective parameters in the model. Lower DIC values indicate better models. The DIC can be easily computed from the posterior samples generated by MCMC methods, making it a convenient choice for model comparison.

Similarly, the widely applicable information criterion (WAIC) estimates the predictive accuracy of a model while considering model complexity. Like DIC, lower WAIC values are preferable, as they signify a better trade-off between fit and complexity.

These model comparison methods enable analysts to make informed decisions about which Bayesian model to select, striking the right balance between explanatory power and model complexity. This careful selection process ultimately leads to more accurate and reliable predictions and inferences.

<u>Model Comparison using DIC</u>

The given below is a sample demonstration of model comparison using the Deviance Information Criterion (DIC) on the given dataset:

Suppose we have two competing models for predicting the salary of data science professionals:

Model 1: salary ~ experience_level + job_title
Model 2: salary ~ experience_level + job_title + company_size

First, fit both models using Bayesian regression and generate posterior samples using an MCMC method like the Metropolis-Hastings algorithm, Gibbs sampling, or HMC. For this demonstration, I'll assume you have already generated the posterior samples.

Next, compute the DIC for each model:

```
fn compute_dic(posterior_samples: &Array2, data: &Array2, model:
&dyn Fn(&Array2) -> f64) -> f64 {

    let deviance = -2.0 * posterior_samples.mapv(|params|
model(¶ms)).mean_axis(Axis(0)).unwrap();
```

```
    let p_d = 2.0 * (posterior_samples.mapv(|params| -2.0 *
model(¶ms)).var_axis(Axis(0), 0.0).unwrap());

    let dic = deviance + p_d;

   dic

}



let dic_model_1 = compute_dic(&posterior_samples_model_1, &data,
&model_1);

let dic_model_2 = compute_dic(&posterior_samples_model_2, &data,
&model_2);

println!("DIC Model 1: {:.2}, DIC Model 2: {:.2}", dic_model_1,
dic_model_2);
```

Compare the DIC values of the two models. The model with the lower DIC is preferred, as it balances model fit and complexity better. In the above sample program, if dic_model_1 is lower than dic_model_2, then Model 1 is considered the better model; otherwise, Model 2 is preferred.

Model Comparison using WAIC

Let's perform model comparison using the Widely Applicable Information Criterion (WAIC) on the given dataset. Continuing from the previous example of DIC, we have two competing models for predicting the salary of data science professionals:

Model 1: salary ~ experience_level + job_title
Model 2: salary ~ experience_level + job_title + company_size

First, fit both models using Bayesian regression and generate posterior samples using an MCMC method like the Metropolis-Hastings algorithm, Gibbs sampling, or HMC. For this demonstration, I'll assume you have already generated the posterior samples.

Next, compute the WAIC for each model:

```
fn compute_waic(posterior_samples: &Array2, data: &Array2, model:
&dyn Fn(&Array2) -> f64) -> f64 {

  let log_pointwise_predictive_density =
data.outer_iter().map(|observation| {



      let log_likelihood = posterior_samples.mapv(|params| model(¶ms));

      let log_predictive_density =
log_likelihood.exp().mean_axis(Axis(0)).unwrap().ln();

      log_predictive_density

  });
  let waic = -2.0 * log_pointwise_predictive_density.sum();

  waic

}

let waic_model_1 = compute_waic(&posterior_samples_model_1, &data,
&model_1);

let waic_model_2 = compute_waic(&posterior_samples_model_2, &data,
&model_2);

println!("WAIC Model 1: {:.2}, WAIC Model 2: {:.2}", waic_model_1,
waic_model_2);
```

Compare the WAIC values of the two models. The model with the lower WAIC is preferred, as it balances model fit and complexity better. In the above sample program, if waic_model_1 is lower than waic_model_2, then Model 1 is considered the better model; otherwise, Model 2 is preferred.

Summary

In this chapter, we focused on Bayesian Statistics, a powerful and flexible approach to statistical modeling and inference that offers a robust framework for dealing with uncertainty. The main advantage of Bayesian methods is their ability to incorporate prior knowledge into the analysis, which can improve the estimation of model parameters and predictions. Bayesian methods have gained popularity in recent years due to advances in computational techniques and their success in addressing complex statistical challenges.

We began by exploring the conceptual foundations of Bayesian statistics, including Bayes' theorem, which connects prior probabilities, likelihoods, and posterior probabilities. We then discussed the process of Bayesian inference, which involves updating our beliefs about the parameters of a model based on observed data. We delved into prior and likelihood, two key components of Bayesian inference, and learned about their roles in determining the posterior distribution. Next, we examined Markov Chain Monte Carlo (MCMC) methods, which are used to generate samples from complex posterior distributions that cannot be directly computed. We specifically discussed the Hamiltonian Monte Carlo (HMC) method, which is a more efficient MCMC algorithm that can handle high-dimensional and highly correlated parameter spaces. We also learned about model comparison and selection, essential steps in the data analysis process that help us choose the most suitable model for a given problem. We discussed various criteria for model comparison, such as the Bayes factor, the Deviance Information Criterion (DIC), and the Widely

Applicable Information Criterion (WAIC), which evaluate model fit while accounting for model complexity.

Throughout the chapter, we provided practical examples and demonstrations of Bayesian methods using Rust and the given dataset. We showcased how to apply these techniques to real-world problems, such as fitting regression models, generating random variables, and comparing models using DIC and WAIC.

Chapter 8: Multivariate Statistical Methods

Multivariate Statistical Methods

Introduction

Multivariate statistical methods involve the analysis of data with multiple variables, which allows us to understand the relationships and interactions among these variables. As opposed to univariate or bivariate methods, which focus on single or pairs of variables, multivariate techniques can provide a more comprehensive view of complex datasets. This can lead to better decision-making, improved predictions, and deeper insights into the underlying structures of the data. Multivariate statistical methods have gained significance in various fields, such as economics, biology, social sciences, and engineering, due to their ability to handle high-dimensional data and uncover hidden patterns.

Overview of Multivariate Techniques

Here, we will discuss some key multivariate techniques, their importance, and the breakthroughs they bring to statistical challenges:

Principal Component Analysis (PCA): PCA is a dimensionality reduction technique that transforms the original set of correlated variables into a new set of uncorrelated variables called principal components. These principal components capture most of the variance in the data while reducing the number of dimensions. PCA is widely used for data visualization, noise reduction, and preprocessing for other machine learning algorithms. It enables us to identify and interpret the underlying structure and patterns in high-dimensional data, which can be challenging to analyze directly.

Factor Analysis (FA): FA is another dimensionality reduction method that aims to identify the latent factors or underlying structures that explain the observed correlations among the variables. It is based on the assumption that the observed variables are linear combinations of the underlying factors, plus some error terms. FA is often used in social sciences and psychology to study unobservable constructs, such as intelligence, personality traits, or customer satisfaction.

Canonical Correlation Analysis (CCA): CCA is a technique for studying the relationship between two sets of variables. It finds linear combinations of variables from each set that are maximally correlated with each other. CCA can be used to identify shared patterns or trends between the two sets, which may not be evident when analyzing each set separately. Applications of CCA include studying the relationship between genetic and environmental factors, economic indicators, or brain imaging data and cognitive tasks.

Discriminant Analysis (DA): DA is a classification technique used to separate observations into predefined groups based on their characteristics. It creates linear decision boundaries that maximize the separation between the groups, taking into account the within-group and between-group variability. DA has been employed in various applications, such as medical diagnosis, credit scoring, or species identification.

Cluster Analysis: Cluster analysis is an unsupervised learning technique that seeks to partition observations into groups or clusters based on their similarity in the multivariate space. The goal is to create clusters that are homogeneous within themselves and heterogeneous between each other. Cluster analysis can reveal natural groupings or structures within the data, which can inform decision-making or further analysis. It has been applied in fields like market segmentation, image analysis, and gene expression studies.

Multivariate Regression: Multivariate regression extends the traditional regression framework to include multiple dependent variables. This allows us to model the relationships between multiple predictors and multiple outcomes simultaneously, while accounting for the correlations among the dependent variables. Applications of multivariate regression include modeling the impact of multiple factors on multiple outcomes, such as the influence of environmental factors on the health of multiple species or the effects of marketing strategies on the sales of multiple products.

To summarize, multivariate statistical methods offer powerful tools for understanding and interpreting complex, high-dimensional data. By considering multiple variables simultaneously, these techniques can reveal deeper insights and provide more accurate predictions than univariate or bivariate methods. As data science professionals, having a solid grasp of multivariate techniques is crucial for tackling the increasingly complex datasets and statistical challenges that arise in modern research and practice.

Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a prevalent multivariate method employed for dimensionality reduction in various fields, including machine learning, data analysis, and visualization. The primary objective of PCA is to generate a new set of uncorrelated variables, referred to as principal components. These components encapsulate most of the variance present in the original dataset while simultaneously reducing the overall number of dimensions, which helps mitigate the curse of dimensionality and facilitate more efficient analysis.

The mathematical underpinnings of PCA are rooted in linear algebra and eigenvalue decomposition. PCA starts by calculating the covariance matrix of the dataset, which quantifies the degree of correlation between the different features. Subsequently, it performs eigenvalue decomposition on this matrix to identify its eigenvectors and eigenvalues. These eigenvectors represent the principal components of the data, while their corresponding eigenvalues indicate the amount of variance captured by each component. The components are then sorted in descending order based on their eigenvalues, ensuring that the first principal component accounts for the largest proportion of variance, the second component captures the next largest proportion, and so on. By selecting a subset of principal components, we can effectively reduce the dimensionality of the dataset while preserving most of its inherent structure and information.

Procedure of PCA

Following is a step-by-step walkthrough to the mathematical architecture of PCA:

Standardize the dataset: Since PCA is sensitive to the scale of the variables, it's essential to standardize the dataset, so each variable has a mean of 0 and a standard deviation of 1.
Calculate the covariance matrix: This step involves calculating the covariance matrix of the standardized dataset. The covariance matrix quantifies the linear relationship between each pair of variables in the dataset.
Compute the eigenvalues and eigenvectors of the covariance matrix: Eigenvalues and eigenvectors are crucial in determining the principal components. Eigenvectors represent the direction of the principal components, while eigenvalues represent the magnitude of the variance captured by each component.
Sort the eigenvalues in descending order and select the top k eigenvectors corresponding to the k largest eigenvalues: The number of selected eigenvectors (k) determines the number of principal components retained in the analysis. These eigenvectors form a matrix called the loading matrix.
Project the original dataset onto the loading matrix: Multiply the standardized dataset by the loading matrix to obtain the new set of principal components.

<u>Sample Program to Implement PCA</u>

Now, let's implement PCA on the given dataset using the ndarray and ndarray-linalg crates:

```
use ndarray::prelude::*;
use ndarray_linalg::Eigh;

use ndarray_rand::{RandomExt, F32};

use ndarray_stats::QuantileExt;



use statrs::function::erf::erf;

fn main() {

  // Load the dataset

   let data = read_data_from_csv(); // Replace this with the function to
read your data from the CSV

  // Standardize the dataset

  let standardized_data = standardize_data(&data);

  // Calculate the covariance matrix

   let covariance_matrix = standardized_data.t().dot(&standardized_data) /
(standardized_data.nrows() - 1) as f64;

   // Compute the eigenvalues and eigenvectors

  let Eigh(vals, vecs) = covariance_matrix.eigh(UPLO::Upper).unwrap();
```

```
    // Sort the eigenvalues in descending order and select the top k
eigenvectors

    let k = 2; // Number of principal components

   let sorted_indices = vals.argsort_rev();

   let top_k_indices = &sorted_indices.slice(s![0..k]);

   let loading_matrix = vecs.select(Axis(1), top_k_indices);

    // Project the standardized data onto the loading matrix

   let principal_components = standardized_data.dot(&loading_matrix);

   // Output the principal components

   println!("Principal Components:\n{:?}", principal_components);

}
```

Replace the read_data_from_csv function with the appropriate function to read your dataset.

After running the code, you will see the output displaying the first two principal components for each observation in the dataset.

Canonical Correlation Analysis (CCA)

Canonical Correlation Analysis (CCA) is a sophisticated statistical technique employed to investigate the associations between two sets of multivariate variables. Its primary objective is to identify linear combinations of variables from both sets that exhibit the highest degree of correlation. In doing so, CCA helps uncover underlying patterns and relationships between the variables, which can then be utilized for further analysis or interpretation. CCA operates by computing the canonical variates, which are linear combinations of variables from each set. The coefficients of these linear combinations, known as canonical weights, are chosen to maximize the correlation between the canonical variates. Through this process, CCA yields multiple pairs of canonical variates and their corresponding canonical correlations, which indicate the strength of the relationships between the variables.

This technique is particularly useful in situations where researchers aim to explore the complex relationships between two multivariate data sets, such as psychological, socioeconomic, or environmental factors. Additionally, CCA can be applied to reduce the dimensionality of the data, making it more manageable and easier to analyze. However, it is important to note that CCA assumes linearity in the relationships between variables and requires a large sample size to ensure the stability of the estimated correlations. Furthermore, the interpretation of the canonical variates can sometimes be challenging, as they may not have a direct or clear meaning in the context of the original data. Despite these limitations, CCA remains a valuable technique for researchers seeking to uncover meaningful connections between two sets of multivariate variables.

Procedure to Perform CCA

Below is an overview of the CCA algorithm:
Standardize the datasets: Similar to PCA, CCA also requires the data to be standardized, with each variable having a mean of 0 and a standard deviation of 1.
Calculate the cross-covariance matrix: Compute the cross-covariance matrix between the two sets of variables. This matrix captures the relationships between the variables in both sets.
Compute the eigenvalues and eigenvectors of the cross-covariance matrix: Similar to PCA, the eigenvalues and eigenvectors are essential for determining the canonical correlations and weights.
Sort the eigenvalues in descending order and select the top k eigenvectors corresponding to the k largest eigenvalues: These eigenvectors form the canonical weights for each set of variables.
Calculate the canonical correlations and canonical variates: Use the canonical weights to compute the canonical correlations (the correlation between the canonical variates) and the canonical variates (the linear combinations of variables).

Sample Program to Implement CCA

Now, let's implement CCA on the given dataset using the ndarray and ndarray-linalg crates:

```
use ndarray::prelude::*;
use ndarray_linalg::SVD;

use ndarray_rand::{RandomExt, F32};



use ndarray_stats::QuantileExt;

use statrs::function::erf::erf;

fn main() {

  // Load the dataset

   let (data1, data2) = read_data_from_csv(); // Replace this with the
function to read your data from the CSV and split it into two sets of
variables

  // Standardize the datasets

  let standardized_data1 = standardize_data(&data1);

  let standardized_data2 = standardize_data(&data2);

  // Calculate the cross-covariance matrix

   let cross_cov_matrix =
standardized_data1.t().dot(&standardized_data2) /
(standardized_data1.nrows() - 1) as f64;
```

```
    // Compute the singular value decomposition (SVD) of the cross-
covariance matrix

    let SVD { u, s, vt } = cross_cov_matrix.svd(true, true).unwrap();

    // Canonical correlations are the singular values of the cross-covariance
matrix

   let canonical_correlations = s;

    // Canonical weights are the left and right singular vectors

   let canonical_weights1 = u.unwrap();

   let canonical_weights2 = vt.unwrap().t();

   // Calculate the canonical variates

   let canonical_variates1 = standardized_data1.dot(&canonical_weights1);

   let canonical_variates2 = standardized_data2.dot(&canonical_weights2);



    // Output the canonical correlations and variates

   println!("Canonical Correlations:\n{:?}", canonical_correlations);
```

```
    println!("Canonical Variates 1:\n{:?}", canonical_variates1);

    println!("Canonical Variates 2:\n{:?}", canonical_variates2);

}
```

Revise the read_data_from_csv function with the suitable function to load your dataset and partition it into two variable sets. Once the code executes, observe the output, which exhibits the canonical correlations and corresponding canonical variates for both sets of variables.

Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis (LDA) is a widely-used technique in the realms of classification and dimensionality reduction. Its primary goal is to identify the most effective linear combinations of features that can optimally distinguish between two or more classes. This is accomplished by maximizing the distance between class means and minimizing the within-class scatter. LDA operates under the assumption that the data is normally distributed and that the classes share identical covariance matrices. These assumptions help simplify the computation process and provide a linear boundary for separating the classes.

Despite its linear nature, LDA can be remarkably effective in various scenarios. It excels in cases where the data adheres to the underlying assumptions, leading to improved classification performance. Additionally, the dimensionality reduction aspect of LDA can assist in mitigating the curse of dimensionality and reduce the risk of overfitting, ultimately enhancing the model's generalization capabilities. However, LDA's reliance on these assumptions can also be a limitation when dealing with data that does not follow a normal distribution or exhibits dissimilar covariance matrices across classes. In such cases, alternative methods, such as Quadratic Discriminant Analysis (QDA) or non-linear classifiers, might be more suitable.

Procedure to Perform LDA Algorithm

Below is an overview of the LDA algorithm:

● Compute the mean of each class: Calculate the mean vector for each class.

Calculate the within-class scatter matrix: Compute the scatter matrix for each class and then sum them up to obtain the within-class scatter matrix.
● Calculate the between-class scatter matrix: Compute the scatter matrix between the classes.
Compute the eigenvalues and eigenvectors of the matrix product of the inverse of the within-class scatter matrix and the between-class scatter matrix.
Sort the eigenvalues in descending order and select the top k eigenvectors corresponding to the k largest eigenvalues: These eigenvectors form the LDA transformation matrix.
Apply the LDA transformation matrix to the original dataset to obtain the transformed dataset.

Sample Program to Implement LDA

Now, let's implement LDA on the given dataset using the ndarray, ndarray-linalg, and statrs crates:

```
use ndarray::prelude::*;

use ndarray_linalg::Inverse;

use statrs::function::erf::erf;

fn main() {

   // Load the dataset and labels
```

```
    let (data, labels) = read_data_and_labels_from_csv(); // Replace this
with the function to read your data and labels from the CSV

    // Compute the mean of each class

   let means = compute_class_means(&data, &labels);

    // Calculate the within-class scatter matrix

    let within_class_scatter_matrix =
compute_within_class_scatter_matrix(&data, &labels, &means);

    // Calculate the between-class scatter matrix



    let between_class_scatter_matrix =
compute_between_class_scatter_matrix(&data, &labels, &means);

    // Compute the eigenvalues and eigenvectors of the matrix product of
the inverse of the within-class scatter matrix and the between-class scatter
matrix

   let matrix_product =
within_class_scatter_matrix.inv().unwrap().dot(&between_class_scatter_
matrix);

   let (eigenvalues, eigenvectors) =
matrix_product.eigh(UPLO::Upper).unwrap();
```

```
   // Sort the eigenvalues in descending order and select the top k
eigenvectors corresponding to the k largest eigenvalues

   let k = 2; // Choose the number of dimensions for the reduced dataset

  let sorted_indices =
eigenvalues.argsort(ndarray::SortOrder::Descending);

  let lda_transformation_matrix = eigenvectors.select(Axis(1),
&sorted_indices.slice(..k));

   // Apply the LDA transformation matrix to the original dataset to obtain
the transformed dataset

  let transformed_data = data.dot(&lda_transformation_matrix);

  // Output the transformed dataset

  println!("Transformed Data:\n{:?}", transformed_data);

}
```

As previously mentioned, replace the read_data_and_labels_from_csv function with a suitable function to read your dataset and labels from the CSV file. Once the code is executed, the output will display the transformed dataset, showcasing the effects of applying LDA on the data.

Independent Component Analysis (ICA)

Independent Component Analysis (ICA) is a prominent multivariate statistical technique primarily employed for blind source separation or feature extraction. The central aim of ICA is to disentangle a set of mixed signals into their original sources, with the assumption that these sources are non-Gaussian and statistically independent from one another. ICA's underlying principle is based on the idea that the observed data is a linear mixture of independent components. The technique strives to recover the original sources by estimating a demixing matrix, which, when applied to the observed data, yields the independent components.

One key distinction between ICA and other methods such as Principal Component Analysis (PCA) lies in their objectives. PCA concentrates on minimizing the variance and discovering orthogonal components, whereas ICA focuses on identifying statistically independent components. This difference in focus makes ICA particularly useful for applications where the sources of interest exhibit statistical independence. ICA has found widespread adoption in diverse fields, such as image processing, where it can be employed to remove noise or artifacts; audio processing, where it can separate mixed audio signals into distinct sources, such as individual speakers or instruments; and financial data analysis, where it helps unveil hidden patterns in financial time series data or detect fraudulent activities.

Moreover, ICA has proven valuable in the field of neuroscience, particularly for analyzing electroencephalogram (EEG) and functional magnetic resonance imaging (fMRI) data, as it helps isolate distinct neural sources from the complex, mixed signals produced by the brain's electrical activity. Despite its many advantages, ICA does have some limitations, including its sensitivity to the initialization of the demixing matrix and its dependence on the assumption of non-Gaussian sources. Nevertheless, when applied appropriately, ICA serves as a powerful technique for uncovering hidden patterns and sources in various types of data.

Overview of ICA Algorithm

Below is a high-level overview of the ICA algorithm:

- Center the data by subtracting the mean from each feature.

Whiten the data: Apply PCA or another whitening method to decorrelate the features and make their variances equal to 1.

- Initialize a random square mixing matrix.

Optimize the mixing matrix using an iterative algorithm such as FastICA, Infomax, or JADE until convergence.

Sample Program to Implement ICA

Now, let's implement ICA using the ndarray and ndarray-rand crates along with the FastICA algorithm:

```
use ndarray::{Array2, Axis};

use ndarray_rand::RandomExt;

use ndarray_rand::rand_distr::StandardNormal;

fn main() {
   // Load the dataset

    let data = read_data_from_csv(); // Replace this with the function to
read your data from the CSV

   // Center the data



   let mean = data.mean_axis(Axis(0)).unwrap();

    let centered_data = &data - &mean;

   // Whiten the data

    let whitened_data = whiten_data(¢ered_data); // Replace this with a
function to whiten your data

    // Initialize a random square mixing matrix

   let n_components = data.ncols();

    let mut mixing_matrix: Array2 = Array2::random((n_components,
n_components), StandardNormal);

    // Optimize the mixing matrix using the FastICA algorithm

   let max_iter = 1000;
```

```
    let tolerance = 1e-4;

    let (unmixing_matrix, _) = fast_ica(&whitened_data, &mut
mixing_matrix, max_iter, tolerance); // Replace this with a function to
apply FastICA

    // Separate the original sources

    let separated_sources = whitened_data.dot(&unmixing_matrix.t());

    // Output the separated sources

    println!("Separated Sources:\n{:?}", separated_sources);

}
```

Once you execute the code, the output will showcase the separated sources. ICA proves to be a potent tool when non-Gaussianity and statistical independence assumptions are valid. Nevertheless, it is crucial to be aware of these assumptions and constraints while employing ICA or other multivariate statistical techniques on real-world datasets to ensure accurate and reliable results.

Multidimensional Scaling (MDS)

Multidimensional Scaling (MDS) is a versatile multivariate statistical technique that aims to visualize the similarity or dissimilarity between objects in a high-dimensional space by projecting them onto a lower-dimensional space, typically 2D or 3D. This technique is applicable across various fields, such as psychology, marketing, and network analysis, offering valuable insights into complex data.

The primary goal of MDS is to maintain the pairwise distances between objects as accurately as possible when mapping them to a lower-dimensional space.

Types of Multidimensional Scaling

This technique can be categorized into two main types: Classical MDS and Non-metric MDS.

Classical MDS (also known as Principal Coordinates Analysis): This type of MDS assumes that the input data comes in the form of a distance matrix. It employs eigendecomposition of the matrix to compute the lower-dimensional representation. Classical MDS works best when the data has a linear structure, and the actual distances between objects are crucial to the analysis.

Non-metric MDS: In contrast to Classical MDS, Non-metric MDS focuses on preserving the rank order of the distances between objects rather than the actual distances. This method is more suitable for data with non-linear relationships and is less sensitive to measurement scale differences. Non-metric MDS uses iterative optimization techniques, such as the stress majorization algorithm or the Shepard-Kruskal algorithm, to find the best-fitting configuration.

MDS is a valuable tool for analyzing complex data sets and revealing hidden structures within the data. The technique's ability to reduce dimensionality while preserving critical relationships between objects makes it an essential asset in various research areas. Some common applications include the study of consumer preferences in marketing research, social network analysis, the examination of similarities in psychological traits, and the comparison of biological species in ecology.

Sample Program to Implement Classical MDS

Let's implement Classical MDS using ndarray and ndarray-linalg crates for the given database:

```
use ndarray::{Array2, Axis};

use ndarray_linalg::Eigh;

fn main() {

  // Load the dataset
```

```
    let data = read_data_from_csv(); // Replace this with the function to
read your data from the CSV

   // Calculate the distance matrix

    let distance_matrix = calculate_distance_matrix(&data); // Replace this
with a function to calculate the distance matrix

   // Apply Classical MDS

    let (mds_coordinates, _) = classical_mds(&distance_matrix, 2); //
Replace this with a function to apply Classical MDS

   // Output the MDS coordinates

   println!("MDS Coordinates:\n{:?}", mds_coordinates);

}
```

Replace the read_data_from_csv, calculate_distance_matrix, and classical_mds functions with appropriate functions to read your dataset from the CSV, calculate the distance matrix, and apply Classical MDS, respectively. After running the code, you will see the output displaying the MDS coordinates in 2D.

MDS is useful for visualizing and understanding the structure and relationships between objects in high-dimensional data. By reducing the dimensionality, it helps users to spot patterns, clusters, and trends in the data. However, it is essential to be aware that the lower-dimensional representation might not capture all the information present in the original high-dimensional space, and some information may be lost during the dimensionality reduction process.

Summary

This chapter focuses on Multivariate Statistical Methods, which are essential for analyzing datasets containing multiple variables simultaneously. These methods aim to explore the relationships between variables, reduce the dimensionality of the data, and uncover hidden patterns or structures. The importance of multivariate statistical methods lies in their ability to handle complex datasets and provide insights that may not be apparent when analyzing each variable separately.

The chapter begins with PCA (Principal Component Analysis), a popular technique for dimensionality reduction. PCA identifies orthogonal axes (principal components) that explain the maximum variance in the data. This method is suitable for linearly correlated data and can help reduce noise and redundancy. It then demonstrates how to implement PCA on the given dataset using Rust libraries. Next, the chapter introduces ICA (Independent Component Analysis), designed to separate mixed signals into statistically independent sources. ICA is useful for blind source separation and feature extraction, particularly when the sources are non-Gaussian and statistically independent. The implementation of ICA on the given dataset is demonstrated using Rust. The chapter then proceeds to MDS (Multidimensional Scaling), which is used to visualize the similarity or dissimilarity between objects in a high-dimensional space by mapping them to a lower-dimensional space. MDS is helpful for understanding the structure and relationships between objects in high-dimensional data. A sample implementation of MDS using Rust is provided for the given dataset.

To sum it up, Chapter 8 provides a comprehensive overview of various multivariate statistical methods, including PCA, ICA, and MDS. It highlights their importance, key concepts, and applicability to different datasets and goals. The chapter also demonstrates how to implement these methods using Rust libraries on a sample dataset, emphasizing the practical aspect of applying these techniques in data analysis. Ultimately, the choice of the best method depends on the specific problem and dataset characteristics, and it is often beneficial to try multiple techniques and compare their results to gain a deeper understanding of the underlying data structure and relationships.

Chapter 9: Nonlinear Models and Machine Learning

Nonlinear Models

Nonlinear models are crucial for addressing complex data patterns that cannot be accurately represented by linear relationships. These models provide superior flexibility and adaptability in tackling real-world problems, where the connections between variables are frequently complex and nonlinear. They have proven to be invaluable tools in diverse domains such as finance, healthcare, and natural language processing, among others.

Some popular nonlinear models include:
Decision Trees: These models recursively divide the data into subsets based on input features, forming a tree-like structure. Decision trees can be utilized for both regression and classification tasks. They are straightforward, easy to interpret, and robust to outliers. Moreover, they can handle both continuous and categorical variables.
Support Vector Machines (SVM): SVM is a versatile algorithm employed for classification and regression tasks. It seeks to find the optimal separating hyperplane (in the case of classification) or the best fitting hyperplane (in the case of regression) by maximizing the margin between classes or minimizing the regression error, respectively. SVMs can tackle linear and nonlinear problems using kernel functions, which map the data into a higher-dimensional space, thereby enabling the discovery of nonlinear relationships.

Neural Networks: These are biologically-inspired models composed of interconnected nodes (neurons) organized into layers. Neural networks can approximate any continuous function and are highly effective in learning complex patterns in large datasets. Deep learning, a subfield of

machine learning, emphasizes neural networks with numerous hidden layers, facilitating the identification of intricate features in data. Applications of neural networks include image recognition, natural language processing, and game playing.
Ensemble Methods: Ensemble methods combine multiple models to enhance predictive performance. Techniques such as bagging (Bootstrap Aggregating), boosting, and stacking are employed to reduce overfitting, increase accuracy, and make the model more robust. Popular ensemble methods encompass Random Forests, Gradient Boosting Machines (GBM), and XGBoost. These methods often outperform single-model approaches and are widely used in various machine learning competitions.

The progression of statistics into machine learning (ML) has been transformative, with ML algorithms furnishing potent tools for resolving intricate problems. Some key breakthroughs encompass:
Handling Large Datasets: ML algorithms can effectively process and learn from vast amounts of data, facilitating the discovery of sophisticated patterns and relationships that would be challenging to uncover using conventional statistical methods.

Feature Learning: ML techniques such as deep learning can automatically learn and extract pertinent features from raw data, eliminating the necessity for manual feature engineering. This capability significantly reduces the time and effort required for data preprocessing and allows for more accurate modeling.
Robustness and Adaptability: ML models can adapt to alterations in data distributions and are frequently more robust to noise and outliers compared to traditional statistical models. This makes them suitable for handling real-world data, which often exhibits such characteristics.
Model Complexity: ML models can capture intricate, nonlinear relationships in data, permitting them to model complex phenomena with greater accuracy. This enables the development of models that can address

a wide range of problems, from simple linear regression to highly complex deep learning tasks.
Generalization: ML algorithms can generalize well to unseen data, rendering them appropriate for predictive modeling and decision-making across various domains. This property is particularly important in situations where models must adapt to new, previously unseen data.
Automation and Scalability: ML algorithms can be effortlessly automated and scaled to handle extensive datasets, rendering them indispensable in data-driven industries. This allows organizations to process and analyze large volumes of data quickly and efficiently, enabling data-driven decision-making and insights.

Nonlinear models have become essential tools for handling complex data patterns that cannot be accurately captured by linear relationships. They offer greater flexibility and adaptability in addressing real-world problems, where relationships between variables are often intricate and nonlinear. With the ongoing advancement. In further topics, we will examine and implement each of these models and will explore how the linear relationships are well correlated and captured.

Decision Trees

Overview

A decision tree is a versatile, flowchart-like structure used in machine learning and data mining for making decisions or predictions based on a set of input features. This tree-like structure consists of three primary components: internal nodes, branches, and leaf nodes. Internal nodes represent the decision points based on the values of the input features. Branches, on the other hand, signify the possible outcomes or consequences of those decisions. Lastly, leaf nodes symbolize the final decision or prediction made by the tree.

The construction of decision trees involves a recursive algorithm that iteratively selects the most suitable feature to split the dataset at each internal node. This selection process is guided by a specific criterion, such as Gini impurity or Information Gain. These criteria help determine the homogeneity of the resulting subsets, with the ultimate goal of maximizing the purity or information content of the splits. Decision trees offer several advantages, including their ability to handle both numerical and categorical data, their ease of interpretation, and their inherent feature selection capability. However, they are also prone to overfitting, which can be mitigated through techniques like pruning or by using ensemble methods like Random Forests or Gradient Boosting Machines.

Building Decision Tree

The given below is a step-by-step illustration of building a decision tree for the given dataset using Rust:

Import the required libraries:

```
use ndarray::prelude::*;



use ndarray_csv::Array2Reader;

use std::fs::File;

use std::io::BufReader;

use std::error::Error;

use decision_tree::prelude::*;
```

Load the dataset and preprocess it as required:

```
fn load_dataset() -> Result, BoxError>> {

  let file = File::open("ds_salaries.csv")?;

  let reader = BufReader::new(file);

  let dataset = Array2::from_reader(reader)?;
```

```
    // Preprocess the dataset as needed

   Ok(dataset)

}
```

Split the dataset into training and testing sets:

```
fn train_test_split(dataset: &Array2, test_size: f64) -> (Array2, Array2,
Array2, Array2) {

    // Implement the train-test split logic here

}
```

Create the decision tree model:

```
fn create_decision_tree() -> DecisionTreef64> {

   let mut tree = DecisionTree::new();

   tree.max_depth = Some(3);

   tree.min_samples_split = 2;

   tree

}
```

Train the decision tree model:

```
fn main() -> Result<(), BoxError>> {

  let dataset = load_dataset()?;

   let (x_train, x_test, y_train, y_test) = train_test_split(&dataset, 0.2);



  let mut tree = create_decision_tree();

  tree.fit(&x_train, &y_train)?;

   // Evaluate the model on the test set

}
```

Evaluate the model:

```
fn evaluate(tree: &DecisionTreef64>, x_test: &Array2, y_test: &Array2) {

  let predictions = tree.predict(&x_test).unwrap();

   // Compute evaluation metrics like accuracy, precision, recall, etc.

}
```

Visualize the decision tree (optional):
Visualizing the tree might not be straightforward in Rust. However, you can export the tree to a format like JSON and use external libraries like Graphviz in Python to visualize the tree.

This above practical program demonstrates how to build a decision tree model for the given dataset using Rust. Please note that you may need to preprocess the dataset, convert categorical variables into numerical values, and handle missing values before training the model.

Support Vector Machines (SVM)

Overview

We will now delve into another prominent non-linear model known as Support Vector Machines (SVM). As versatile classifiers, SVMs can adeptly manage data that is either linearly or nonlinearly separable. They achieve this by determining the most suitable separating hyperplane or boundary that distinguishes between different classes.

At the heart of SVMs is the ingenious "kernel trick," which allows data to be projected into a higher-dimensional space. This transformation renders the previously non-linearly separable data linearly separable, making it possible for SVMs to find the optimal hyperplane. The kernel trick uses a wide variety of kernel functions, such as linear, polynomial, radial basis function (RBF), and sigmoid, to cater to different data distributions and patterns. SVMs are known for their strong generalization capabilities and robustness, which make them suitable for a diverse range of classification tasks. They excel at handling high-dimensional data and are less prone to overfitting compared to some other machine learning models. However, they may require more computational resources and careful hyperparameter tuning to achieve the best results. With these qualities, SVMs have earned a reputation as a powerful and flexible tool in the field of machine learning.

Building SVM Model

The given below is a step-by-step illustration of building a Support Vector Machine for the given dataset using Rust:

Import the required libraries:

```
use ndarray::prelude::*;


use ndarray_csv::Array2Reader;

use std::fs::File;

use std::io::BufReader;

use std::error::Error;

use rusty_machine::learning::svm::{SVM, SvmParameter};

use rusty_machine::learning::toolkit::kernel;

use rusty_machine::learning::SupModel;
```

Load the dataset and preprocess it as required (same as the previous example):

```
fn load_dataset() -> Result, BoxError>> {

  // ...
}

fn train_test_split(dataset: &Array2, test_size: f64) -> (Array2, Array2,
Array2, Array2) {

  // ...

}
```

Create the SVM model:

```
fn create_svm() -> SVMkernel::RBF> {

  let svm_params = SvmParameter::default()

      .with_kernel(kernel::RBF::new(0.5)) // Change the kernel and its
parameters as needed

      .with_c(1.0)

      .with_shrinking(true);

  SVM::new(svm_params)

}
```

Train the SVM model:

```
fn main() -> Result<(), BoxError>> {

  let dataset = load_dataset()?;


   let (x_train, x_test, y_train, y_test) = train_test_split(&dataset, 0.2);

  let mut svm = create_svm();

  svm.train(&x_train, &y_train)?;

   // Evaluate the model on the test set

}
```

Evaluate the model:

```
fn evaluate(svm: &SVMkernel::RBF>, x_test: &Array2, y_test: &Array2)
{

  let predictions = svm.predict(&x_test).unwrap();

   // Compute evaluation metrics like accuracy, precision, recall, etc.

}
```

This example demonstrates how to build an SVM model for the given dataset using Rust. Please note that you may need to preprocess the dataset, convert categorical variables into numerical values, and handle missing values before training the model.

Neural Networks

Fundamentals of Neural Networks

Neural Networks represent a formidable category of machine learning models, especially adept at tackling intricate, high-dimensional data. Inspired by the human brain's neural structure, these networks consist of multiple layers of interconnected neurons, which are systematically organized into three primary groups: input, hidden, and output layers. The input layer is responsible for receiving raw data, while the output layer produces the final predictions or classifications. Between these two layers are the hidden layers, which perform complex transformations and feature extraction to enable the model to learn intricate patterns within the data. The neurons in each layer communicate through weighted connections, allowing information to flow from one layer to another.

One of the key strengths of Neural Networks is their ability to learn hierarchical representations, which means they can automatically identify and extract relevant features from the data without manual intervention. This makes them particularly effective in tasks such as image recognition, natural language processing, and speech recognition, where traditional algorithms struggle to cope with the complexity and scale of the data. Moreover, Neural Networks can be scaled up by adding more layers and neurons to increase their capacity for learning complex relationships, resulting in deep learning models known as Deep Neural Networks (DNNs). While their impressive capabilities come at the cost of increased computational demands and training time, the use of modern hardware accelerators like GPUs and dedicated AI chips has significantly improved their efficiency.

Building Neural Network Model

The given below is a step-by-step illustration of building a simple Neural Network for the given dataset using Rust:

Import the required libraries:

```
use ndarray::prelude::*;

use ndarray_csv::Array2Reader;

use std::fs::File;

use std::io::BufReader;

use std::error::Error;

use tch::{nn, nn::Module, nn::OptimizerConfig, Device, Tensor};
```

Load the dataset and preprocess it as required (same as the previous example):

```
fn load_dataset() -> Result, BoxError>> {

  // ...
}

fn train_test_split(dataset: &Array2, test_size: f64) -> (Array2, Array2,
Array2, Array2) {

  // ...

}
```

Define the neural network structure:

```
fn create_net(vs: &nn::Path) -> impl nn::Module {

  nn::seq()

      .add(nn::linear(vs, 11, 64, Default::default())) // 11 input features

      .add_fn(|xs| xs.relu())

      .add(nn::linear(vs, 64, 64, Default::default()))

      .add_fn(|xs| xs.relu())


      .add(nn::linear(vs, 64, 1, Default::default())) // 1 output value

}
```

Train the neural network:

```
fn main() -> Result<(), BoxError>> {

  let dataset = load_dataset()?;

   let (x_train, x_test, y_train, y_test) = train_test_split(&dataset, 0.2);

  let device = Device::cuda_if_available();

  let vs = nn::VarStore::new(device);

  let net = create_net(&vs.root());

   let mut opt = nn::Adam::default().build(&vs, 1e-3)?;

  for epoch in 1..=500 {

      let loss = net

          .forward(&Tensor::from_ndarray(&x_train).to_device(device))

          .mse_loss(&Tensor::from_ndarray(&y_train).to_device(device),
tch::Reduction::Mean);

      opt.backward_step(&loss);
```

```
        println!("Epoch: {}, Loss: {:?}", epoch, f64::from(loss));

    }

    // Evaluate the model on the test set

}
```

Evaluate the model:

```
fn evaluate(net: &impl nn::Module, x_test: &Array2, y_test: &Array2) {

    let predictions =
net.forward(&Tensor::from_ndarray(&x_test)).to_ndarray().unwrap();



    // Compute evaluation metrics like Mean Squared Error, R2 Score, etc.

}
```

This example demonstrates how to build a simple Neural Network for the given dataset using Rust and the tch-rs library. Please note that you may need to preprocess the dataset, convert categorical variables into numerical values, and handle missing values before training the model.

Ensemble Methods

Overview

Ensemble methods represent a category of machine learning techniques designed to improve accuracy and generalization by aggregating the predictions of multiple base models. This approach capitalizes on the strengths of individual models while mitigating their weaknesses, ultimately resulting in a more robust and accurate composite model.

There are several popular ensemble methods, including Bagging, Boosting, and Stacking.
Bagging, or Bootstrap Aggregating, is a technique where multiple base models are trained independently using random subsets of the training data, sampled with replacement. The final prediction is obtained by averaging the predictions (for regression tasks) or by taking a majority vote (for classification tasks) across all base models. Bagging helps reduce overfitting and variance while improving the overall stability of the model.
Boosting, on the other hand, focuses on iteratively training base models to correct the errors of their predecessors. Each subsequent model places more emphasis on instances that were misclassified by the previous model. The final prediction is derived by weighting the predictions of all base models and then combining them. This method often leads to higher accuracy compared to bagging but may be more prone to overfitting.

Stacking, also known as Stacked Generalization, is a technique that involves training multiple base models using different algorithms or configurations, and then training a higher-level meta-model to combine their predictions. The meta-model learns to optimally weigh and blend the predictions from the base models, leading to better overall performance.

These ensemble methods are widely employed in machine learning due to their ability to improve prediction accuracy, generalization, and robustness. By leveraging the strengths of multiple base models and utilizing various strategies for combining their predictions, ensemble techniques can effectively address complex problems and deliver superior performance across a wide range of applications.

Building Bagging Ensemble of Decision Tree

The given below is a step-by-step illustration of building a simple Bagging ensemble of Decision Trees for the given dataset using Rust:

Import the required libraries:

```
use ndarray::prelude::*;

use ndarray_csv::Array2Reader;

use std::fs::File;

use std::io::BufReader;

use std::error::Error;

use randomforest::RandomForestRegressor;
```

Load the dataset and preprocess it as required (same as the previous examples):

```
fn load_dataset() -> Result, BoxError>> {

   // ...

}

fn train_test_split(dataset: &Array2, test_size: f64) -> (Array2, Array2,
Array2, Array2) {



   // ...

}
```

Train the Bagging ensemble of Decision Trees:

```
fn main() -> Result<(), BoxError>> {

   let dataset = load_dataset()?;

    let (x_train, x_test, y_train, y_test) = train_test_split(&dataset, 0.2);

    let mut rf = RandomForestRegressor::new(50); // 50 Decision Trees
```

```
  rf.fit(&x_train, &y_train.view().into_shape(y_train.len()).unwrap())?;

   // Evaluate the model on the test set

}
```

Evaluate the model:

```
fn evaluate(rf: &RandomForestRegressor, x_test: &Array2, y_test:
&Array2) {

  let predictions = Array::from(rf.predict(&x_test));

   // Compute evaluation metrics like Mean Squared Error, R2 Score, etc.

}
```

This example outlines the process of constructing a Bagging ensemble of Decision Trees using Rust programming language and the randomforest crate. Keep in mind that prior to training the model, it is essential to preprocess the dataset. This includes converting categorical variables into numerical values and addressing any missing values. Employing Rust and the randomforest crate enables you to create a robust, efficient, and scalable Bagging ensemble model, which can effectively handle various data complexities and improve overall prediction accuracy.

Summary

This entire chapter delves into the realm of non-linear models and the advancement of statistics into machine learning. It begins by discussing various non-linear models and the significant breakthroughs that machine learning algorithms and techniques have brought to statistical challenges. These models play a vital role in capturing complex relationships within data that linear models often fail to capture.

The chapter first introduces Decision Trees, a popular non-linear model, and discusses their mathematical architecture. It then provides a step-by-step demonstration of implementing Decision Trees on the given dataset using Rust. The example shows how to preprocess the data, train the model, and evaluate its performance. The chapter then moves on to Random Forests, which are ensembles of Decision Trees, and explains their mechanism and how they can help improve model accuracy and generalization. A sample implementation of Random Forests using Rust is provided, showing how to create an ensemble of Decision Trees, train the model, and evaluate its performance.

Next, the chapter covers Support Vector Machines (SVM), a powerful technique for classification and regression tasks. It explains the concepts behind SVMs and provides a step-by-step example of implementing them using Rust. The example demonstrates how to load and preprocess the data, train an SVM model, and evaluate its performance on the given dataset. The chapter then explores Ensemble methods, a class of machine learning techniques that combine the predictions of multiple base models to achieve better accuracy and generalization. Popular ensemble methods, such as Bagging, Boosting, and Stacking, are discussed in detail. The chapter provides a step-by-step illustration of building a simple Bagging ensemble of Decision Trees for the given dataset using Rust, including loading the dataset, preprocessing it, training the ensemble, and evaluating its performance.

To sum it up, this chapter presents a comprehensive overview of non-linear models and their significance in addressing complex data relationships. It explains the underlying concepts of these models and provides practical examples using Rust, demonstrating the growing potential of Rust in the machine learning and statistics domain.

Chapter 10: Model Evaluation and Validation

Model Evaluation and Validation

Introduction

Model Evaluation and Validation Techniques are crucial aspects of the machine learning and statistical modeling process. Model evaluation and validation involve assessing the performance of a model and ensuring its generalization to unseen data. These techniques help identify the best models, fine-tune their parameters, and avoid overfitting, thus leading to better predictions and more accurate insights.

The need for model evaluation and validation arises from the inherent complexity and uncertainty in real-world data. Given a dataset, multiple models may capture the underlying relationships in the data to varying degrees of success. Therefore, it is essential to evaluate and compare different models to choose the one that performs the best on unseen data. Several model evaluation and validation techniques are used to assess model performance and ensure that they generalize well to new data.

Some popular techniques include:
Train-test split: This technique involves dividing the dataset into training and test sets. The model is trained on the training set and its performance is evaluated on the test set. This approach helps assess the model's ability to generalize to unseen data.

Cross-validation: Cross-validation is an extension of the train-test split method, where the dataset is divided into 'k' equally-sized partitions or "folds." The model is trained and tested 'k' times, each time using a different fold as the test set and the remaining folds as the training set. The

average performance across all 'k' iterations is used to evaluate the model. Cross-validation helps to mitigate the risk of overfitting and provides a more robust estimate of model performance.
Metrics: Various metrics are used to measure model performance, depending on the task (e.g., classification, regression, clustering). For classification tasks, metrics such as accuracy, precision, recall, F1-score, and area under the ROC curve (AUC-ROC) are commonly used. For regression tasks, metrics like mean squared error (MSE), mean absolute error (MAE), and R-squared are popular.
Confusion Matrix: A confusion matrix is a table that shows the true positive (TP), true negative (TN), false positive (FP), and false negative (FN) predictions for a classification model. It helps visualize the performance of the model and identify areas for improvement.
Learning Curves: Learning curves plot the model's performance on the training and validation sets as a function of the number of training samples or iterations. They can help diagnose issues like overfitting and underfitting and guide the selection of the optimal model complexity.
Hyperparameter tuning: Models often have hyperparameters that control their complexity or learning process. Techniques like grid search, random search, and Bayesian optimization can be used to find the best set of hyperparameters for a given model, leading to improved performance.

Model selection: Model selection techniques, such as Akaike Information Criterion (AIC), Bayesian Information Criterion (BIC), and cross-validated performance, help in comparing and choosing the best model among several competing models.

Model evaluation and validation techniques are essential tools for selecting and fine-tuning machine learning and statistical models. These techniques help assess model performance, ensure generalization, and avoid overfitting, ultimately resulting in more accurate and reliable predictions.

Train-test Split Technique

Exploring Train-test Split

Train-test split is a fundamental and efficient method used to assess the performance of a machine learning model. This technique revolves around partitioning the available dataset into two distinct sets: the training set and the test set. Generally, a larger portion of the dataset is allocated for training purposes (typically around 70-80%), while the remaining smaller portion is reserved for testing (usually about 20-30%). The primary objective of this method is to teach the model using the training set, which contains various features and associated target values. Once the model has been trained, its performance is then evaluated using the test set. The test set, containing previously unseen data, provides a means to measure how effectively the model can generalize and make accurate predictions on new, unexplored data points.

The train-test split technique is crucial in preventing overfitting, a common issue in machine learning where a model performs exceptionally well on the training data but fails to deliver satisfactory results when exposed to unseen data. By segregating the dataset into separate training and testing sets, it becomes possible to identify and mitigate overfitting early in the development process.

Implementing Train-test Split

The given below is a step-by-step demonstration of how to carry out train-test split on the given dataset using Rust:

Import necessary libraries and load the dataset

First, import the necessary libraries:

```
extern crate ndarray;


extern crate ndarray_csv;

extern crate rand;

use ndarray::{Array2, Axis};

use ndarray_csv::{Array2Reader, Array2Writer};

use rand::seq::SliceRandom;

use std::fs::File;

use std::io::{BufReader, BufWriter};
```

Now, load the dataset:

```
let file =
File::open("https://raw.githubusercontent.com/kittenpub/database-
repository/main/ds_salaries.csv").unwrap();

let buf_reader = BufReader::new(file);

let mut dataset = Array2::::read_csv(buf_reader).unwrap();
```

Shuffle the dataset

To ensure a fair distribution of data points in the training and test sets, shuffle the dataset before splitting:

```
let mut rng = rand::thread_rng();

dataset.shuffle(Axis(0), &mut rng);
```

Determine the train-test split ratio and sizes

Choose a train-test split ratio, such as 80% for the training set and 20% for the test set:

```
let train_ratio: f64 = 0.8;

let train_size = (train_ratio * dataset.nrows() as f64).round() as usize;

let test_size = dataset.nrows() - train_size;
```

Split the dataset into training and test sets

Slice the dataset to create the training and test sets:

```
let train_set = dataset.slice(s![0..train_size, ..]);

let test_set = dataset.slice(s![train_size.., ..]);
```

Save the training and test sets to CSV files (optional)

You can save the resulting sets to CSV files for further analysis:

```
let train_file = File::create("train_set.csv").unwrap();

let test_file = File::create("test_set.csv").unwrap();

let train_buf_writer = BufWriter::new(train_file);

let test_buf_writer = BufWriter::new(test_file);

let mut train_writer = csv::Writer::from_writer(train_buf_writer);

let mut test_writer = csv::Writer::from_writer(test_buf_writer);

train_set.write_csv(&mut train_writer).unwrap();

test_set.write_csv(&mut test_writer).unwrap();
```

Once done, you can then train your model on the train_set and evaluate its performance on the test_set. The train-test split allows you to check if your model can generalize well to new, unseen data.

Cross-validation Technique

Understanding Cross-validation

Cross-validation represents a more sophisticated technique for evaluating models in comparison to the train-test split method. This approach is especially valuable when working with limited data since it utilizes the entire dataset for both training and testing purposes, thus mitigating the risk of overfitting. Among the various cross-validation strategies, K-fold cross-validation is widely recognized and employed.
The K-fold cross-validation method involves partitioning the dataset into K equally sized segments, referred to as folds. The model undergoes training and evaluation K times, with each iteration using a unique fold as the testing set and the remaining K-1 folds as the training set. This process ensures that every data point has the opportunity to be part of the test set, enhancing the reliability of the model evaluation. Once all iterations are complete, the final performance metric is derived by calculating the average of the performance metrics obtained during each individual iteration. This aggregated result offers a more comprehensive and accurate understanding of the model's performance, contributing to the development of more robust and reliable machine learning models that are better equipped to handle real-world scenarios.

Implementing K-fold Cross-validation

The given below is a step-by-step demonstration of how to carry out K-fold cross-validation on the given dataset using Rust:

Define the number of folds

Choose the number of folds (K) for cross-validation. A common choice is K=10:

```
let k: usize = 10;
```

Calculate fold size

Determine the size of each fold:

```
let fold_size = dataset.nrows() / k;
```

Perform K-fold cross-validation

Iterate through the K folds, each time using a different fold as the test set and the remaining K-1 folds as the training set:

```
let mut performance_metrics: Vec = Vec::new();

for i in 0..k {

    // Prepare the training and test sets

    let start = i * fold_size;
    let end = (i + 1) * fold_size;

   let train_set = dataset.slice(s![(0..start).chain(end..dataset.nrows()), ..]);

   let test_set = dataset.slice(s![start..end, ..]);

    // Train your model on the train_set

    // Evaluate the model on the test_set

    // Store the performance metric in the performance_metrics vector

}

// Calculate the average performance metric

let avg_performance_metric: f64 = performance_metrics.iter().sum::() / k
as f64;
```

The average performance metric provides a more robust estimate of the model's generalization ability. It reduces the variance in the performance metric compared to a simple train-test split, giving you greater confidence in the model's performance on unseen data.

Hyperparameter Tuning

Overview

Hyperparameter tuning is the crucial process of optimizing a model's hyperparameters to achieve enhanced performance. Hyperparameters, unlike other parameters, are not learned from the data; rather, they are predetermined by the model's creator before the training process begins. Examples of hyperparameters include learning rates, regularization parameters, and the number of hidden layers in a neural network.

To provide a comprehensive understanding of hyperparameter tuning, let's delve into the most prevalent methods employed in this process:
Grid search: The grid search method involves defining a range of potential values for each hyperparameter and then systematically testing every possible combination. While effective, this technique can be computationally demanding, particularly for models with an extensive set of hyperparameters.
Random search: As an alternative to the exhaustive approach taken by grid search, random search selects and evaluates a random assortment of hyperparameter combinations. This technique frequently uncovers suitable hyperparameter values more rapidly than grid search, offering a more efficient solution.

Bayesian optimization: This advanced method entails constructing a probabilistic model that captures the relationship between hyperparameters and overall model performance. Through iterative updates to this model, Bayesian optimization intelligently explores the hyperparameter space and swiftly converges to an optimal solution.

Each of these hyperparameter tuning methods has its advantages and drawbacks. Grid search offers a thorough exploration of the hyperparameter space but can be resource-intensive. Random search, while faster, may not be as exhaustive in its search for optimal hyperparameters. Bayesian optimization strikes a balance between the two, offering an intelligent and efficient approach to finding the best hyperparameters.

Perform Hyperparameter Tuning using Grid Search

The given below is a step-by-step demonstration of how to perform hyperparameter tuning using the given dataset in Rust:

Define the hyperparameter search space

```
let hyperparameter_space = [

  // Hyperparameter 1: (start_value, end_value)

  (0.0, 1.0),

  // Hyperparameter 2: (start_value, end_value)

  (1, 100),

   // Add more hyperparameters as needed
];
```

Choose a search method

We'll use grid search. Create a function that generates all possible combinations of hyperparameters:

```
fn generate_hyperparameter_combinations(hyperparameter_space: &[(f64, f64)]) -> Vec> {

    // Implement the function to generate all possible combinations

}
```

Perform hyperparameter tuning

Iterate through the hyperparameter combinations and train the model with each combination:

```
let mut best_hyperparameters: Vec = vec![];

let mut best_performance_metric = f64::MIN;

for combination in generate_hyperparameter_combinations(&hyperparameter_space) {
    // Train your model using the hyperparameter combination

   // Evaluate the model performance

    // Update the best_hyperparameters and best_performance_metric if
necessary

}
```

Upon completion of the tuning process, the best_hyperparameters vector holds the most suitable hyperparameters tailored for the provided dataset. It is crucial to emphasize that this specific example employs grid search as its optimization technique. However, alternative methods such as random search or Bayesian optimization can also be implemented to achieve similar outcomes. To successfully integrate these alternative strategies, one must adjust the hyperparameter search approach in a suitable manner. By doing so, you can effectively explore different hyperparameter configurations and potentially discover even more optimal solutions tailored to the unique characteristics of your dataset. Therefore, adopting an appropriate hyperparameter search technique is essential for obtaining the best possible model performance.

Model Selection Techniques: AIC and BIC

The Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) are powerful model selection techniques that assist in determining the optimal model among a range of candidate models. Both criteria emphasize the balance between goodness-of-fit and model complexity to ensure the most suitable model is chosen for a given dataset.

Akaike Information Criterion (AIC)

AIC, or Akaike Information Criterion, is a model selection metric that evaluates a model's goodness-of-fit while considering the number of parameters involved. A lower AIC value signifies a better-fitting model.

The formula for AIC is as follows:

$$AIC = 2k - 2\ln(L)$$

In this equation, 'k' represents the total number of model parameters, and 'L' denotes the maximized likelihood of the model given the data.

Bayesian Information Criterion (BIC)

BIC, or Bayesian Information Criterion, is another model selection metric that assesses the goodness-of-fit of a model while incorporating a penalty for model complexity. Similar to AIC, a lower BIC value corresponds to a better-fitting model.

The formula for BIC is as follows:

BIC = k * ln(n) - 2ln(L)

In this equation, 'k' stands for the total number of model parameters, 'n' represents the number of observations, and 'L' refers to the maximized likelihood of the model given the data.

Although AIC and BIC share conceptual similarities, BIC generally imposes a more significant penalty on model complexity than AIC. Consequently, BIC tends to favor simpler models in comparison to AIC. Both AIC and BIC are valuable tools in model selection, with each criterion focusing on achieving an optimal balance between goodness-of-fit and model complexity. By taking into account the number of parameters and the maximized likelihood of the model given the data, these criteria help researchers and data analysts identify the most suitable model for a particular dataset. While AIC and BIC share some similarities, the heavier penalty imposed on model complexity by BIC often results in the selection of simpler models compared to AIC. Ultimately, the choice between AIC and BIC depends on the specific context and goals of the analysis, as well as the preferences of the researcher or analyst involved.

Implement AIC and BIC

To implement AIC and BIC calculation in Rust for the given dataset, follow these steps:

Define a function to compute the likelihood of the model given the data.

```
fn compute_likelihood(model: &Model, data: &Data) -> f64 {

    // Implement the function to compute the likelihood of the model given
the data

}
```

Define functions to calculate AIC and BIC.

```
fn calculate_aic(model: &Model, data: &Data) -> f64 {

    let k = model.number_of_parameters() as f64;

   let l = compute_likelihood(model, data).ln();

    2.0 * k - 2.0 * l

}

fn calculate_bic(model: &Model, data: &Data) -> f64 {



    let k = model.number_of_parameters() as f64;

    let n = data.number_of_observations() as f64;
```

```
  let l = compute_likelihood(model, data).ln();

   k * n.ln() - 2.0 * l

}
```

Evaluate the AIC and BIC for each candidate model.

```
for model in candidate_models {

  let aic = calculate_aic(&model, &data);

  let bic = calculate_bic(&model, &data);

   // Store and compare the AIC and BIC values to choose the best model

}
```

When evaluating various models, comparing the Akaike Information Criterion (AIC) and Bayesian Information Criterion (BIC) values enables you to identify the model that strikes an optimal balance between goodness-of-fit and model complexity, tailored to your specific dataset. These criteria assist in preventing overfitting or underfitting by penalizing more complex models, ultimately guiding the selection process towards a model that provides a robust and accurate representation of the underlying data structure.

Resampling Methods

Resampling methods constitute a set of statistical techniques that play a crucial role in model evaluation, hypothesis testing, and estimation of uncertainties inherent in data. These methods revolve around the core concept of recurrently sampling from the available data and recalculating relevant statistics. This process allows researchers and analysts to gain insights into the variability of the statistics, or to estimate their underlying distribution, thus enhancing the robustness of their conclusions.

Two prevalent types of resampling methods that have gained considerable traction in the field of statistics are bootstrapping and permutation tests.

Bootstrapping

Bootstrapping is a resampling method used to estimate the sampling distribution of a statistic by repeatedly drawing samples (with replacement) from the original dataset. It is particularly useful when the underlying distribution of the data is unknown or when the sample size is small. Bootstrapping can be applied to estimate confidence intervals, test hypotheses, and evaluate the stability of model parameters.

In a bootstrap procedure, you:
Draw a random sample (with replacement) of the same size as the original dataset.
- Compute the statistic of interest from the resampled data.

Repeat the both previous steps a large number of times (e.g., 1,000 or 10,000) to generate a distribution of the statistic.

Estimate the sampling distribution of the statistic, such as calculating confidence intervals or standard errors.

Permutation Tests

Permutation tests, also known as randomization tests or exact tests, are non-parametric resampling methods used for hypothesis testing. They involve rearranging the observed data to create all possible permutations of the data (or a large random subset of permutations), and then calculating the test statistic for each permutation. By comparing the observed test statistic to the distribution of permuted test statistics, you can calculate a p-value to test the null hypothesis.

In a permutation test, you:

- Calculate the observed test statistic for the original data.

Randomly permute the data (or a subset of the data) and calculate the test statistic for the permuted data.
Repeat the previous step a large number of times (e.g., 1,000 or 10,000) to generate a distribution of permuted test statistics.
Calculate the p-value as the proportion of permuted test statistics that are as extreme or more extreme than the observed test statistic.

Both bootstrapping and permutation tests offer flexible and powerful methods for statistical inference, as they do not rely on strong assumptions about the underlying data distribution. They can be applied to a wide range of problems and are particularly useful when working with small sample sizes or non-normally distributed data.

Perform Bootstrapping and Permutation Test

In the below example, I'll demonstrate how to perform bootstrapping and permutation tests using Rust. We'll use the rand crate for random number generation, and the ndarray crate for working with arrays. Add these dependencies to your Cargo.toml:

```
[dependencies]

rand = "0.8"

ndarray = "0.15"
```

For this demonstration, let's assume we want to estimate the mean salary and compare the mean salaries between two groups (e.g., two different job titles) in our dataset. First, import the required modules:

```
use rand::seq::SliceRandom;

use ndarray::{Array, Axis};
```

Load the dataset and preprocess it as we did before. Then, split the data into two groups based on job titles:

```
// Assuming the dataset is loaded and preprocessed as a DataFrame named "df"
```

```
let group1 = df.select_rows(&df.column("job_title").eq("Data Scientist"));

let group2 = df.select_rows(&df.column("job_title").eq("Data
Engineer"));

let salaries1 = Array::from(group1.column("salaryinusd").to_vec());

let salaries2 = Array::from(group2.column("salaryinusd").to_vec());
```

<u>Implementing Bootstrapping</u>

Create a function to draw a bootstrap sample:

```
fn bootstrap_sampleClone>(data: &[T], rng: &mut
rand::rngs::ThreadRng) -> Vec {

  data.choose_multiple(rng, data.len()).cloned().collect()



}
```

Run the bootstrap procedure:

```
let mut rng = rand::thread_rng();

let n_bootstrap = 1000;
let mut bootstrap_means1 = Vec::with_capacity(n_bootstrap);

let mut bootstrap_means2 = Vec::with_capacity(n_bootstrap);

for _ in 0..n_bootstrap {

    let sample1 = bootstrap_sample(&salaries1, &mut rng);

    let sample2 = bootstrap_sample(&salaries2, &mut rng);

   let mean1 = Array::from(sample1).mean_axis(Axis(0)).unwrap()[0];

   let mean2 = Array::from(sample2).mean_axis(Axis(0)).unwrap()[0];

   bootstrap_means1.push(mean1);

   bootstrap_means2.push(mean2);

}
```

Calculate the confidence intervals:

```
bootstrap_means1.sort_unstable_by(|a, b| a.partial_cmp(b).unwrap());

bootstrap_means2.sort_unstable_by(|a, b| a.partial_cmp(b).unwrap());

let ci1 = (
    bootstrap_means1[(0.025 * n_bootstrap as f64) as usize],

    bootstrap_means1[(0.975 * n_bootstrap as f64) as usize],

);

let ci2 = (

    bootstrap_means2[(0.025 * n_bootstrap as f64) as usize],

    bootstrap_means2[(0.975 * n_bootstrap as f64) as usize],

);


println!("Group 1 mean salary 95% CI: {:?}", ci1);

println!("Group 2 mean salary 95% CI: {:?}", ci2);
```

<u>Implementing Permutation Test</u>

Calculate the observed difference in means:

```
let observed_diff = salaries1.mean_axis(Axis(0)).unwrap()[0] -
salaries2.mean_axis(Axis(0)).unwrap()[0];
```

Run the permutation test:

```
fn permute_groupsClone>(group1: &[T], group2: &[T], rng: &mut
rand::rngs::

ThreadRng) -> (Vec, Vec) {

let mut combined = group1.to_owned();

combined.extend(group2.iter().cloned());

combined.shuffle(rng);

let split_at = group1.len();

combined.split_at(split_at)

}

let n_permutations = 1000;

let mut permuted_diffs = Vec::with_capacity(n_permutations);

for _ in 0..n_permutations {

let (permuted_group1, permuted_group2) = permute_groups(&salaries1,
&salaries2, &mut rng);

let mean1 = Array::from(permuted_group1).mean_axis(Axis(0)).unwrap()
[0];
let mean2 = Array::from(permuted_group2).mean_axis(Axis(0)).unwrap()
[0];

permuted_diffs.push(mean1 - mean2);

}
```

Calculate the p-value:

```
let n_extreme = permuted_diffs.iter().filter(|&&d| d.abs() >=
observed_diff.abs()).count();

let p_value = n_extreme as f64 / n_permutations as f64;

println!("Permutation test p-value: {:?}", p_value);
```

We have successfully completed both bootstrapping and permutation tests on our dataset, allowing us to gain valuable insights. The bootstrapping procedure was employed to generate confidence intervals for the mean salaries of both groups under consideration. By doing so, we were able to estimate the range within which the true population means are likely to fall, giving us a better understanding of the data. On the other hand, the permutation test was utilized to assess the statistical significance of the observed difference in mean salaries between the two groups. This test provided us with a p-value, which helped us determine whether the observed difference was due to chance or if it reflected a genuine disparity between the groups. By using these complementary approaches, we were able to conduct a robust analysis of our dataset, providing crucial information to guide future decisions and actions.

Summary

In this chapter, we delved into the importance of model evaluation and validation, which are crucial for assessing the performance of machine learning and statistical models. We discussed various techniques to evaluate and validate models, including train-test split, cross-validation, hyperparameter tuning, model selection techniques like AIC and BIC, and resampling methods such as bootstrapping and permutation tests.

We began by understanding the train-test split, which involves dividing the data into training and testing subsets, and then demonstrated how to carry out this technique on the given dataset. Next, we explored cross-validation, a more robust method to assess model performance, and implemented k-fold cross-validation on the dataset. After that, we delved into hyperparameter tuning, discussing its significance in optimizing model performance, and demonstrated how to conduct a simple grid search to find the best hyperparameters. We also touched upon model selection techniques like AIC and BIC, which help in comparing and selecting the best-fitting models. Finally, we covered resampling methods, focusing on bootstrapping and permutation tests. We explained their conceptual foundations and demonstrated how to apply both techniques to the given dataset. The bootstrapping procedure provided confidence intervals for the mean salaries, while the permutation test assessed the significance of the observed difference in mean salaries.

To sum it up, this chapter emphasized the importance of model evaluation and validation techniques, which are essential for ensuring that the models we build are robust, reliable, and accurate. The techniques presented here provide a solid foundation for understanding and applying different methods for evaluating and validating machine learning and statistical models in real-world scenarios.

Chapter 11: Text and Natural Language Processing

Overview of Natural Language Processing (NLP)

The rise and development of Natural Language Processing (NLP) can be traced back to the mid-20th century, with early roots in computational linguistics and artificial intelligence (AI). NLP is a subfield of AI that aims to enable computers to understand, interpret, and generate human language. Over the years, NLP has evolved tremendously, powered by advancements in technology, data availability, and AI algorithms. Its adoption has led to numerous practical applications that have transformed our lives.

The birth of NLP dates back to the 1950s with the development of the first-ever machine translation system, known as the Georgetown-IBM experiment. The system translated Russian sentences into English, demonstrating the potential for machines to process natural language. In the 1960s, the focus shifted to creating rule-based systems, which relied on manually crafted grammar rules and lexicons. An example is the work of linguist Noam Chomsky, who developed formal grammars as a basis for understanding language structure. In the 1970s, researchers began to develop algorithms that could parse and generate text. These early systems, such as SHRDLU and LUNAR, could process simple commands and queries in restricted domains. During this time, the concept of "scripts" was introduced by Roger Schank, which allowed systems to understand and generate text by following predefined scenarios.

The 1980s saw a shift towards probabilistic and statistical methods in NLP. Researchers started using statistical models, like the hidden Markov models (HMMs), to analyze and generate text based on patterns observed in large corpora. This marked the beginning of the data-driven approach in NLP. The 1990s saw the rise of machine learning techniques, such as decision trees and support vector machines, which enabled the development of more sophisticated NLP systems. In the 2000s, the internet facilitated the availability of massive amounts of text data, propelling the development of more advanced statistical methods. Researchers started using Bayesian models and deep learning techniques, such as recurrent neural networks (RNNs), to create more accurate and efficient NLP systems. During this time, the focus shifted towards end-to-end systems that could learn directly from data, without relying on manual feature engineering.

The breakthrough moment for NLP came in 2013 with the introduction of word embeddings, specifically Word2Vec by Mikolov et al. This technique enabled the representation of words as continuous vectors, capturing semantic and syntactic relationships among them. This spurred a flurry of research, leading to the development of more advanced embeddings, such as GloVe and fastText. The transformer architecture, introduced by Vaswani et al. in 2017, enabled the creation of highly efficient and scalable models that could handle long-range dependencies in text. This laid the groundwork for the development of large-scale pre-trained models, such as OpenAI's GPT and Google's BERT, which demonstrated unprecedented performance across a wide range of NLP tasks.

The rapid advancement in NLP has led to its widespread adoption across industries. Today, NLP is used in various applications, including search engines, virtual assistants, sentiment analysis, machine translation, text summarization, and chatbots, among others. Organizations are leveraging

NLP to enhance customer support, automate content generation, and derive insights from unstructured data.

Key Processes of NLP

NLP has made significant advancements, leading to its widespread use in applications such as chatbots, virtual assistants, sentiment analysis, machine translation, and more. One of the reasons for its success is the use of key processes that contribute to NLP's ability to understand and process text data.

One of the essential processes in NLP is tokenization. Tokenization involves breaking down text into individual words or tokens, which is crucial as it allows NLP systems to analyze text at the word level. By analyzing words individually, NLP algorithms can identify patterns and relationships between words and phrases, which is useful for tasks such as sentiment analysis and named entity recognition.

Another important process in NLP is stopword removal. Stopwords are common words that do not carry much meaning in the context of text analysis, such as "the," "and," and "is." Removing stopwords can help reduce the dimensionality of the data, making it easier for NLP algorithms to analyze and process text.

Stemming and lemmatization are also key processes in NLP. Both processes aim to reduce words to their base or root form, which can help consolidate similar words and reduce noise in the text data. Stemming is a crude process that removes affixes from words, while lemmatization is more sophisticated, taking into account the morphological structure and context of the word.

Part-of-speech (POS) tagging is another process that helps NLP algorithms understand the syntactic structure and relationships between words in the text. POS tagging involves identifying the grammatical category of each word in a sentence, such as noun, verb, adjective, and adverb.

Named Entity Recognition (NER) is the process of identifying and classifying named entities, such as people, organizations, and locations, in the text. NER can be useful for extracting structured information from unstructured text data, such as customer feedback or social media posts.

Dependency parsing is another important process in NLP that involves analyzing the grammatical structure of a sentence to identify the relationships between words. By identifying the relationships between words, NLP algorithms can understand the semantic meaning of sentences and identify complex relationships in the text.

Sentiment analysis is a process that aims to determine the sentiment or emotion expressed in a piece of text. Sentiment analysis can be useful for understanding public opinion, customer feedback, and social media trends.

Machine translation is a process that focuses on automatically translating text from one language to another. Recent advancements in neural machine translation have greatly improved the quality and efficiency of this process, making it possible for people to communicate with others who speak different languages.

Text summarization is a process that involves creating a concise and coherent summary of a large piece of text while preserving the main ideas and essential information. Text summarization can be useful for extracting key insights from large volumes of text data, such as news articles or research papers. Finally, question answering systems aim to provide accurate and relevant answers to natural language questions posed by users. These systems have gained prominence with the rise of digital assistants like Siri, Alexa, and Google Assistant.

These above key processes in NLP enable machines to understand and process human language, making it possible to perform a variety of tasks that were once impossible. By breaking down text into individual words or tokens, removing stop words, and reducing words to their base or root form, NLP algorithms can analyze text at the word level, identify patterns and relationships between words, and understand the semantic meaning of sentences. These processes, along with others like POS tagging, NER, and machine translation, have led to significant advancements in NLP and its applications. NLP has emerged as a powerful tool for processing and understanding the vast amounts of text data generated daily. The various processes and techniques involved in NLP have made it possible to analyze, interpret, and generate human language effectively, paving the way for numerous applications that have transformed the way we interact with technology and access information.

Text Preprocessing and Tokenization

Text preprocessing and tokenization are two essential tasks in natural language processing (NLP) that are performed on raw textual data before further analysis or processing. Text preprocessing involves cleaning and transforming the raw data to make it suitable for analysis. Tokenization is the process of breaking down the text into smaller meaningful units called tokens.

Text preprocessing is necessary to ensure that the text is free of unwanted noise such as special characters, punctuations, numbers, and other non-textual elements. It also involves converting the text to a standardized format, such as lowercase or uppercase, removing stop words, and stemming or lemmatizing words. The main goal of text preprocessing is to improve the quality of the text data and make it more suitable for further analysis.

Tokenization is a process of breaking down the text into smaller tokens that can be words, phrases, or even sentences. Tokenization is an essential step in NLP because most machine learning models work with numerical data. By converting the text data into tokens, we can represent the textual data numerically, which can be processed by machine learning models.

Key Preprocessing Techniques

Some common text preprocessing techniques include:

Lowercasing: Converting all text to lowercase can help establish uniformity and make it easier to match tokens, as algorithms are usually case-sensitive.

Removing punctuation, special characters, and numbers: These elements can introduce noise and are often unnecessary for text analysis. By removing them, you can reduce the complexity of the text.
Removing stopwords: Stopwords are common words like 'the,' 'is,' and 'in,' which don't carry much meaning and can be removed to reduce the dimensionality of the text data.
Stemming and Lemmatization: These techniques reduce words to their base or root form. Stemming involves truncating words, while lemmatization uses a linguistic approach to convert words to their base form. Both techniques help in grouping similar words together.

Common Tokenization Approaches

After preprocessing the text, tokenization can be performed using several approaches:
Word tokenization: In this method, the text is split into individual words based on spaces or punctuation marks.
Sentence tokenization: The text is split into sentences, usually based on punctuation marks like periods, question marks, or exclamation marks.
Subword tokenization: This approach breaks text down into smaller units like syllables, characters, or morphemes. It's particularly useful for languages where word boundaries are not easily identifiable, or for tasks that require a more granular understanding of the text.

Implementing Text Preprocessing and Tokenization

Let us try preprocessing the text and tokenizing the textual dataset using Rust. Following is the sample dataset carrying more than 200,000 text messages:

https://raw.githubusercontent.com/kittenpub/database-repository/main/200000_noemoticon_nlp_data.csv

To begin with text preprocessing and tokenization, we'll follow these steps:

Import necessary libraries

```
use csv::Reader;

use reqwest::blocking::get;

use std::error::Error;

use unicode_segmentation::UnicodeSegmentation;
```

Load the data from the given CSV

```
fn load_data(url: &str) -> Result, BoxError>> {

  let mut data = Vec::new();

  let response = get(url)?.text()?;
```

```
    let mut reader = Reader::from_reader(response.as_bytes());
    for result in reader.records() {
        let record = result?;
        data.push(record[5].to_string());
    }
    Ok(data)
}
```

Preprocess the text

```
fn preprocess_text(text: &str) -> String {
    text.to_lowercase()
        .chars()
        .filter(|c| c.is_alphanumeric() || c.is_whitespace())
        .collect()
}
```

Tokenize the text

```
fn tokenize(text: &str) -> Vec {


  text.unicode_words().map(|s| s.to_string()).collect()

}
```

<u>Sample Program to Perform Preprocessing and Tokenization</u>

Now, let's demonstrate how to use these functions on the given database.

```
fn main() -> Result<(), BoxError>> {

  let url = "https://raw.githubusercontent.com/kittenpub/database-
repository/main/200000_noemoticon_nlp_data.csv";

  let data = load_data(url)?;

   // Process and tokenize the first 10 records

  for text in data.iter().take(10) {

      let preprocessed_text = preprocess_text(text);

      let tokens = tokenize(&preprocessed_text);
```

```
        println!("Original text: {}", text);

        println!("Tokens: {:?}", tokens);

        println!();

    }

    Ok(())

}
```

This code snippet will load the data from the provided CSV file, preprocess the first 10 records by converting them to lowercase and removing non-alphanumeric characters, and tokenize the preprocessed text by splitting it into words using the UnicodeSegmentation crate.

The output of the above codes will display the original text and the corresponding tokens as below:

```
Original text: @switchfoot http://twitpic.com/2y1zl - Awww, that's a bummer.  You shoulda got David Carr of Third Day to do it. ;D

Tokens: ["switchfoot", "httptwitpiccom2y1zl", "awww", "thats", "a", "bummer", "you", "shoulda", "got", "david", "carr", "of", "third", "day", "to", "do", "it", "d"]
```

```
Original text: is upset that he can't update his Facebook by texting it... and might cry as a result  School today also. Blah!

Tokens: ["is", "upset", "that", "he", "cant", "update", "his", "facebook", "by", "texting", "it", "and", "might", "cry", "as", "a", "result", "school", "today", "also", "blah"]

Original text: @Kenichan I dived many times for the ball. Managed to save 50%  The rest go out of bounds

Tokens: ["kenichan", "i", "dived", "many", "times", "for", "the", "ball", "managed", "to", "save", "50", "the", "rest", "go", "out", "of", "bounds"]

...
```

The output consists of the original text from the dataset and the corresponding tokens generated after preprocessing and tokenization. The tokens have been transformed to lowercase, and non-alphanumeric characters have been removed.

Stopword Removal Process

Stopword removal is the process of eliminating common words that don't carry much meaning and are often found in large quantities in text data. Examples of stopwords include "the," "is," "in," "and," etc. Removing stopwords can help reduce the dimensionality of the text data and improve the performance of text processing algorithms.

Sample Program to Perform Stopword Removal

To perform stopword removal on the given database, you can follow these steps:

Import necessary libraries and load the data:

```
extern crate reqwest;

extern crate csv;

extern crate regex;

use csv::Reader;

use regex::Regex;

use std::collections::HashSet;
```

```
#[tokio::main]

async fn main() -> Result<(), Boxstd::error::Error>> {

  let url = "https://raw.githubusercontent.com/kittenpub/database-
repository/main/200000_noemoticon_nlp_data.csv";

  let text = reqwest::get(url).await?.text().await?;

  // Read the CSV data

  let mut reader = Reader::from_reader(text.as_bytes());

  let mut records = reader.records();

   // Your text preprocessing steps here

  Ok(())

}
```

Create a list of stopwords
You can use an existing list of stopwords or create your own based on the specific requirements of your analysis. Below is a quick sample:

```
fn get_stopwords() -> HashSet {
    let stopwords = [
        "a", "an", "and", "are", "as", "at", "be", "by", "for", "from", "has", "he", "in", "is",
        "it", "its", "of", "on", "that", "the", "to", "was", "were", "will", "with",
    ];
    stopwords.iter().map(|s| s.to_string()).collect()
}
```

Implement a function to remove stopwords from a given text

```
fn remove_stopwords(text: &str, stopwords: &HashSet) -> String {
    let re = Regex::new(r"[^\w\s]").unwrap();
    let cleaned_text = re.replace_all(text, "").to_string();
    let words: Vec<&str> = cleaned_text.split_whitespace().collect();
    words
        .iter()
        .filter(|&word| !stopwords.contains(&word.to_lowercase()))

        .cloned()

        .collect::>()

        .join(" ")

}
```

Apply the stopword removal function

```
let stopwords = get_stopwords();

while let Some(result) = records.next() {

  let record = result?;

  let text = &record[1];

  let cleaned_text = remove_stopwords(text, &stopwords);

   // Continue processing the cleaned_text or store it for further analysis

}
```

This above snippet or the sample program demonstrates how to remove stopwords from the given dataset. You can modify the stopwords list and text preprocessing functions to fit your project demands

Stemming and Lemmatization

Stemming and lemmatization are natural language processing techniques used to reduce words to their root or base forms, which can improve text analysis by reducing variations of the same word. Stemming is a more crude approach that chops off word endings, while lemmatization is a more advanced technique that considers the context and part of speech to derive the base form of a word.

Unfortunately, a readily accessible solution for lemmatization has yet to be developed. While Rust may not yet possess a native and popular library for stemming and lemmatization comparable to Python's NLTK or SpaCy, it does offer the rust_stemmers library as a viable option for stemming.

Perform Stemming

To perform stemming with the rust_stemmers library, add the rust_stemmers library to your Cargo.toml:

```
[dependencies]

rust_stemmers = "1.2.0"
```

Import the rust_stemmers library and its necessary components:

```
extern crate rust_stemmers;
```

```
use rust_stemmers::{Algorithm, Stemmer};
```

Create a stemming function using the desired stemming algorithm (e.g., Porter):

```
fn stem_text(text: &str) -> String {

  let stemmer = Stemmer::create(Algorithm::Porter);

  let words: Vec<&str> = text.split_whitespace().collect();

  words

     .iter()



     .map(|word| stemmer.stem(word))

     .collect::>()

     .join(" ")

}
```

Apply the stemming function to the cleaned text:

```
let stemmed_text = stem_text(&cleaned_text);
```

```
// Continue processing the stemmed_text or store it for further analysis
```

Kindly be aware that this sample program showcases the concept of stemming rather than lemmatization. To achieve lemmatization in Rust, a more intricate approach is needed due to the necessity of comprehending the context and word class. This might entail incorporating an external natural language processing library or engaging a related service, ensuring a more accurate and satisfying experience to you as a proud reader of this book.

Information Retrieval with TF-IDF

TF-IDF, which stands for Term Frequency-Inverse Document Frequency, is a numerical statistic used in natural language processing and information retrieval. It is a technique that assigns a weight to each word in a document, based on its importance in the document and within the entire corpus. The importance of a word increases with its frequency in a document but is offset by its frequency across all documents.

TF-IDF Components

TF-IDF has two components:
Term Frequency (TF): This measures the frequency of a word in a document. It can be calculated as the number of times a word appears in a document divided by the total number of words in the document.
Inverse Document Frequency (IDF): This measures the importance of a word across the entire corpus. It can be calculated as the logarithm of the total number of documents divided by the number of documents containing the word.

The product of TF and IDF gives the TF-IDF score for each word in a document.

Implementation of TF-IDF

Discover a straightforward implementation of the Term Frequency-Inverse Document Frequency (TF-IDF) algorithm within the Rust programming language through this concise example:

Create a function to calculate the term frequency:

```
fn term_frequency(word: &str, document: &str) -> f64 {

  let words: Vec<&str> = document.split_whitespace().collect();



   let word_count = words.iter().filter(|&w| w == &word).count() as f64;

   let total_words = words.len() as f64;

  word_count / total_words

}
```

Create a function to calculate the inverse document frequency:

```
fn inverse_document_frequency(word: &str, documents: &[String]) -> f64
{

   let num_documents = documents.len() as f64;

  let num_documents_with_word = documents

      .iter()
```

```
        .filter(|doc| doc.contains(word))
        .count() as f64;
    (num_documents / (1.0 + num_documents_with_word)).ln()
}
```

Create a function to calculate the TF-IDF score for each word in a document:

```
fn tf_idf(word: &str, document: &str, documents: &[String]) -> f64 {
    let tf = term_frequency(word, document);
    let idf = inverse_document_frequency(word, documents);
    tf * idf
}
```

Use the tf_idf function to calculate the TF-IDF scores for the words in your documents:

```
let documents = vec![
    "the quick brown fox jumps over the lazy dog",
    "the quick red fox jumps over the lazy dog",

    "the quick orange fox jumps over the lazy dog",

];

let document = "the quick brown fox jumps over the lazy dog";

let words: Vec<&str> = document.split_whitespace().collect();

for word in words {

    let score = tf_idf(word, document, &documents);

    println!("TF-IDF score for {}: {}", word, score);

}
```

This code will output the TF-IDF scores for each word in the document.

Word Embeddings and Word2Vec

Word embeddings are dense vector representations of words that capture their semantic meaning. They allow algorithms to work with text data by converting it into numerical format. One popular word embedding technique is Word2Vec, which generates embeddings by training a shallow neural network on a large corpus. We can use the finalfusion crate to read pre-trained word embeddings and use them in your Rust project.

To begin with , first you'll need to add finalfusion to your Cargo.toml:

```
[dependencies]

finalfusion = "0.16.0"
```

Then, download pre-trained word embeddings in the finalfusion format. You can find pre-trained embeddings in different formats on various websites, like the GloVe or fastText websites. After downloading, convert them to the finalfusion format using the finalfusion Python library or another converter.

Now, you can use the pre-trained embeddings in your Rust project:

```
use finalfusion::prelude::*;

use std::fs::File;
use std::io::BufReader;

fn main() {

   let path = "path/to/your/embeddings.fifu";

   let f = File::open(path).unwrap();

   let reader = BufReader::new(f);

   let embeddings = Embeddings::read_embeddings(reader).unwrap();



    let example_sentence = "the quick brown fox jumps over the lazy dog";

   let words: Vec<&str> = example_sentence.split_whitespace().collect();

   for word in words {

       match embeddings.embedding(word) {

           Some(embedding) => {

               println!("Embedding for {}: {:?}", word, embedding);

           }
```

```
                None => {
                    println!("Embedding not found for {}", word);
                }
            }
        }
    }
```

In the above sample program, we use the finalfusion crate to read the pre-trained embeddings and print the embeddings for the words in the example_sentence. Note that we're using a sample sentence here; you'll need to read and preprocess the text from the given database and to work directly with the given database, you'll need to read the CSV file, preprocess the text, and then apply the word embeddings to the preprocessed text as explained in many of the previous chapters. You can use the csv crate to read the CSV file, but keep in mind that you'll need to apply the text preprocessing steps discussed earlier in this book, such as tokenization, stopword removal, and stemming or lemmatization.

Summary

Overall, this chapter delves into the world of Text and Natural Language Processing (NLP). It starts by emphasizing the growing importance and widespread adoption of NLP in various domains, and the key processes involved in NLP are explained conceptually. NLP techniques enable machines to understand, interpret, and generate human language in a way that is both meaningful and useful. The chapter then focuses on various text preprocessing steps, which play a crucial role in preparing the raw text data for further analysis. Tokenization is introduced as the process of breaking down the text into individual words or tokens. This step enables the conversion of unstructured text data into a more structured format that can be analyzed more effectively.

Next, stopword removal is discussed. Stopwords are common words such as "and," "the," and "is" that do not carry significant meaning and can be removed to reduce noise in the data and improve the efficiency of text analysis. By eliminating these uninformative words, the relevant information can be more easily processed and analyzed. Stemming and lemmatization are also covered in the chapter. These techniques are employed to reduce words to their base or root form, thereby further reducing noise and enabling the grouping of similar words. Stemming typically involves removing word suffixes, while lemmatization is a more advanced process that takes into account the word's context and morphological structure. The chapter then moves on to the concept of Term Frequency-Inverse Document Frequency (TF-IDF), which is used to weigh the importance of words in a document or corpus. The mathematical architecture of TF-IDF is explained, followed by a practical demonstration of its implementation on the given database. Finally, the chapter addresses word embeddings, which are dense vector representations that capture the semantic meaning of words. The example demonstrates how to use pre-trained word embeddings in Rust with the finalfusion crate, illustrating the process with a sample sentence. Note that in practice, you would need to read and preprocess the text from the given database before applying the word embeddings.

To sum it up, Chapter 11 provides an overview of key NLP concepts and techniques, focusing on text preprocessing, feature extraction, and word embeddings. The practical demonstrations help readers understand how to implement these techniques in Rust using real-world data.

Epilogue

As we conclude our journey through "Statistics with Rust" it is important to take a moment to reflect on the wealth of knowledge and practical skills acquired throughout the course of this book. Designed for statisticians and data professionals, this comprehensive resource has demonstrated the advantages of Rust in modern statistical methods and provided practical examples of Rust's potential in various aspects of data analysis and machine learning.

Throughout the 11 in-depth chapters, we have explored the Rust programming environment, essential libraries for data professionals, and various statistical methods and techniques. From data handling, preprocessing, and visualization to advanced topics like machine learning, natural language processing, and network analysis, this book has offered a thorough understanding of Rust-based data analysis.

The field of statistics is ever-evolving, with new methodologies, techniques, and tools emerging constantly. By mastering Rust and its application in modern statistical methods, you have positioned yourself at the forefront of this exciting discipline. As you continue to work on more complex projects, the knowledge and skills you have gained will enable you to tackle new challenges with confidence, efficiency, and effectiveness.

To stay ahead in this fast-paced field, it is crucial to remain updated with the latest advancements and to continually refine your skills. The Rust community is continuously expanding, with developers and researchers contributing to new libraries and frameworks tailored to statistics and data analysis. Engaging with the community will not only help you stay informed about the latest developments but also provide opportunities to collaborate, learn from others, and contribute to the growth of Rust in the realm of statistics.

As you progress in your career, consider sharing your expertise and insights with the Rust and statistics communities. Open-source projects, blog posts, talks, and workshops are excellent ways to give back to the community, inspire others, and foster the growth of Rust in statistical applications.

Overall, "Statistics with Rust" has provided you with the tools, techniques, and knowledge required to excel in Rust-based statistical analysis. However, this is not the end of your journey; it is merely the beginning. As you continue to grow and evolve as a data professional, never stop learning, exploring new ideas, and pushing the boundaries of what is possible in statistics.

We hope this book has inspired you to embrace the power of Rust in your data projects and to continue innovating, creating, and excelling in the field of statistics. The future of statistics with Rust is bright, and we eagerly anticipate the remarkable contributions you will undoubtedly make in this domain.

Thank you for choosing "Statistics with Rust" as your companion in this exciting journey. We wish you the best of luck as you continue to revolutionize your data projects with Rust and make a lasting impact in the world of statistics.

Thank You