Data Science at the Linux Terminal with Nushell

Luis Gómez Guzmán

November 10, 2026

Contents

1 Introduction **11**

2 Software Installation **13**
2.1 Windows 10+ . . . 13
2.1.1 Installing Ubuntu on WSL . . . 13
2.1.2 Installing Docker Desktop . . . 13
2.1.3 Launching the Linux Environment . . . 13
2.2 Linux, WSL2, and the `dsin` Container (Data Science in Nushell) . . . 14
2.2.1 Installing Docker on Linux / WSL (Debian/Ubuntu) . . . 14
2.2.2 Installing Docker on Linux (Arch Linux / CachyOS / Manjaro) . . . 14
2.2.3 Pulling and Running the `dsin` Container . . . 14
2.2.4 Defining a Shell Shortcut (Optional) . . . 15

3 Nushell Fundamentals and Data Structures **16**
3.1 Data Science . . . 16
3.1.1 Obtaining Data . . . 17
3.1.2 Scrubbing Data . . . 17
3.1.3 Exploring Data . . . 17
3.1.4 Modeling Data . . . 17
3.1.5 Interpreting Data . . . 18
3.2 History of the Terminal . . . 18
3.2.1 The Shell . . . 19
3.2.2 Advantages of the Command Line for Data Science . . . 19
3.3 The Terminal . . . 20
3.3.1 The Interactive Environment (REPL) . . . 20
3.3.2 Running Commands and Saving Output to Files . . . 20
3.3.3 Command-Line Tool Categories . . . 21
3.3.4 Traditional Shells (Bash and Zsh): Architecture, Philosophy, and Limitations 22
3.3.4.1 The Role of Traditional Shells . . . 22

3.3.4.2 The Unix Philosophy and Plain-Text Streams . . . 22
3.3.4.3 Limitations of Plain-Text Streams for Data Science . . . 22
3.3.4.4 The Nushell Paradigm: Structured Data and Typed Pipelines . . . 23
3.4 Bashisms and Productivity Shortcuts in Nushell . . . 24
3.4.1 Bang-Bang History Expansion (!) . . . 24
3.4.2 External Process Control and Sequencing . . . 24
3.4.3 The Nushell Environment Is Scoped . . . 24
3.5 Basic Elements of Nushell . . . 25
3.5.1 Getting Help . . . 27
3.6 Data Types and Variables in Nushell . . . 28
3.6.1 The Nushell Type System . . . 28
3.6.2 Nushell as a Compiled Language . . . 30
3.6.3 Variable Declarations: let, mut, and const . . . 31
3.6.3.1 Naming and Referencing Rules . . . 31
3.6.3.2 Runtime Immutability: let . . . 31
3.6.3.3 Controlled Imperative Mutability: mut . . . 32
3.6.3.4 Parse-Time Constants: const . . . 33
3.6.3.5 Functional Pipelines versus Imperative Mutation . . . 34
3.6.3.5.1 Performance Implications . . . 35
3.7 Navigating the Filesystem . . . 36
3.7.1 Inspecting Directory Contents . . . 36
3.7.2 Glob Patterns (Wildcards) . . . 36
3.7.2.1 Programmatic Glob Generation . . . 37
3.8 Working with Strings . . . 38
3.8.1 String Formats . . . 38
3.8.2 Single-Quoted Strings . . . 38
3.8.3 Double-Quoted Strings . . . 38
3.8.4 Raw Strings . . . 39
3.8.5 Bare-Word Strings . . . 39
3.8.6 Backtick-Quoted Strings . . . 40
3.8.7 Executing Strings as External Commands . . . 40
3.8.8 String Concatenation and Joins . . . 40
3.8.9 String Interpolation . . . 41
3.8.10 Splitting Strings . . . 41
3.8.11 String Transformation Commands (str) . . . 42
3.8.11.1 Trimming . . . 42
3.8.11.2 Substrings and Indexing . . . 42
3.8.11.3 Padding . . . 42
3.8.11.4 Reversing . . . 43
3.8.11.5 Structured Parsing . . . 43
3.8.12 String Comparisons . . . 43
3.8.13 Type Conversions . . . 43
3.8.14 ANSI Terminal Colors . . . 43

3.9 Working with Lists . . . 44
3.9.1 Creating Lists . . . 44
3.9.2 Modifying Lists . . . 44
3.9.3 Appending, Prepending, and Slicing . . . 44
3.9.3.1 Spread Operator (...) . . . 45
3.9.4 Iterating over Lists . . . 45
3.9.5 Accessing List Elements . . . 46
3.9.6 Transforming Lists . . . 48
3.10 Working with Records . . . 49
3.10.1 Creating Records . . . 49
3.10.2 Updating Records . . . 50
3.10.3 Iterating and Transforming Records . . . 50
3.10.3.1 Transposing Records for Sequence Processing . . . 50
3.10.3.2 Native Record Commands . . . 51
3.10.3.2.1 Filtering Fields with record where . . . 51
3.10.3.2.2 Transforming Keys and Values with record each . . . 51
3.10.3.2.3 Recursive Processing with record walk . . . 52
3.10.3.2.4 Declarative Field Conversions with record apply . . . 52
3.10.3.2.5 Row Streaming Across Tables . . . 53
3.11 Working with Tables . . . 53
3.11.1 Sorting Data . . . 53
3.11.2 Selecting Data . . . 54
3.11.3 Retrieving Data from a Table . . . 54
3.11.4 Modifying Data in a Table . . . 54
3.11.4.1 Concatenating Tables . . . 55
3.11.4.2 Merging Tables . . . 55
3.11.5 Adding a Column . . . 56
3.11.6 Updating a Column . . . 57
3.11.7 Moving Columns . . . 57
3.11.8 Renaming Columns . . . 57
3.11.9 Rejecting and Removing Columns . . . 57
3.12 Navigating and Accessing Structured Data . . . 57
3.12.1 Foundational Concepts . . . 57
3.12.2 Cell Paths . . . 58
3.12.2.1 Records . . . 58
3.12.2.2 Lists . . . 58
3.12.2.3 Tables . . . 59
3.12.2.3.1 Example: Accessing a Table Row (Record) . . . 59
3.12.2.3.2 Example: Accessing a Table Column (List) . . . 60
3.12.2.3.3 Example: Accessing a Table Cell (Value) . . . 60
3.12.3 Nested Data . . . 60
3.12.3.1 Example: Accessing Nested Table Data . . . 60
3.12.4 Using `get` and `select` . . . 60

3.12.4.1 Example: `get` versus `select` on a Table Row . . . 61
3.12.4.2 Example: `select` with Multiple Rows and Columns . . . 61
3.12.5 Keys and Column Names with Spaces . . . 61
3.12.6 Handling Missing Data . . . 62
3.12.6.1 The Optional Operator . . . 62
3.12.6.1.1 Example: The Optional Operator . . . 62
3.12.6.2 Supplying a Default Value for Missing or Null Data . . . 62
3.12.7 Other Commands for Accessing Structured Data . . . 63
3.13 In-Terminal File Editing with `ttt` . . . 63
3.13.1 Basic Usage of `ttt` . . . 63
3.14 Version Control . . . 64
3.14.1 Git: The Essential Tool . . . 64
3.14.2 Cloning and Installing CLI Tools: An Example with ddgr . . . 65
3.14.2.1 1. Interactive Terminal Mode . . . 66
3.14.2.2 2. Structured Output and Pipeline Integration in Nushell . . . 66
3.14.3 GitHub and GitLab . . . 66
3.14.4 Best Practices for Data Science Repositories . . . 67
3.14.5 Commit and Push . . . 67
3.14.6 Contributing to a Repository . . . 68
3.14.7 Guidelines for Commit Messages . . . 69
3.14.8 Additional Resources . . . 69
3.14.9 For Further Exploration . . . 69

4 Programming and Creating Command-Line Tools 70
4.1 Programming in Nushell . . . 70
4.1.1 Operators . . . 70
4.1.1.1 Order of Operations . . . 71
4.1.1.2 Types . . . 72
4.1.1.3 Regular Expression and Substring Matching Operators . . . 72
4.1.1.4 Case Sensitivity . . . 73
4.1.1.5 The Spread Operator . . . 73
4.1.1.5.1 In List Literals . . . 73
4.1.1.5.2 In Record Literals . . . 74
4.1.1.5.3 In Command Invocations . . . 75
4.1.2 Variables in Scripts and CLI Tools . . . 76
4.1.3 Custom Commands . . . 76
4.1.4 Command Names . . . 77
4.1.5 Subcommands . . . 77
4.1.6 Parameter Types . . . 77
4.1.7 Parameters with Default Values . . . 78
4.1.8 Optional Positional Parameters . . . 78
4.1.9 Flags . . . 79
4.1.10 Rest Parameters . . . 81

4.1.11 Documenting a Command . . . 82
4.1.12 Closures . . . 84
4.1.12.1 Syntax and Parameters . . . 84
4.1.12.2 Dynamic Invocation with `do` . . . 84
4.1.12.3 Higher-Order Functions in Pipelines . . . 84
4.1.12.4 Lexical Scoping and Immutability . . . 85
4.1.13 Pipeline Output . . . 85
4.1.14 Pipeline Input . . . 85
4.1.15 Control Flow . . . 86
4.1.15.1 Conditionals . . . 86
4.1.15.1.1 `if` . . . 86
4.1.15.1.2 `match` . . . 87
4.1.15.2 Loops . . . 88
4.1.15.2.1 Loops versus Functional Iterators . . . 88
4.1.15.2.2 `for` . . . 90
4.1.15.2.3 `while` . . . 90
4.1.15.2.4 `loop` . . . 91
4.1.15.2.5 `break` . . . 91
4.1.15.2.6 `continue` . . . 91
4.1.15.2.7 Errors . . . 91
4.1.15.2.8 Other Statements . . . 92
4.2 Converting One-Liners into Command-Line Tools . . . 93
4.2.1 Step 1: Create the File . . . 94
4.2.2 Step 2: Grant Execution Permissions . . . 95
4.2.3 Step 3: Define the Shebang . . . 96
4.2.4 Step 4: Remove Fixed Input . . . 96
4.2.5 Step 5: Add Typed Arguments and Options (`def main`) . . . 97
4.2.6 Step 6: Extend `$env.PATH` . . . 98
4.3 Creating Command-Line Tools with Python and R . . . 98
4.3.1 Porting the Tool to Python and R . . . 98
4.3.2 Streaming Data from `stdin` . . . 99

5 Data Acquisition **100**
5.1 Copying Local Files into the Docker Container . . . 100
5.2 Downloading Data from the Internet with `http get` . . . 100
5.2.1 Fetching JSON Endpoints and Web Pages . . . 100
5.2.2 Fetching from Multiple URLs and Dispatching `http post` Requests . . . 101
5.2.3 Following Redirects and Inspecting HTTP Headers . . . 101
5.3 Decompressing and Extracting Archives . . . 102
5.4 Working with Spreadsheets (Microsoft Excel) and CSV Files . . . 103
5.5 Querying Relational Databases (SQLite) . . . 104
5.6 Consuming Web APIs . . . 104
5.6.1 API Key Authentication and URL Construction . . . 105

5.6.2 Geocoding with the Google Maps API . . . 105
5.6.2.1 Obtaining an API Access Key . . . 105
5.6.2.2 Anatomy of the Request and url join . . . 106
5.6.2.3 Implementing the maps.nu Module . . . 107
5.6.3 Automated Translation with Google Translate . . . 108
5.6.3.1 Obtaining an API Access Key . . . 108
5.6.3.2 Implementing the translate.nu Module . . . 108
5.6.4 Structured Web Search and Ingestion with the Ollama Search API . . . 110
5.6.4.1 Obtaining an Access Token (API Key) . . . 110
5.6.4.2 Implementing the ollama_search.nu Module . . . 110
5.6.5 Real-Time Streaming APIs . . . 112

6 Data Cleaning and Transformation 113
6.1 Transformations in a Data Pipeline . . . 113
6.2 Plain Text . . . 113
6.2.1 Filtering Lines . . . 114
6.2.1.1 Pattern-Based Filtering . . . 114
6.2.1.2 Random Sampling . . . 115
6.2.2 Extracting Values . . . 115
6.2.3 Replacing and Stripping Values . . . 116
6.3 Parsing Command Output . . . 116
6.4 CSV and Structured Tables . . . 117
6.4.1 Headers, Bodies, and Columns . . . 117
6.4.2 Executing SQL Queries on CSV . . . 117
6.4.3 Extracting and Reordering Columns . . . 118
6.4.4 Filtering Rows . . . 118
6.4.5 Combining Columns and Quoted Delimiter Robustness . . . 118
6.4.6 Combining Multiple CSV Files . . . 118
6.4.6.1 Horizontal Merging (`merge`) . . . 118
6.4.6.2 Relational Joins (`join`) . . . 119
6.5 Working with XML/HTML and JSON . . . 119
6.6 The Nushell Polars Plugin . . . 120
6.6.1 Use Cases . . . 120
6.6.2 Foundational Concepts . . . 120
6.6.2.1 Polars DataFrames Versus Nushell Tables . . . 120
6.6.3 Basic Usage . . . 120
6.6.3.1 Data Ingestion . . . 120
6.6.3.2 Data Exploration . . . 121
6.6.3.3 Type Casting and Schema Enforcement . . . 121
6.6.4 Data Manipulation . . . 122
6.6.4.1 Column Selection . . . 122
6.6.4.2 Row Filtering . . . 123
6.6.4.3 Sorting Data . . . 123

6.6.4.4 Grouping and Aggregation . . . 123
6.6.5 File Formats and Persistence . . . 125
6.6.6 Column Pivoting . . . 125
6.6.7 Best Practices and Recommendations . . . 126
6.6.8 Summary . . . 127

7 Exploratory Data Analysis and Visualization 127
7.1 Inspecting Data and Its Properties . . . 127
7.1.1 Inspecting Data Types and Schema Inference . . . 128
7.1.2 Identifying Unique Values and Removing Invariant Columns . . . 128
7.2 Computing Descriptive Statistics . . . 129
7.2.1 Summary Statistics with `math` and `polars summary` in Nushell . . . 129
7.2.2 Transformations and Statistical Summaries via R and Python Scripts . . . 129
7.3 Creating Statistical Visualizations from the Command Line . . . 130
7.3.1 Viewing Images Directly in the Terminal with `timg` . . . 130
7.3.2 Remote Alternatives: Copying Graphics via `scp` . . . 131
7.4 Catalog of Statistical Graphics via R and Python Scripts . . . 132
7.4.1 Bar Charts . . . 132
7.4.2 Histograms . . . 132
7.4.3 Density Plots . . . 132
7.4.4 Line Charts for Time Series . . . 133
7.4.5 Scatter Plots and Trend Curves . . . 134
7.4.6 Box Plots and Violin Plots . . . 134
7.4.7 Equivalent Statistical Visualization in Python . . . 135
7.5 Summary . . . 136

8 Serial and Parallel Data Pipelines 137
8.1 Serial Processing: Iterating Over Data with `each` . . . 139
8.1.1 Iterating Over Lines and Rows . . . 139
8.1.2 Iterating Over Files . . . 139
8.2 Parallel Processing in Nushell with `par-each` . . . 139
8.2.1 Concurrent Execution and Thread Control . . . 141
8.3 Process Parallelization with GNU `parallel` from Nushell . . . 141
8.3.1 Specifying Arguments and Replacement Tokens . . . 141
8.3.2 Controlling Concurrent Jobs . . . 142
8.3.3 Output Logging and Tagging . . . 142
8.3.4 Parallel Tools and Block Processing . . . 143
8.4 Distributed Computing Across Multiple Nodes . . . 143
8.4.1 Discovering Active Cluster Nodes . . . 143
8.4.2 Running Commands Across All Cluster Nodes . . . 143
8.4.3 Distributing Local Data Streams Across Cluster Nodes . . . 143
8.4.4 Processing Files on Remote Nodes . . . 144
8.5 Summary . . . 144

9 Polyglot Data Modeling **145**
9.1 Data Preparation and Exploration of the Wine Dataset . . . 145
9.2 Dimensionality Reduction (PCA and t-SNE) . . . 148
9.2.1 Dimensionality Reduction with CLI Tools . . . 148
9.2.2 Dimensionality Reduction with `Rscript` . . . 152
9.2.3 Dimensionality Reduction with Python . . . 153
9.2.4 Multicollinearity Diagnostics (VIF) and Principal Component Regression (PCR) . . . 154
9.3 Clustering . . . 157
9.3.1 K-Means Clustering . . . 157
9.3.2 K-Means Diagnostics in R and Python . . . 161
9.3.3 Agglomerative Hierarchical Clustering . . . 163
9.3.4 Multi-Metric Validation and Visual Elbow Diagnostics . . . 165
9.3.5 Density-Based Clustering with DBSCAN and K-Medoids with PAM . . . 169
9.4 Regression: Predicting Wine Quality . . . 171
9.4.1 Online Learning with Vowpal Wabbit . . . 171
9.4.1.1 Direct Terminal Pre-Processing, Execution, and Post-Processing . 172
9.4.1.2 Encapsulating the Workflow in `to-vw` and `vw-regress` . . . 174
9.4.1.3 Comparative Rerun with `to-vw` and `vw-regress` . . . 176
9.4.2 Multiple Linear Regression with `Rscript` . . . 177
9.4.3 Penalized Ridge and Ensemble Regression with Python . . . 178
9.5 Supervised Classification . . . 180
9.5.1 Classification Experiments with SKLL . . . 180
9.5.1.1 Declarative Experimentation with SciKit-Learn Laboratory . . . 180
9.5.1.2 Direct CLI Execution and Nushell TSV Post-Processing . . . 182
9.5.1.3 Encapsulating TSV Projection in `skll-metrics` . . . 182
9.5.1.4 Comparative Rerun with `skll-metrics` . . . 183
9.5.2 Binomial Logistic Regression with `Rscript` . . . 183
9.5.3 Logistic and Random Forest Classification in Python . . . 185
9.5.4 Decision Trees on Imbalanced Banking Data . . . 186
9.5.5 k-Nearest Neighbors Classification . . . 189
9.5.6 Generative Naïve Bayes vs. RBF Kernel SVMs . . . 192
9.5.7 Feedforward Neural Networks: Multi-Layer Perceptron . . . 193
9.6 Modern Pipeline-Native Modeling . . . 196
9.6.1 Topological Manifold Projection with UMAP . . . 197
9.6.1.1 Inspecting and Running `umap_reduce.py` Directly . . . 197
9.6.1.2 Encapsulating the Pipeline in `umap-reduce` . . . 199
9.6.1.3 Comparative Rerun and Visual Manifold Comparison . . . 200
9.6.2 Gradient Boosted Decision Trees from the CLI with LightGBM . . . 202
9.6.2.1 Direct CLI Pre-Processing, Training, Prediction, and Feature Attribution . . . 203
9.6.2.2 Encapsulating Training and Importance in `lgbm-regress` . . . 206
9.6.2.3 Comparative Rerun and Visual Feature-Gain Diagnostics . . . 208

9.6.3 Vectorized In-Process Analytical Regression with DuckDB . . . 211
9.6.3.1 Direct CLI SQL Querying and Nushell JSON Ingestion . . . 211
9.6.3.2 Encapsulating Schema-Driven SQL Generation in `duckdb-ols` . . . 212
9.6.3.3 Comparative Rerun and Multivariate Residual Diagnostics . . . 213
9.6.4 Streaming Multi-Model Classification and Neural Networks . . . 215
9.6.4.1 Inspecting and Invoking `ml_benchmark.py` Directly . . . 216
9.6.4.2 Encapsulating Multi-Model Benchmarking in `ml-benchmark` . . . 220
9.6.4.3 Comparative Rerun and Visual Accuracy-Latency Trade-Offs . . . 221
9.6.5 The Pure-Rust Machine Learning Frontier . . . 225
9.6.5.1 Supervised Classification with Random Forests . . . 226
9.6.5.2 Unsupervised Clustering and Dimensionality Reduction . . . 226
9.6.5.3 Tensor Inference and Neural Networks . . . 227
9.6.5.4 Deep Learning Neural Regression on CPU . . . 227
9.6.5.5 In-Memory Tensor Linear Algebra . . . 227
9.7 Chapter Summary . . . 228

10 Conclusion: The Command Line as a Data Intelligence Laboratory 230
10.1 The OSEMN Lifecycle in Typed Shells . . . 230
10.2 The Intersection of Software Engineering and Statistics . . . 231
10.3 Computational Efficiency, Out-of-Core Limits, and the Zero-Copy Arrow Frontier . . . 232
10.4 Methodological Guide for Applied Production Projects . . . 233

11 For Further Exploration 235
11.1 The Command Line . . . 235
11.2 Shell Programming . . . 235
11.3 Python, R, and SQL . . . 235
11.4 APIs . . . 235
11.5 Machine Learning . . . 235

12 References 236
12.1 Nushell Ecosystem . . . 236
12.2 Data Science at the Command Line (DSitCL) . . . 236
12.3 Command-Line Tools Integrated into `dsin` . . . 236
12.4 Academic Bibliography and Literature . . . 237

List of Figures

1 The data science lifecycle (OSEMN framework). . . . 16
2 Early mainframe computer systems. . . . 18
3 Electromechanical teleprinters (teletypewriter or tty). . . . 19
4 Classic video display terminals (DEC VT100). . . . 20
5 Video terminal cabled to a PDP-11 minicomputer. . . . 21
6 Standard output redirection in the terminal . . . 23

7 Faceted scatter plot of total bills and tips across days of the week . . . 131
8 Composite terminal gallery of histogram and kernel density estimation . . . 132
9 Bar chart generated with ggplot2 . . . 133
10 Histogram of total bill values . . . 134
11 Kernel density estimation . . . 135
12 Time series of traffic accidents . . . 136
13 Scatter plot with LOESS trend curve . . . 137
14 Box plot of bills by day . . . 138
15 Dual-panel empirical distribution and scatter plot generated via Python matplotlib 138
16 Conceptual comparison between sequential and parallel execution . . . 140
17 Execution flow of GNU parallel distributing arguments across multiple cores . . . 141
18 Interleaved outputs when multiple processes write concurrently . . . 142
19 Class-conditional density of sensory quality across red and white wines . . . 147
20 Bidimensional projections obtained via dimensionality reduction . . . 151
21 Dual-panel K-Means cluster validation showing within-cluster inertia and mean silhouette score across k = 2 to 10 on wine-both.csv . . . 169
22 Comparison of bidimensional projections across PCA, t-SNE, and UMAP on wine attributes . . . 203
23 LightGBM feature split-gain percentage and split-count diagnostics on white wine predictors . . . 211
24 Multivariate OLS residual diagnostics and Gaussian density overlay on white wine sensory quality . . . 216
25 Comparative weighted F1, accuracy, and training latency across five classification architectures on banking data . . . 225

List of Tables

1 Fundamental data types in Nushell. . . . 29
2 Assignment operators for mutable variables in Nushell. . . . 32
3 Variable binding models in Nushell: `let`, `mut`, and `const`. . . . 34

1 Introduction

In modern data science practice, the sheer scale, heterogeneity, and volatility of incoming information streams demand tools that move beyond rigid graphical interfaces and the computational overhead of monolithic notebook platforms. For over half a century, the Unix command-line console has served as the gold standard for scientific reproducibility, unattended automation, and algorithmic efficiency. Its founding philosophy remains remarkably pertinent: build dependable systems out of small, focused programs designed to handle one concrete task with precision, and connect them effortlessly across standardized communication channels.

Traditional shells like Bash and Zsh, however, were conceived in an era when text in transit was treated strictly as unstructured character streams. Under this legacy model, manipulating data matrices, multi-tiered tabular structures, or nested records requires fragile chains of regular expressions and text-slicing utilities such as `sed`, `awk`, and `cut`. This paradigm introduces persistent friction. It breaks when delimiters shift unexpectedly, chokes on escape sequences, and strips variables of their fundamental data types.

Nushell (`nu`) resolves this architectural bottleneck by redesigning the shell around the demands of contemporary data analysis. Rather than piping opaque byte streams from command to command, Nushell pipelines transport typed, structured data: two-dimensional tables with explicit schemas, key-value records, ordered lists, and primitive types governed by strict semantics. Because Nushell natively understands integers, floating-point numbers, time zones, file paths, and memory sizes, you can filter, sort, transform, and aggregate datasets using a clean, declarative syntax.

This expressive clarity is paired with high computational throughput. Nushell integrates directly with Polars, a blazingly fast columnar engine written in Rust and powered by Apache Arrow. This pairing brings industrial-grade analytical capabilities directly into your shell. Multithreaded analytical queries across millions of rows execute in fractions of a second, without sacrificing the interactive immediacy of the command line.

This textbook builds upon the foundational methodology introduced by Jeroen Janssens in *Data Science at the Command Line* (O'Reilly Media). Janssens demonstrated that the command line provides a nimble, versatile environment for practical data science. Yet command-line tooling has evolved substantially since that work was published.

Where the original text relied on plain-text stream processors and auxiliary suites like `csvkit`, this book adopts Nushell and its structured data model. This approach preserves the core Unix virtues of composition and modularity while providing typed tables, clear schemas, and robust data handling.

This book is designed as a hands-on laboratory, meant to be read with a terminal running side by side. For this reason, most code examples omit command output (with the exception of terminal plots, figures, and essential schema demonstrations). We expect you to execute each pipeline, inspect intermediate tables, and explore Nushell interactively.

Omitting printed outputs serves three concrete pedagogical goals: first, it encourages active learn-

ing through direct execution rather than passive reading; second, it leverages interactive shell features like `explore`, live table scrolling, and autocompletion, which are best appreciated first-hand; and third, it develops the habit of inspecting data structures with `describe` and diagnosing issues independently. We encourage you to tweak parameters, test alternative filters, and examine the resulting tables in your own shell.

To ensure that every pipeline runs in a strictly reproducible environment unaffected by local host quirks, this text relies on a unified container image named `dsin` (Data Science in Nushell), available publicly on GitHub Container Registry at `ghcr.io/kurokirasama/dsin:latest`. This container serves as a self-contained data science workstation. It bundles the latest Nushell release configured for analytical work, its official plugin suite (including native Polars integration, structured format parsers, relational database querying, Git inspection, and terminal plotting), complete Python and R scientific stacks, high-performance modeling engines (Vowpal Wabbit, Tapkee, and SKLL), terminal utilities such as the `ttt` text editor and `timg` image viewer, and the complete set of course datasets neatly organized by chapter inside `/data`.

The curriculum follows the classic OSEMN lifecycle (*Obtain*, *Scrub*, *Explore*, *Model*, and *iNterpret*), originally formulated by Hilary Mason and Chris Wiggins. Our progression begins in Chapter 2 with software setup and container preparation using Docker and `dsin`. Chapter 3 establishes the operational fundamentals of the terminal alongside Nushell's data types, while Chapter 4 covers structured scripting, closures, and building modular command-line tools. Chapter 5 examines data ingestion from web APIs, compressed archives, Excel spreadsheets, and SQLite databases. Chapter 6 explores data scrubbing, restructuring, and columnar analytics with Polars. Statistical data exploration and in-terminal plotting take center stage in Chapter 7, followed by pipeline orchestration, serial iteration, and multi-core parallelization in Chapter 8. Chapter 9 develops polyglot statistical and machine learning workflows across dimensionality reduction, unsupervised clustering, regression, and supervised classification. Finally, Chapter 10 synthesizes the architectural, computational, and methodological principles that turn the command line into a rigorous data intelligence laboratory.

2 Software Installation

This section provides instructions for installing the required software environment across Windows 10+ and Linux (including Debian/Ubuntu and Arch-based distributions).

2.1 Windows 10+

2.1.1 Installing Ubuntu on WSL

1. Enable WSL1 by executing in an elevated PowerShell session (Run as Administrator):

```
Enable-WindowsOptionalFeature -Online -FeatureName Microsoft-Windows-Subsystem-Linux
```

2. Enable the Virtual Machine Platform (VMP) in PowerShell:

```
Enable-WindowsOptionalFeature -Online -FeatureName VirtualMachinePlatform
```

3. List available distributions (Ubuntu 24.04 is recommended):

```
wsl --list --online
```

4. Install the chosen distribution:

```
wsl --install -d Ubuntu-24.04
```

5. Set the installed distribution to use WSL version 2 in PowerShell:

```
wsl --set-version Ubuntu-24.04 2
```

To configure WSL 2 as the default for all future distributions:

```
wsl --set-default-version 2
```

6. Update the WSL kernel by downloading the installer package from the official WSL repository.

2.1.2 Installing Docker Desktop

1. Download the Docker Desktop installer for Windows from Docker documentation and launch it. Ensure the option "Use WSL 2 instead of Hyper-V" remains checked.
2. Enable "Start Docker Desktop when you log in" in Docker settings.
3. Restart your computer.

2.1.3 Launching the Linux Environment

To enter the installed distribution, run in `cmd` or `powershell`:

```
wsl
```

2.2 Linux, WSL2, and the `dsin` Container (Data Science in Nushell)

Throughout this book, we will work inside an integrated container image named **`dsin`** (*Data Science in Nushell*), published at `ghcr.io/kurokirasama/dsin:latest`. Unlike conventional container environments built around plain-text shells, `dsin` launches directly into **Nushell** as its default login shell inside `/data`, preconfigured with:

- **Nushell (`nu`)** and its full suite of registered official plugins: **`nu_plugin_polars`** (Polars and Apache Arrow DataFrame engine), **`nu_plugin_formats`**, **`nu_plugin_gstat`**, **`nu_plugin_inc`**, **`nu_plugin_query`**, **`nu_plugin_file`**, **`nu_plugin_plot`** (terminal plotting), and **`nu_plugin_port_extension`**.
- **Integrated `/data` directory:** Contains the complete chapter data sets organized by chapter number (`/data/ch03`, `/data/ch04`, `/data/ch05`, `/data/ch06`, `/data/ch07`, `/data/ch08`, and `/data/ch09`), eliminating the need for manual downloads.
- **Terminal text editor `ttt`** available on `.PATH` with preconfigured syntax highlighting and keybindings.
- **Terminal image viewer `timg`** on `.PATH` for displaying figures, plots, and graphical output directly in your console.
- **Polyglot ecosystem:** R (`Rscript`, `tidyverse`, `ggplot2`, `rpart`, `class`, `cluster`), Python 3 (`pandas`, `scikit-learn`, `scipy`, `matplotlib`, `seaborn`, `skll`), **GNU Parallel (`parallel`)**, **Tapkee (`tapkee`)**, and **Vowpal Wabbit (`vw`)**.
- **Book modeling module (`dsin_tools.nu`):** Custom Nushell commands (`to-vw`, `vw-regress`, `tapkee-reduce`, `kmeans-cluster`) pre-loaded at shell startup.

2.2.1 Installing Docker on Linux / WSL (Debian/Ubuntu)

```
sudo apt update && sudo apt upgrade -y
sudo apt install -y ca-certificates curl gnupg lsb-release git
sudo usermod -aG docker $USER
```

2.2.2 Installing Docker on Linux (Arch Linux / CachyOS / Manjaro)

```
sudo pacman -Syu --needed docker docker-buildx docker-compose git
sudo systemctl enable --now docker.service
sudo usermod -aG docker $USER
```

2.2.3 Pulling and Running the `dsin` Container

To pull the official image from GitHub Container Registry:

```
docker pull ghcr.io/kurokirasama/dsin:latest
```

To start an interactive session with the complete analytical environment ready for use:

```
docker run --rm -it ghcr.io/kurokirasama/dsin:latest
```

2.2.4 Defining a Shell Shortcut (Optional)

If you prefer to launch the container by typing `dsin` in your host terminal, add an alias to your host shell configuration file (`~/.bashrc` or `~/.config/nushell/config.nu`):

```
alias dsin='docker run --rm -it ghcr.io/kurokirasama/dsin:latest'
```

Once inside the `dsin` container, confirm your initial working directory, inspect the bundled data folders in `/data`, and check that all required CLI tools and plugins are recognized:

```
pwd

ls /data

version

which nu ttt timg parallel vw tapkee run_experiment Rscript python3

plugin list
```

3 Nushell Fundamentals and Data Structures

3.1 Data Science

According to Hilary Mason and Chris H. Wiggins (2010), *data science* is framed around five interconnected stages, commonly referred to as the OSEMN framework:

1. Obtaining data.
2. Scrubbing data.
3. Exploring data.
4. Modeling data.
5. Interpreting data.

This book focuses hands-on attention on the first four computational stages.

Although these five stages are often presented sequentially, practical data science work moves fluidly back and forth between them, frequently executing multiple stages in parallel (Figure 1). Data science is fundamentally an iterative, non-linear discipline.

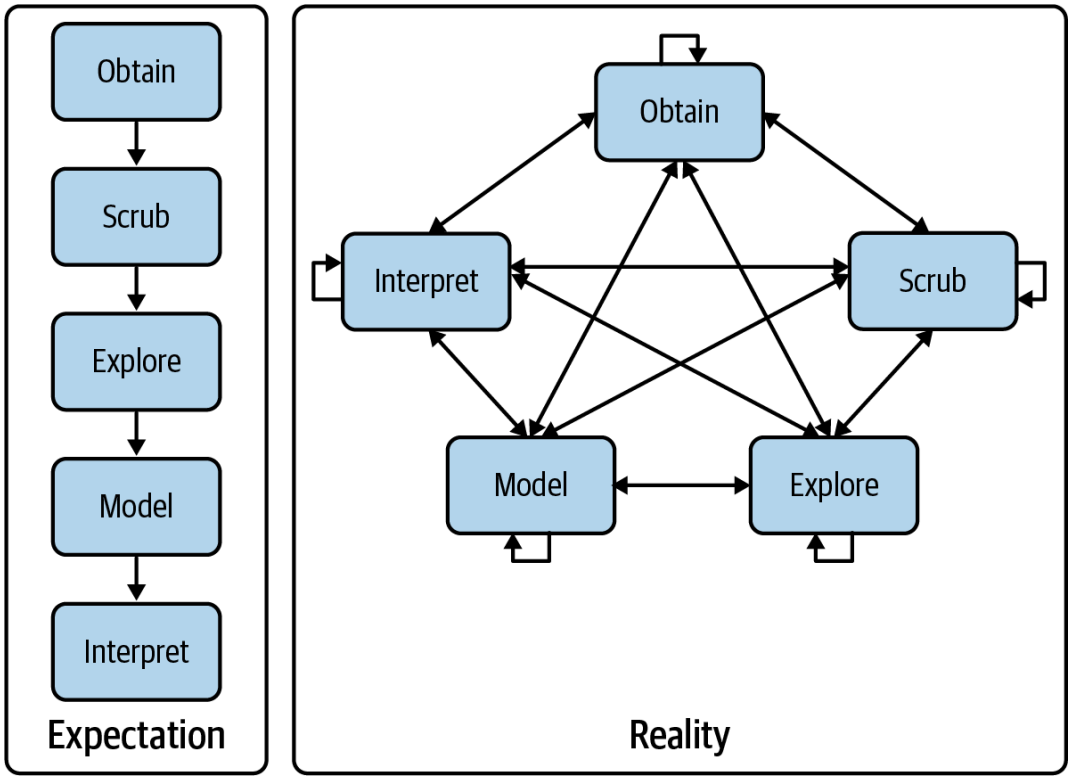


Figure 1: The data science lifecycle (OSEMN framework).

3.1.1 Obtaining Data

Without data, there is no data science; acquiring information is therefore the indispensable starting point. Depending on the problem domain, this stage involves:

- Downloading data from external servers, web endpoints, or public repositories.
- Querying relational databases or REST APIs (for instance, SQLite, PostgreSQL, or web services).
- Parsing and ingesting structured or semi-structured files (such as CSV, JSON, HTML, XML, or Excel spreadsheets).
- Capturing real-time streams from physical sensors or digital loggers.

Common storage formats encountered in practice include plain text (`.txt`), delimited tables (`.csv`, `.tsv`), hierarchical representations (`.json`, `.xml`, `.html`), and columnar formats (`.parquet`).

3.1.2 Scrubbing Data

Raw data rarely arrives clean. Real-world datasets routinely suffer from missing observations, formatting inconsistencies, schema violations, corrupt encodings, and irrelevant columns. In these situations, data must be scrubbed before analysis can take place. Essential operations include:

- Filtering invalid or out-of-range rows.
- Selecting and reordering relevant columns.
- Normalizing, trimming, and recasting string values.
- Imputing or dropping missing values and deduplicating rows.
- Converting between serialization formats.

In industry practice, data acquisition and scrubbing account for roughly 80% of total project effort. The remaining 20% encompasses exploratory analysis, modeling, and communication.

3.1.3 Exploring Data

Once the data is scrubbed and structured, exploratory data analysis begins. This stage builds a reliable mental model of the underlying distributions and relationships. Working directly in the terminal, you will:

- Inspect schemas, data types, and value distributions.
- Compute descriptive summary statistics (central tendency, variance, quantiles).
- Generate high-fidelity visual representations and plots directly in your console.

3.1.4 Modeling Data

When the goal is to explain observed relationships or forecast future values, statistical and machine learning models are required. Standard modeling techniques include dimensionality reduction, unsupervised clustering, regression, and supervised classification.

While implementing novel algorithms from scratch is rarely done in shell pipelines, orchestrating training pipelines, evaluating performance metrics, and managing model inferences directly from the command line is exceptionally powerful.

3.1.5 Interpreting Data

The final stage is interpreting the findings. This stage centers on:

- Drawing substantive, actionable conclusions from model outputs.
- Evaluating empirical validity and algorithmic bias.
- Communicating results clearly to technical and non-technical stakeholders.

This stage demands domain expertise and critical judgment rather than automated scripts.

3.2 History of the Terminal

A *terminal emulator* is a program that provides access to the *command-line interface* through a particular command interpreter called a *shell*.

Historically, a terminal was a dedicated physical hardware device used to transmit keystrokes to and display characters from a host computer.

Early computers were massive, room-sized installations (Figure 2). The earliest interactive terminals were electromechanical teletypewriters (commonly abbreviated *tty*), as illustrated in Figure 3.



Figure 2: Early mainframe computer systems.

As semiconductor and cathode-ray tube technology progressed, mechanical printers gave way to video display terminals (Figure 4).

It is worth emphasizing that the device pictured in Figure 4 contains no onboard computer; it is solely an interface terminal connected across serial lines to a remote mainframe. Figure 5 illustrates a terminal cabled directly to its host computer.



Figure 3: Electromechanical teleprinters (teletypewriter or tty).

Today, dedicated physical terminals are obsolete. Instead, software applications known as *terminal emulators* simulate historical display hardware, communicating directly with the operating system without requiring graphical widgets (windows, buttons, dialog boxes).

3.2.1 The Shell

The *shell* is the command interpreter that parses and executes instructions entered into the terminal. Numerous shells exist, including Bash, Zsh, Fish, and Nushell. Bash historically serves as the default shell across most GNU/Linux distributions.

In this textbook, we work exclusively with Nushell. In Nushell, each command line begins with an interactive prompt (such as `/data>`), signaling that the environment is ready for input.

3.2.2 Advantages of the Command Line for Data Science

Using the command line for data science offers substantial operational advantages:

- **Agility:** The command line supports rapid, interactive data exploration, returning immediate feedback as pipelines are assembled.
- **Integration:** Shell commands bridge disparate tools, programming runtimes, and external services cleanly.
- **Scalability and Automation:** Command sequences are effortlessly captured in reusable scripts and automated cron jobs.
- **Extensibility:** Shell pipelines orchestrate tools written in any language (Rust, C, Python, R).
- **Ubiquity:** Command-line environments are universally available across local workstations, cloud virtual machines, and high-performance computing clusters.



Figure 4: Classic video display terminals (DEC VT100).

3.3 The Terminal

Before exploring Nushell's typed data structures, let us review the core mechanics of the terminal environment and examine how traditional shells operated before Nushell.

3.3.1 The Interactive Environment (REPL)

Command-line interaction is governed by an ongoing loop known as a **REPL** (Read-Eval-Print Loop):

1. The terminal displays an interactive prompt (for instance, `/data/ch03>`), indicating the active directory and signaling readiness for input.
2. The user types a command pipeline and presses **`Enter`**.
3. The shell parses the text, checks syntax, evaluates the pipeline, and executes the designated programs or built-in functions.
4. The resulting output renders in the terminal window, and the prompt reappears.

3.3.2 Running Commands and Saving Output to Files

To work through the examples in this chapter, switch into the chapter workspace:

```
cd /data/ch03
```

From this point forward, commands run relative to `/data/ch03`. Inspect the files in this directory:

```
ls
```



Figure 5: Video terminal cabled to a PDP-11 minicomputer.

To examine the first 5 rows of `movies.txt`, open the file in the current directory (`open movies.txt`), split its contents into lines (`lines`), and select the first 5 elements (`first 5`):

```
open movies.txt | lines | first 5
```

In reproducible data pipelines, we frequently need to write processed outputs to disk rather than merely inspecting them on screen. Use the `save -f` command to write data to a file:

```
open movies.txt | lines | first 5 | to text | save -f movies_top5.txt

open movies_top5.txt
```

3.3.3 Command-Line Tool Categories

When you enter a command name in the terminal, it resolves to one of five distinct categories:

1. **Nushell built-in command:** An internal command executed directly in memory by Nushell's runtime engine, operating natively on structured data (for example, `ls`, `open`, `where`, `select`, `math avg`).
2. **Custom command (function):** A user-defined or module-exported command declared with `def` or `export def` (such as `to-vw` or `tapkee-reduce`).
3. **Alias:** A shorthand identifier that expands into another command with predefined flags or arguments (`alias ll = ls -l`).
4. **Interpreted executable script:** A text file with executable permissions and an initial shebang line (`#!/usr/bin/env nu`, `#!/usr/bin/env python3`, `#!/usr/bin/env Rscript`).
5. **Compiled binary executable:** A machine-code program residing on the system `.PATH` (such as `ttt`, `timg`, `vw`, `tapkee`, or `git`).

You can check how any command resolves by using `which`:

```
which ls open ttt timg python3 Rscript
```

3.3.4 Traditional Shells (Bash and Zsh): Architecture, Philosophy, and Limitations

To understand why **Nushell** represents a fundamental architectural advancement for data science, it helps to examine how traditional Unix shells (such as the Bourne Shell `sh`, Bourne-Again Shell `bash`, and Z Shell `zsh`) process data.

3.3.4.1 The Role of Traditional Shells

In Unix-like systems, a traditional shell performs two distinct roles simultaneously:

- **Interactive command interpreter:** It acts as the user's interface to the operating system kernel. When you enter a command in `bash`, the shell locates the binary across the directories defined in the `PATH` environment variable, `asks the kernel to spawn a child process via`fork()`and`execve()‘, and waits for the process to exit.
- **Scripting and automation language:** It provides syntax for control flow (`if/then/fi`, `for`, `while`), environment variable management, and process chaining to automate system administration routines.

3.3.4.2 The Unix Philosophy and Plain-Text Streams

Formulated at Bell Labs during the 1970s by Ken Thompson, Dennis Ritchie, and Douglas McIlroy (who invented Unix pipes in 1973), the **Unix philosophy** rests on three central tenets:

1. *Write programs that do one thing and do it well.*
2. *Write programs to work together.*
3. *Write programs to handle text streams, because that is a universal interface.*

Under POSIX-compliant shells like `bash`, processes communicate across three unstructured byte streams:

- **Standard input (`stdin`, file descriptor 0):** The byte stream read by the program.
- **Standard output (`stdout`, file descriptor 1):** The byte stream where normal program output is written.
- **Standard error (`stderr`, file descriptor 2):** An unbuffered byte stream reserved for diagnostics and error messages.

Connecting commands via the pipe operator (`|`) redirects the `stdout` of one process directly into the `stdin` of the next.

3.3.4.3 Limitations of Plain-Text Streams for Data Science

While treating data as unstructured byte streams was revolutionary in 1973, it introduces severe friction for modern data science workflows:

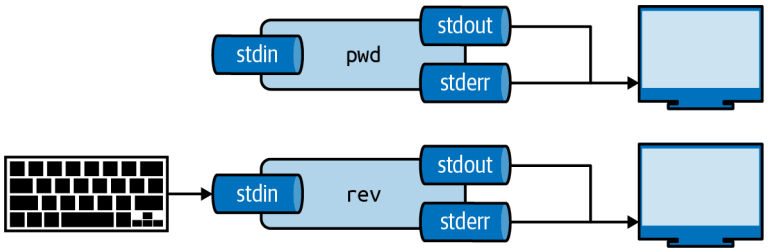


Figure 6: Standard output redirection in the terminal

1. **Loss of types and semantic schemas:** Traditional pipes discard all type metadata. Numbers, dates, and nested records collapse into undifferentiated ASCII/UTF-8 text. Every subsequent command in the pipeline must guess column widths, parse delimiters, and reconstruct types from scratch.
2. **Fragility against whitespace and quoting:** If a field contains commas, tabs, spaces, or newline characters, downstream tools (`cut`, `awk`, `xargs`) frequently mangle the record unless guarded by brittle escaping rules.
3. **Redundant serialization overhead:** In a pipeline of five text-based tools, data is converted from internal binary representation to ASCII text and parsed back into binary at each stage.
4. **Ad-hoc parsing:** Extracting specific fields requires brittle regular expressions that easily break when input formats change slightly.

3.3.4.4 The Nushell Paradigm: Structured Data and Typed Pipelines

Nushell replaces unstructured byte streams with **typed data pipelines**. In Nushell, the pipe operator (`|`) does not stream raw bytes; it passes rich in-memory data structures:

```
open /data/ch03/movies.txt | lines | find -i "comedy" | length
```

Compare this to the equivalent Bash pipeline:

```
cat /data/ch03/movies.txt | grep -i "comedy" | wc -l
```

While the syntax looks deceptively similar, what flows between commands is fundamentally different. In Nushell, `open` parses the file format, `lines` produces a typed list of strings, `find` filters the list, and `length` computes the integer count of matching elements directly in memory without character parsing or subshell spawning.

To write data to disk in Nushell, redirect the structured pipeline to the `save` command:

```
"hello" | save -f output.txt
```

Data flows naturally through Nushell pipelines until reaching a terminal command or rendering in the console. You can emit and inspect any data type directly (scalars, lists, records, and tables) by composing commands with the pipe operator (`|`).

3.4 Bashisms and Productivity Shortcuts in Nushell

Although Nushell is not a POSIX shell, it deliberately incorporates familiar interactive shortcuts from Bash (often referred to as *bashisms*) to streamline daily command-line work without compromising structured semantics:

3.4.1 Bang-Bang History Expansion (`!`)

Powered by Nushell's interactive line editor (`reedline`), history expansion shortcuts are supported out of the box:

- **`!!`:** Inserts the previous command into the editing buffer.
- **`!$`:** Inserts the last argument or whitespace-delimited token from the previous command.
- **`!<index>`:** Retrieves the command at that specific history index (inspected via `history`).
- **`!-<n>`:** Retrieves the command executed $ steps back in history.
- **`!<prefix>`:** Retrieves the most recent command beginning with that prefix.

Unlike Bash, which immediately executes expanded commands upon pressing `Enter`, Nushell **places the expanded command into your line editor**, allowing you to inspect, modify, and confirm the text before execution. You can explore the interactive tutorial by running:

```
tutor bashisms
```

3.4.2 External Process Control and Sequencing

- **Sequencing with semicolons (`;`):** Chain multiple commands sequentially using `;` (for example, `cd /data; ls`). Logical operators in Nushell are the explicit keywords `and` and `or` rather than `&&` and `||`.
- **Explicit external command invocation (`^`):** When a Nushell built-in shares a name with an operating system binary on `.PATH` (such as `ls`, `ps`, or `find`), prefixing the command with a caret `^` (`^ls`, `^ps`) forces the execution of the external system binary.
- **One-shot environment variables:** To define an environment variable scoped exclusively to a single external process execution, prefix the command with the assignment (for example, `FOO=bar ^my_script`).
- **External stream redirections:** Nushell supports redirection operators for external processes (`o>` for *stdout*, `e>` for *stderr*, `o+e>` for combined streams, and `e>|` to pipe errors).

3.4.3 The Nushell Environment Is Scoped

Nushell draws multiple design principles from compiled programming languages. Foremost among these principles is that programming languages should avoid global mutable state. Traditional shells rely heavily on global mutation to modify the environment, but Nushell intentionally departs from this paradigm.

In Nushell, blocks control their own environment. Any environmental modifications remain strictly scoped to the block where they occur.

In practice, this enables concise, robust pipelines when traversing subdirectories. For example, to compile every project subdirectory within the current folder:

```
ls | where type == dir | each { |row|
    cd $row.name
    make
}
```

The `cd` command modifies the `$env.PWD` environment variable, but this change never escapes the block closure. Consequently, each iteration begins afresh from the original directory before navigating into the next target subdirectory.

Maintaining a scoped environment makes pipelines deterministic, readable, and straightforward to debug. When intentional environment updates are necessary, Nushell provides dedicated commands like `def --env` and `load-env` to batch environment modifications explicitly.

3.5 Basic Elements of Nushell

This section introduces the foundational concepts and interactive workflows of Nushell.

Nushell operates under the principle that all data flowing through pipelines is structured. Consequently, you can query, filter, and transform command outputs using declarative, relational operations similar to SQL. To understand this distinction, contrast the unstructured plain-text stream generated by the operating system's external utility with the structured table returned natively by Nushell.

When invoking the traditional external system binary by prefixing it with a caret `^`:

```
^ls
```

> **Note on the `^` operator:** In Nushell, the `^` symbol acts as an escape operator for external operating system binaries. It instructs the engine to bypass any internal built-in command, alias, or custom function and execute the matching executable binary found on `$env.PATH` (in this case, `/bin/ls`). Its output is an unstructured character stream.

Conversely, running Nushell's native built-in `ls` command:

```
ls
```

`ls` returns a typed, structured table rather than raw text. Each row represents a file or directory, and each column (`name`, `type`, `size`, `modified`) has a concrete semantic data type. Just as in a database or a spreadsheet, you can sort, filter, project, and query these rows directly.

To sort the table by file size, pipe the output of `ls` into `sort-by`:

```
ls | sort-by size | reverse
```

```
ls | sort-by size -r

ls | sort-by modified
```

We notice that we do not pass flags to `ls` to sort its output. Instead, we use Nushell's general-purpose `sort-by` command on the structured table emitted by `ls`. To display the largest files first, append `reverse` or use the `-r` flag on `sort-by`.

Nushell provides many commands that operate directly on tables. For example, use `where` to filter rows by size:

```
ls | where size > 100kb
```

Consistent with the Unix philosophy of tool composition, commands connect cleanly to form versatile combinations. Consider another built-in command:

```
ps
```

This command returns a table of all active processes running on the system, showing their process ID, name, status, and resource consumption. To isolate processes actively consuming CPU time:

```
ps | where cpu > 0
```

Nushell supports not only flat tabular structures, but arbitrarily nested hierarchies as well. For example, examine the output of `help commands`:

```
help commands
```

Because the output is extensive, filter down to the row corresponding to the `each` command:

```
help commands | where name == each

help commands | where name == each | first
```

Extracting a single row from a table produces a *record*, which is an ordered set of key-value pairs. Notice how the `signatures` column contains nested records rather than scalar values. To inspect a specific column value, use `get`:

```
help commands | where name == each | first | get description

help commands | where name == each | first | get params
```

The `get` command navigates directly into structured data (tables, records, or lists). You can supply nested path segments to traverse structures to arbitrary depth:

```
help commands | where name == each | first | get category
```

```
help commands | where name == each | first | get command_type

help commands | where name == each | first | get description

help commands | where name == each | first
```

These path specifications are known as *cell paths*. Nushell uses the cell path to extract the designated value from a table, record, or list. Cell paths also support numeric row indices:

```
help commands | where name == each | first | get params

help commands | where name == each | first | get params.0

help commands | where name == each | first | get params.3.description
```

To view deeply nested structures expanded in the console, use `table -e`:

```
help commands | where name == each | first | table -e
```

Standard operating system utilities can always be invoked by prefixing them with `^` when a Nushell built-in shares their name. If no built-in conflicts, the caret prefix is optional:

```
^ls

open tips.csv | first 5

uname
```

Every Nushell command that outputs structured data does so as a table or list:

```
# inspect running system processes
ps

# sort active processes by highest CPU usage
ps | where cpu > 0 | sort-by cpu -r
```

3.5.1 Getting Help

You can inspect the documentation for any built-in Nushell command using `help` or the `--help` flag. You can also search command descriptions by keyword using `help -f <topic>`:

```
help explore

help http get

help -f open
```

Additionally, Nushell provides an interactive tutorial built directly into the terminal, designed for progressive, hands-on learning:

- `tutor`: Displays the welcome message and general instructions.
- `tutor list`: Lists all available learning modules (tables, cell paths, expressions, variables, blocks, conditionals, closures).
- `tutor <topic>`: Opens the interactive lesson for the specified concept (for example, `tutor table`, `tutor closure`, or `tutor pipeline-input`).
- `tutor -f <query>` or `tutor --find <query>`: Searches across lesson texts for specific explanations and patterns.

```
tutor

tutor list

tutor table
```

3.6 Data Types and Variables in Nushell

Having worked practically with structured tables, relational filters with physical units (`where size > 100kb`), and cell paths, we now formalize the core mechanics of Nushell's type system, its two-phase compilation architecture, and variable declarations with `let`, `mut`, and `const`.

3.6.1 The Nushell Type System

One of the most consequential design choices in Nushell is its strongly typed data model. In legacy shells like Bash or Zsh, pipelines pass undifferentiated byte streams, forcing users to constantly parse text with `awk`, `cut`, or regular expressions. In Nushell, every value that travels through a pipeline retains its explicit type identity. This preserves semantic meaning, allows compile-time and runtime validation, and makes complex transformations reliable.

To inspect the runtime type of any value or pipeline output, use the `describe` command:

```
[
  (42 | describe),
  (3.14 | describe),
  ("data science" | describe),
  (true | describe),
  (2026-10-21 | describe),
  (15min | describe),
  (2.5gb | describe)
]
```

Nushell categorizes types into primitive scalars, dimensional quantities (filesizes and durations), structured collections, and language runtime types. Table 1 summarizes the core catalog:

Table 1: Fundamental data types in Nushell.

Category	Data Type	Example Syntax and Literals	Operational Description
Scalars	`int`	`42`, `-10`, `0x2A`	64-bit signed integer for discrete counts and arithmetic.
Scalars	`float`	`3.14159`, `-0.001`, `Infinity`	64-bit IEEE 754 double-precision floating-point number.
Scalars	`string`	`"text"`, `'simple'`, `r#'raw'#`	UTF-8 encoded text string supporting string interpolation `$"..."`.
Scalars	`bool`	`true`, `false`	Boolean value for logical branching and filter predicates.
Scalars	`nothing`	`null`	Represents the explicit absence of a value or an empty result.
Dimensional	`filesize`	`512b`, `10kb`, `4mb`, `2gb`, `1tb`	Byte magnitude supporting unit arithmetic (`2gb - 500mb`).
Dimensional	`duration`	`100ms`, `5sec`, `2min`, `3hr`, `7day`	Time interval supporting automatic unit conversion (`1hr + 30min`).
Dimensional	`datetime`	`2026-10-21`, `2026-10-21T15:30:00Z`	Absolute timestamp in ISO 8601 format with timezone support.
Structures	`list<T>`	`[1 2 3]`, `["a" "b" "c"]`	Ordered collection of elements indexed sequentially.
Structures	`record<...>`	`{name: "Carlos", age: 30}`	Set of key-value pairs analogous to a JSON object.
Structures	`table<...>`	`[[id value]; [1 100] [2 200]]`	Two-dimensional table equivalent to a homogeneous list of records.

Category	Data Type	Example Syntax and Literals	Operational Description
Structures	`range`	`1..10`, `0..<5`, `1..`	Memory-efficient contiguous or half-open interval.
Advanced	`binary`	`0x[deadbeef]`, `0x[48 65 6c 6c 6f]`	Raw byte sequence for network payloads and binary files.
Advanced	`closure`	`{|x| $x * 2 }`, `{ $in > 10 }`	Anonymous executable block with parameters or pipeline input `$in`.
Advanced	`glob`	`*.csv`, `**/*.parquet`	Filesystem pattern matching specification.
Advanced	`cell-path`	`$.name`, `$.column.0`	Declarative path accessing internal record keys or table cells.

Dimensional types represent a major analytical asset: Nushell natively understands physical units and normalizes comparisons automatically. When filtering records by file size (`where size > 50mb`) or latency thresholds (`where latency < 200ms`), the engine performs exact unit arithmetic without requiring manual algebraic conversions.

3.6.2 Nushell as a Compiled Language

A key architectural aspect of Nushell where it diverges from dynamic shells is its two-phase execution model: source code is compiled before it runs. Nushell does not provide an `eval` function that injects arbitrary new syntax into an executing environment. Tasks like importing files must use paths known at compile time, much like `#include` directives in C++ or `use` statements in Rust.

For example, the following sequence fails when executed as a script:

```
"def abc [] { 1 + 2 }" | save -f output.nu

source "output.nu"

abc
```

The `source` statement expands code during the compilation phase, but the preceding `save` command has not yet executed. Nushell analyzes the entire block as a cohesive unit before evaluating runtime instructions. Because `output.nu` does not exist on disk when the parser runs, `source`

cannot locate the definition of `abc`.

Similarly, building source paths dynamically at runtime fails:

```
source $"($my_path)/common.nu"
```

If `$my_path` is a regular runtime variable declared with `let`, the compilation phase cannot resolve the path string.

However, if `$my_path` is declared as a compile-time constant, the parser can evaluate the string ahead of execution:

```
const my_path = ([$nu.home-dir nushell] | path join)

source $"($my_path)/common.nu"
```

Nushell uses a single compilation pass for source code, keeping parsing strictly separated from evaluation. This architecture enables reliable IDE integration, descriptive compile-time diagnostics, and clean interoperability with developer tooling.

3.6.3 Variable Declarations: let, mut, and const

Binding values in Nushell centers around three keywords: `let`, `mut`, and `const`. Each defines specific semantics regarding evaluation timing (compile time versus runtime), memory immutability, and closure capture.

3.6.3.1 Naming and Referencing Rules

Once declared, a variable is referenced by prefixing its identifier with a dollar sign (`$`):

Nushell restricts certain characters in variable names to prevent syntactic ambiguity with operators:

```
.  [  (  {  +  -  *  ^  /  =  !  <  >  &  |
```

In declaration statements, prefixing the variable name with `$` is optional (`let $var = 42` is identical to `let var = 42`).

3.6.3.2 Runtime Immutability: let

Emphasizing functional programming principles, Nushell defaults to immutability. Variables declared with `let` are evaluated dynamically at runtime and remain strictly unmodifiable throughout their scope:

```
let rate = 0.15

$rate = 0.20 # Error: nu::shell::assignment_requires_mutable_variable
```

While immutable bindings cannot be reassigned with `=`, Nushell fully supports **variable shadowing**. You can declare a new binding with the same name in the same scope or an inner block, replacing the reference without violating referential transparency:

```
let count = 10

do {
  let count = 50
  $count # evaluates to 50 in inner scope
}

$count # outer scope remains 10

# Explicit shadowing in the current scope: now 11
let count = $count + 1
```

Immutable bindings declared with `let` are thread-safe and can be captured freely inside functional closures such as `each` and `par-each`.

3.6.3.3 Controlled Imperative Mutability: mut

When algorithms require iterative state accumulation (such as in `loop` or `while` constructs), Nushell provides the `mut` keyword. Mutable variables can be updated in place using compound assignment operators:

```
mut total = 0

$total += 15

$total *= 2

$total # result: 30
```

Nushell supports several assignment operators for mutable variables, detailed in Table 2:

Table 2: Assignment operators for mutable variables in Nushell.

Operator	Operational Description
`=`	Assigns a new value directly to the mutable variable.
`+=`	Adds a value to the variable in place (`$val += 5`).
`-=`	Subtracts a value from the variable in place (`$val -= 2`).
`*=`	Multiplies the variable by a value in place (`$val *= 3`).

Operator	Operational Description
`/=`	Divides the variable by a value in place (`$val /= 2`).
`++=`	Appends an item or concatenates a list onto an existing list variable.

Compound arithmetic operators (`+=`, `-=`, `*=`, `/=`) only apply to types that support the corresponding arithmetic operation. The `++=` operator requires that the target variable or operand be a list.

Crucially, **closures and nested functions cannot capture mutable variables**. The following pipeline triggers an immediate compiler error:

```
mut counter = 0

[1 2 3] | each { $counter += 1 } # Error: nu::shell::capture_of_mutable_var
```

This restriction protects pipeline integrity: Nushell pipelines stream elements lazily and frequently run concurrently across threads (via `par-each`). Allowing closures to mutate external state would introduce data races and nondeterministic behavior.

3.6.3.4 Parse-Time Constants: const

Nushell enforces a strict two-stage lifecycle:

1. **Parse-time (static compilation):** The Nushell parser analyzes the complete source code of a script or command block before execution begins. In this stage, syntax is validated, command signatures are resolved, keywords are checked, and modules are loaded (`source`, `use`).
2. **Runtime (dynamic evaluation):** Data streams through pipelines, expressions are evaluated, files are written (`save`), external processes are spawned, and results are returned.

Understanding this boundary is vital. Consider the following sequence:

```
"def abc [] { 1 + 2 }" | save -f output.nu

source "output.nu"

abc
```

When entered line by line in the interactive REPL, this works as expected: the first line compiles and runs, creating `output.nu` on disk; when the second line is entered, `output.nu` exists physically, allowing the parser to read it and register `abc`.

However, executing the block as a single script invocation fails immediately with `nu::parser::sourced_file_not_found`:

```
Error: nu::parser::sourced_file_not_found
  × File not found: output.nu
  help: sourced files need to be available before your script is run
```

Because `source` is a parse-time directive, Nushell attempts to read `output.nu` during the compilation pass to check the call to `abc`. At that point, the preceding `save` command has not yet executed.

To provide values needed during compilation (such as static module paths or default parameter values in command signatures), use `const`. Constants declared with `const` are evaluated statically during parse time and can only be bound to literal values or pure expressions that the compiler can evaluate ahead of time:

```
const module_path = ([$nu.home-dir my_scripts] | path join)

use $"($module_path)/utilities.nu" *
```

Supplying a runtime variable declared with `let` to `use` fails compilation immediately, because runtime values do not exist in memory when the script is parsed.

Table 3 compares the operational characteristics of the three binding mechanisms:

Table 3: Variable binding models in Nushell: `let`, `mut`, and `const`.

Criterion	`let` (Immutable)	`mut` (Mutable)	`const` (Compile-time Constant)
Evaluation Timing	Runtime	Runtime	Parse-time
Direct Reassignment (`=`)	Not permitted	Permitted (`$var = ...`, `+=`)	Not permitted
Variable Shadowing	Yes (`let x = $x + 1`)	No (reassignment is used)	Yes (in inner scopes)
Closure Capture	Yes (thread-safe for streaming/parallel)	No (rejected by compiler)	Yes
Usable in `source` / `use`	No (unknown at parse time)	No	Yes (resolved statically)
Recommended Usage	Default choice for 90%+ of code	Imperative loops (`while`, `loop`)	Module paths, fixed schemas

3.6.3.5 Functional Pipelines versus Imperative Mutation

In idiomatic Nushell code, immutable variables and functional transformations are strongly preferred for analytical tasks. Pipeline-oriented transformations offer three major advantages:

1. **Performance:** They avoid repetitive reallocations of intermediate buffers in memory.

2. **Native streaming:** Subsequent pipeline stages begin processing data as soon as the first item is emitted, without waiting for earlier loops to finish.
3. **Concurrency:** Functional pipelines easily parallelize across multi-core processors using `par-each`.

Common accumulation patterns that require mutable variables in imperative languages are expressed declaratively in Nushell:

- **Index tracking:** Instead of incrementing a counter variable, combine `enumerate` with `each`:

```
ls |

  enumerate |

  each { |elt|
    $"Item #($elt.index) has size ($elt.item.size)"
  }
```

- **Accumulation and reduction:** To fold a collection into an aggregate value, use `reduce`:

```
[one two three four five six] | reduce { |current, longest|
  if ($current | str length) > ($longest | str length) {
    $current
  } else {
    $longest
  }
}
```

- **Dynamic sequence generation:** To consume data from generators or streams on demand, use `generate`:

```
generate { |counter|
  if $counter < 5 {
    { out: ($counter * 10), next: ($counter + 1) }
  }
} 0
```

3.6.3.5.1 Performance Implications

Using pipeline filters with immutable variables is often orders of magnitude faster than imperative mutation using `for` and `while` loops. When appending items to a mutable list inside a `for` loop, each `append` allocates a new list buffer:

- **Imperative approach with `for` loop and mutable variable:**

```
timeit {
  mut randoms = []
  for _ in 1..500 {
```

```
    $randoms = ($randoms | append (random int))
  }
}
```

- **Idiomatic functional approach with `each`:**

```
timeit {
  let randoms = 1..500 | each { random int }
}
```

In addition to executing much faster, `each` streams its results immediately through the pipeline, allowing downstream commands to process records concurrently.

3.7 Navigating the Filesystem

Navigating and inspecting the filesystem is a fundamental responsibility of any shell. Nushell provides dedicated commands that operate cleanly on structured filesystem metadata:

3.7.1 Inspecting Directory Contents

```
cd /data/ch03

ls
```

As demonstrated earlier, `ls` returns directory contents as a structured table.

`ls` also accepts an optional glob pattern to filter entries. For example, to list only CSV files:

```
cd /data/ch03

ls *.csv
```

3.7.2 Glob Patterns (Wildcards)

The asterisk (*) in `*.csv` represents a *glob pattern* matching paths on the filesystem. The pattern `*.csv` matches any filename ending with `.csv`.

The most general wildcard is `*`, which matches any sequence of characters in the current directory. You can combine it with prefixes or file extensions, such as `*.bak` or `temp*`.

Nushell also supports the recursive glob `**`, which traverses nested subdirectories. For instance, `ls **/*.csv` searches through all directory levels starting from `/data`:

```
cd /data

ls **/*.csv
```

In addition to `*`, Nushell glob syntax supports single-character wildcards with `?` and character classes with `[...]`.

Literal wildcards (`*`, `?`, `[]`) can be escaped by enclosing the path in single quotes, double quotes, or raw string syntax (`r#'...'#`). For example, to inspect a literal directory named `[slug]`, use `ls "[slug]"` or `ls '[slug]'`.

However, strings enclosed in backticks do not escape glob characters. Compare these three cases:

1. Bare words: Interpreted as glob patterns.

```
rm *myfile*
```

2. Quoted strings: Treated as literal strings with escaped wildcard characters.

```
rm "*myfile*"
```

3. Backtick-quoted strings: Interpreted as glob patterns, allowing spaces within glob paths.

```
rm `*myfile*`
```

   Nushell also includes a standalone `glob` command for building advanced file matching pipelines.

3.7.2.1 Programmatic Glob Generation

When glob patterns must be generated dynamically in code, use these patterns:

1. `into glob`: Converts a string into a typed glob value:

```
let current_month = date now | format date '%Y-%m'

let glob_pattern = $"*($current_month)*" | into glob

ls $glob_pattern
```

2. The spread operator (`...`) combined with `glob`:

```
let current_month = date now | format date '%Y-%m'

ls ...(glob $"*($current_month)*")
```

3. Explicit type annotation:

```
let current_month = date now | format date '%Y-%m'

let glob_pattern: glob = $"*($current_month)*"

ls $glob_pattern
```

3.8 Working with Strings

Strings represent sequences of characters used for filenames, column labels, text corpora, and configuration settings. Nushell provides several string literal formats tailored for different situations:

3.8.1 String Formats

String Format	Example	Escaping Rules	Notes
Single-quoted strings	`'[^\n]+'`	None	Cannot contain single quotes inside the string.
Double-quoted strings	`"The\nEnd"`	C-style backslash escapes	Literal backslashes must be escaped.
Raw strings	`r#'Raw string'#`	None	May contain single and double quotes.
Bare-word strings	`ozymandias`	None	Only alphanumeric word characters; cannot appear in command position.
Backtick-quoted strings	`[^\n]+`	None	Simple string that may include whitespace.
Single-quoted interpolation	`$'Captain ($name)'`	None	Cannot contain unescaped single quotes.
Double-quoted interpolation	`$"Captain ($name)"`	C-style backslash escapes	Literal backslashes and parentheses must be escaped.

3.8.2 Single-Quoted Strings

Single-quoted strings are enclosed in `'`. They do not interpret escape sequences, making them ideal for regular expressions and literal text:

```
'hello world'

'The

end'
```

3.8.3 Double-Quoted Strings

Double-quoted strings are enclosed in `"` and support standard backslash escape sequences:

```
"hello\nworld"
```

Supported escape sequences include: - `\"` (double quote) - `\'` (single quote) - `\\` (backslash) - `\/` (forward slash) - `\b` (backspace) - `\f` (form feed) - `\r` (carriage return) - `\n` (newline) - `\t` (tab) - `\u{X...}` (Unicode code point with 1 to 6 hexadecimal digits)

3.8.4 Raw Strings

Raw strings allow single and double quotes inside text without escaping. They are delimited by `r#'` and `'#`:

```
r#'Raw strings can contain 'quoted' text.'#
```

You can increase the number of `#` delimiters to nest raw strings:

```
r###'r##'This is an example of a raw string.'##'###
```

3.8.5 Bare-Word Strings

Simple alphanumeric words can often be written without quotes:

```
print hello

print "hello"

[hello] | describe
```

However, bare words in the leading position of an expression or inside parentheses `(...)` are evaluated as command names:

```
hello

Error: nu::shell::external_command

  × External command failed

   ╭─[entry #5:1:1]

 1 | hello

   · ──┬──

   ·   └── executable was not found

   ╰────
```

```
  help: program not found
```

If a bare word conflicts with a built-in keyword (`true`, `false`, `null`), Nushell treats it as that keyword rather than a string:

```
true | describe

[true] | describe

[trueX] | describe

"trueX" | describe
```

In structured scripts, using explicit quotes is recommended to avoid parsing ambiguities.

3.8.6 Backtick-Quoted Strings

Enclosed by ‘, backtick strings can contain spaces while functioning similarly to bare words:

```
^ls
```

They are especially useful for passing paths with spaces to commands:

```
ls `./my dir/*`
```

3.8.7 Executing Strings as External Commands

Prefixing any string expression or variable with ^ instructs Nushell to execute its contents as an external command:

```
^ls

^'ls'

let foo = 'ls'

^$foo
```

You can also use `run-external` for advanced process configuration.

3.8.8 String Concatenation and Joins

Nushell provides several operators and commands for combining strings:

```
['foo', 'bar'] | each {|s| '~/' ++ $s}

['foo', 'bar'] | each {|s| '~/' + $s}
```

Regular expressions can also prepend or append substrings across collections:

```
['foo', 'bar'] | str replace -r '^' '~/'

['foo', 'bar'] | str replace -r '$' '~/'
```

To join list elements into a single string:

```
"hello" | append "world!" | str join " "

ls | get name | str join "/"
```

Using `reduce`:

```
1..10 | reduce -f "" {|elt, acc| $acc + ($elt | into string) + " + "}
```

For most join operations, `str join` is the clearest and most idiomatic choice.

3.8.9 String Interpolation

String interpolation allows evaluating subexpressions directly inside text literals, delimited by `$"..."` (with escape processing) or `$'...'` (without escape processing):

```
let name = "Alice"

$"greetings, ($name)"

$"greetings, (ls | get name.2)"
```

Subexpressions inside `(...)` evaluate fully, and their results are concatenated into the final string.

Literal parentheses inside interpolated strings can be escaped with backslashes:

```
$"2 + 2 is (2 + 2) \(you guessed it!)"
```

Interpolated strings can be evaluated at parse time when assigned to constants:

```
const x = $"(2kb)"
```

3.8.10 Splitting Strings

Use `split row` to break a string into a list based on a delimiter:

```
"red,green,blue" | split row ","
```

Use `split column` to parse delimited text into a structured table:

```
"red,green,blue" | split column ","
```

Use `split chars` to divide text into individual characters:

```
'aeiou' | split chars
```

3.8.11 String Transformation Commands (`str`)

Common string operations are organized as subcommands of `str`. Inspect the full suite with `help str`.

We check whether a string contains a substring:

```
"hello world" | str contains "o wo"
```

The regex matching operator `=~` provides pattern searching.

3.8.11.1 Trimming

Use `str trim` to remove leading and trailing whitespace:

```
'       My   string   ' | str trim
```

Specify `--left` (`-l`) or `--right` (`-r`) to trim only one side, or pass `--char` (`-c`) to strip specific characters:

```
'=== Nu shell ===' | str trim -r -c '='
```

3.8.11.2 Substrings and Indexing

Extract slices of text using character index ranges:

```
'Hello World!' | str index-of 'o'

'Hello World!' | str index-of 'r'

'Hello World!' | str substring 4..8
```

3.8.11.3 Padding

Use `fill` to pad strings to a designated width:

```
'1234' | fill -a right -c '0' -w 10

'1234' | fill -a left -c '0' -w 10
```

3.8.11.4 Reversing

Use `str reverse` to invert character order:

```
'Nushell' | str reverse

['Nushell' 'is' 'cool'] | str reverse
```

3.8.11.5 Structured Parsing

Use `parse` to extract structured fields into a table using pattern templates or regular expressions:

```
'Nushell 0.80' | parse '{shell} {version}'

'where all data is structured!' | parse --regex '(?P<subject>\w*\s?\w+) is (?P<adjective>\w+)'
```

Standard delimited streams can be parsed directly with `from csv`, `from tsv`, or `from ssv`:

```
"acronym,long\nAPL,A Programming Language" | from csv

"name  duration\nonestop.mid  4:06" | from ssv

"rank\tsuit\nJack\tSpades\nAce\tClubs" | from tsv
```

3.8.12 String Comparisons

In addition to standard comparison operators (`==`, `!=`), Nushell provides dedicated string predicates:

```
'APL' =~ '^\w{0,3}$'

'FORTRAN' !~ '^\w{0,3}$'
```

```
'JavaScript' starts-with 'Java'

'OCaml' ends-with 'Caml'
```

3.8.13 Type Conversions

Convert to string: 1. `into string`: `123 | into string` 2. Interpolation: `$'(123)'`

Convert from string: 1. `into <type>`: `'123' | into int`

3.8.14 ANSI Terminal Colors

Apply ANSI styling with `ansi`:

```
$'(ansi purple_bold)This text is a bold purple!(ansi reset)'
```

Always terminate styled output with `ansi reset` to restore default terminal colors.

3.9 Working with Lists

In Nushell, a list is an ordered collection of values, conceptually equivalent to a single table column without a header label. Commands that operate on table columns work seamlessly on lists. For example, `where` filters list items directly:

```
[bell book candle] | where ($it =~ 'b')
```

3.9.1 Creating Lists

Lists are constructed with square brackets, separating elements with spaces, commas, or newlines:

```
[foo bar baz]

[foo, bar, baz]
```

Nushell lists serialize to and from JSON arrays (`["Item1", "Item2", "Item3"]`) transparently.

3.9.2 Modifying Lists

Use `insert` to place a value at a designated index:

```
[1, 2, 3, 4] | insert 2 10
```

Use `update` to replace an element at a given position:

```
[1, 2, 3, 4] | update 1 10
```

3.9.3 Appending, Prepending, and Slicing

Add items with `prepend`, `append`, or the list concatenation operator `++`:

```
let colors = [yellow green]

let colors = $colors | prepend red

let colors = $colors | append purple

let colors = $colors ++ ["blue"]

let colors = ["black"] ++ $colors
```

```
$colors # [black red yellow green purple blue]
```

Use `skip` to discard leading elements, and `drop` to truncate elements from the end:

```
let colors = [red yellow green purple]

let colors = $colors | skip 1

let colors = $colors | drop 2

$colors # [yellow]
```

Extract head and tail elements using `first` and `last`:

```
let colors = [red yellow green purple black magenta]

let colors = $colors | last 3

$colors # [purple black magenta]
```

```
let colors = [yellow green purple]

let colors = $colors | first 2

$colors # [yellow green]
```

3.9.3.1 Spread Operator (...)

Flatten nested lists or interpolate elements into a new list with `...`:

```
let x = [1 2]

[
   ...$x
   3
   ...(4..7 | take 2)
]
```

3.9.4 Iterating over Lists

Use `each` to execute a code block for every element in a list. The block receives each element as an argument (or as `$in`). Combine with `enumerate` to track element indices:

```
let names = [Mark Tami Amanda Jeremy]

$names | each { |elt| $"Hello, ($elt)!" }
```

```
$names | enumerate | each { |elt| $"($elt.index + 1) - ($elt.item)" }
```

Filter list items with `where`:

```
let colors = [red orange yellow green blue purple]

$colors | where ($it | str ends-with 'e')
```

```
let scores = [7 10 8 6 7]

$scores | where $it > 7 # [10 8]
```

Aggregate list values using `reduce`. The block accepts the current element (`elt`) and an accumulator (`acc`). Use `--fold` (`-f`) to set an initial accumulator value:

```
let scores = [3 8 4]

$"total = ($scores | reduce { |elt, acc| $acc + $elt })" # total = 15

$"total = ($scores | math sum)" # idiomatic equivalent

$"product = ($scores | reduce --fold 1 { |elt, acc| $acc * $elt })" # product = 96

$scores | enumerate | reduce --fold 0 { |elt, acc| $acc + $elt.index * $elt.item } # 0*3 + 1*8 +
↪  2*4 = 16
```

3.9.5 Accessing List Elements

Access list items by numeric index using cell paths (`$list.index`):

```
let names = [Mark Tami Amanda Jeremy]

$names.1 # returns Tami
```

When the index is held in a variable, retrieve the element with `get`:

```
let names = [Mark Tami Amanda Jeremy]

let index = 1

$names | get $index # returns Tami
```

Query the number of items with `length`:

```
[red green blue] | length
```

We check for empty lists with `is-empty`:

```
let colors = [red green blue]

$colors | is-empty # false

let colors = []

$colors | is-empty # true
```

Test membership using `in` and `not-in`:

```
let colors = [red green blue]

'blue' in $colors # true

'yellow' in $colors # false

'gold' not-in $colors # true
```

Evaluate conditions across items with `any`:

```
let colors = [red green blue]

# Does any color name end with "e"?
$colors | any {|elt| $elt | str ends-with "e" } # true

# Is any color name shorter than 3 characters?
$colors | any {|elt| ($elt | str length) < 3 } # false

let scores = [3 8 4]

# Is any score greater than 7?
$scores | any {|elt| $elt > 7 } # true

# Is any score odd?
$scores | any {|elt| $elt mod 2 == 1 } # true
```

Test whether all elements satisfy a condition using `all`:

```
let colors = [red green blue]

# Do all color names end with "e"?
$colors | all {|elt| $elt | str ends-with "e" } # false
```

```
# Are all color names at least 3 characters long?
$colors | all {|elt| ($elt | str length) >= 3 } # true

let scores = [3 8 4]

# Are all scores greater than 7?
$scores | all {|elt| $elt > 7 } # false

# Are all scores even?
$scores | all {|elt| $elt mod 2 == 0 } # false
```

3.9.6 Transforming Lists

Use `flatten` to collapse nested lists into a single list:

```
[1 [2 3] 4 [5 6]]

[1 [2 3] 4 [5 6]] | table -e

[1 [2 3] 4 [5 6]] | flatten # [1 2 3 4 5 6]

[[1 2] [3 [4 5 [6 7 8]]]]

[[1 2] [3 [4 5 [6 7 8]]]] | table -e

[[1 2] [3 [4 5 [6 7 8]]]] | flatten

[[1 2] [3 [4 5 [6 7 8]]]] | flatten | flatten | flatten # [1 2 3 4 5 6 7 8]
```

Use `wrap` to turn a list into a single-column table:

```
let zones = [UTC CET Europe/Moscow Asia/Yekaterinburg]

# Display world clock for selected time zones
$zones |

  wrap 'Zone' |

  upsert Time {|row|
    (date now | date to-timezone $row.Zone | format date '%Y.%m.%d %H:%M')
  }
```

3.10 Working with Records

Records correspond conceptually to individual rows within a structured table. In essence, a record represents a "single-row table." Consequently, most commands engineered to transform a table row also operate seamlessly on individual records. For example, `update` can be invoked directly on a record:

```
let my_record = {
   name: "Sam"
   age: 30
   }

$my_record | update age { $in + 1 }
```

3.10.1 Creating Records

A record is an ordered mapping of zero or more key-value pairs. It behaves similarly to a JSON object and can be declared using identical syntax:

```
# Nushell
{ "apples": 543, "bananas": 411, "oranges": 0 }

╭─────────┬─────╮
│ apples  │ 543 │
│ bananas │ 411 │
│ oranges │ 0   │
╰─────────┴─────╯

# JSON
'{ "apples": 543, "bananas": 411, "oranges": 0 }' | from json
```

In Nushell, key-value entries in a record may also be separated by whitespace or newlines rather than commas.

Because records frequently contain numerous attributes, Nushell renders them vertically by default rather than horizontally across columns. To render a record inline as a single string representation, we convert it to NUON (Nushell Object Notation) via `to nuon`:

```
{
   name: "Sam"
   rank: 10
} | to nuon
```

3.10.2 Updating Records

Just like with lists, values can be inserted into existing records:

```
{ "apples": 543, "bananas": 411, "oranges": 0 } | insert pears { 21 }
```

Values can also be updated directly:

```
{ "apples": 543, "bananas": 411, "oranges": 0 } | update oranges { 100 }
```

To derive an updated record with new fields while maintaining immutability:

- Invoke the `merge` command:

```
let first_record = { name: "Sam", rank: 10 }

$first_record | merge { title: "Mayor" }
```

- Or apply the spread operator (`...`) to expand the fields of the original record inside a new record:

```
let first_record = { name: "Sam", rank: 10 }

{
   ...$first_record
   title: "Mayor"
 }
```

3.10.3 Iterating and Transforming Records

Data processing rarely leaves records untouched. In practice, pipelines must inspect fields, filter out noise, reshape keys, and coerce values into appropriate types. Nushell approaches record manipulation through two distinct mechanisms: transposing records into tabular streams for sequence processing, and invoking native `record` subcommands for direct in-memory transformations.

3.10.3.1 Transposing Records for Sequence Processing

When you need to process record entries as independent observations (such as constructing a formatted log line from every key-value pair), transposing the record yields a two-column table:

```
{ "apples": 543, "bananas": 411, "oranges": 0 } |
  transpose fruit count |
  each {|f| $"We have ($f.count) ($f.fruit)" }
```

By projecting keys and values into columns, transposition brings the full power of table commands like `each`, `where`, and `sort-by` into play. Rebuilding a record from a filtered table, however, historically required an awkward roundtrip:

```
# Legacy transposition roundtrip to filter fields:
{ a: 10, b: 25, c: 5 } |
  transpose k v |
  where v > 20 |
  transpose -r -d
```

3.10.3.2 Native Record Commands

To eliminate transposition overhead and keep pipelines readable, Nushell provides dedicated subcommands under the `record` namespace (`record where`, `record each`, `record walk`, and `record apply`). These commands manipulate records directly without intermediate tabular conversions.

3.10.3.2.1 Filtering Fields with record where

The `record where` command filters record attributes in place. The filtering closure evaluates either `key` or `value`, retaining only the entries that satisfy the predicate:

```
{ apples: 543, bananas: 411, oranges: 0 } | record where value > 0
```

```
╭─────────┬─────╮
│ apples  │ 543 │
│ bananas │ 411 │
╰─────────┴─────╯
```

Filtering by key is just as straightforward, making it easy to prune sensitive tokens or unneeded metadata:

```
{ id: 101, token: "secret_xyz", name: "Alice" } | record where key != "token"
```

```
╭──────┬───────╮
│ id   │ 101   │
│ name │ Alice │
╰──────┴───────╯
```

3.10.3.2.2 Transforming Keys and Values with record each

The `record each` command builds a fresh record by evaluating a closure over every key-value pair. The closure accepts two positional arguments, `k` and `v`, and can return a single-field record `{new_key: new_value}`, a two-element list `[new_key, new_value]`, or `null` to drop the field entirely:

```
# Scale values and convert keys to uppercase in a single pass:
{ apples: 543, bananas: 411 } |
  record each {|k, v| {($k | str uppercase): ($v * 10)}}
```

```
┌─────────┬──────┐
│ APPLES  │ 5430 │
│ BANANAS │ 4110 │
└─────────┴──────┘
```

Returning a two-element list allows swapping keys and values in place:

```
{ a: "x", b: "y" } | record each {|k, v| [$v, $k]}
```

```
┌───┬───┐
│ x │ a │
│ y │ b │
└───┴───┘
```

3.10.3.2.3 Recursive Processing with record walk

Hierarchical, nested records are common when parsing JSON responses or reading configuration trees. Rather than writing ad-hoc recursive functions to traverse arbitrary nesting, `record walk` descends through every branch, passing each leaf `value` and its corresponding `cell-path` into the closure. This makes redaction and deep transformations concise:

```
let payload = {
  user: "sam",
  auth: { token: "secret_123", key: "password" },
  host: "cluster-01"
}

$payload | record walk {|val, path|
  if ($path | to text) =~ '(?i)token|key|password' { '***' } else { $val }
}
```

```
┌──────┬──────────────────────┐
│ user │ sam                  │
│ auth │ {record 2 fields}    │
│ host │ cluster-01           │
└──────┴──────────────────────┘
```

3.10.3.2.4 Declarative Field Conversions with record apply

Preparing raw inputs for statistical analysis often demands explicit type casting across several fields. Instead of chaining repetitive `update` operations, `record apply` accepts a mapping of field names to conversion closures:

```
{ age: "30", score: "95.5", active: "true" } | record apply {
  age: {into int},
```

```
  score: {into float},
  active: {into bool}
}
```

```
╭────────┬────────╮
│ age    │ 30     │
│ score  │ 95.50  │
│ active │ true   │
╰────────┴────────╯
```

3.10.3.2.5 Row Streaming Across Tables

Because Nushell structures tables as streams of records, every command in the `record` family executes row by row across tabular data. Applying declarative conversions to an entire dataset requires nothing more than piping the table directly into `record apply`:

```
[[name age]; [Alice "25"] [Bob "30"]] | record apply { age: {into int} }
```

```
╭───┬───────┬─────╮
│ # │ name  │ age │
├───┼───────┼─────┤
│ 0 │ Alice │  25 │
│ 1 │ Bob   │  30 │
╰───┴───────┴─────╯
```

3.11 Working with Tables

Structured tables represent the primary medium for inspecting tabular datasets in Nushell. Nu provides a comprehensive suite of built-in table manipulation commands, enabling rapid filtering, projection, and reduction to isolate the exact subsets required for analysis.

To begin exploring tabular operations, we obtain an initial table from the filesystem:

```
ls
```

3.11.1 Sorting Data

We sort tables using `sort-by`, passing one or more column names as sorting keys. To order directory entries by file size:

```
ls | sort-by size
```

Tables can be sorted by any column whose data type implements comparison. In this directory listing, we could alternatively sort by `name`, `accessed`, or `modified`.

3.11.2 Selecting Data

We project specific columns or slice specific rows using the `select` command:

```
ls | select name size
```

This isolates the attributes under study. If we wish to inspect only the five smallest files in the directory:

```
ls | sort-by size | first 5
```

Notice how the pipeline orders the table by size first to bring the smallest files to the top, and then applies `first 5` to extract the leading rows.

Rows can also be skipped sequentially:

```
ls | sort-by size | first 5 | skip 2
```

Specific row indices can be extracted directly:

```
ls | sort-by name | select 5
```

3.11.3 Retrieving Data from a Table

Up to this point, our transformations have reshaped tables into smaller tabular subsets. Often, analytical routines require stepping beyond tabular containers to extract raw scalar values or individual column vectors. To extract file names directly as a raw list:

```
ls | get name
```

At first glance, this invocation may resemble our earlier `select` pipeline:

```
ls | select name
```

While their syntax appears related, their computational behavior is fundamentally distinct:

- `select`: Produces a new structured table containing solely the projected columns.
- `get`: Extracts the underlying cell values from the specified column, returning them as a typed list.

3.11.4 Modifying Data in a Table

In addition to selecting subsets, analytical workflows require reshaping tabular contents. This includes appending rows, merging disjoint attributes, and calculating new values. Following Nushell functional design, commands in this section do not mutate tables in place; each invocation yields an entirely new table.

3.11.4.1 Concatenating Tables

We concatenate tabular streams vertically using `append`:

```
let first = [[a b]; [1 2]]

let second = [[a b]; [3 4]]

$first

$second

$first | append $second
```

If column schemas do not match exactly, Nushell dynamically expands the schema, inserting `null` values where corresponding entries are absent:

```
let first = [[a b]; [1 2]]

let second = [[a b]; [3 4]]

let third = [[a c]; [3 4]]

$first | append $second | append $third
```

We can also use the binary append operator `++` as an inline shorthand:

```
$first ++ $second ++ $third
```

3.11.4.2 Merging Tables

We combine columns from two or more tables side-by-side using `merge`:

```
let first = [[a b]; [1 2]]

let second = [[c d]; [3 4]]

$first | merge $second
```

With a third table:

```
let third = [[e f]; [5 6]]
```

All three tables can be joined sequentially:

```
$first | merge $second  | merge $third
```

Alternatively, we can combine an arbitrary collection of tables dynamically using `reduce`:

```
[$first $second $third] | reduce {|elt, acc| $acc | merge $elt }
```

3.11.5 Adding a Column

We add new columns to an existing table using `insert`:

```
{edition: 2024} | save -f edition.json

open edition.json

open edition.json | save -f edition_copia.json
```

We insert a column designated `next_edition` with the scalar value 2025:

```
open edition.json | insert next_edition 2025
```

This rendering can occasionally surprise newcomers because a single-row table displays fields vertically. In Nushell tables, row numbers denote observations, whereas column headers denote variables. Appending an additional record clarifies the tabular grid:

```
open edition.json | insert next_edition 2025 | append {edition: 2021 next_edition: 2023}
```

Re-inspecting the original file confirms that its contents remain unchanged:

```
open edition.json

open edition.json | save -f edition_copia.json
```

Nushell operations are purely functional: they operate on intermediate values in memory without producing unsolicited side effects on disk. This enables safe pipeline composition until the final pipeline stage explicitly commits results using `save`:

```
open edition.json |

  insert next_edition 2025 |

  append {edition: 2021 next_edition: 2023} |

  save -f edition2.json

open edition.json

open edition2.json
```

3.11.6 Updating a Column

Analogous to `insert`, the `update` command replaces the existing contents of a column:

```
open edition.json

open edition.json | save -f edition_copia.json
```

Now, update the edition field:

```
open edition.json | update edition 2025
```

Nushell also supplies `upsert`, which dynamically updates a column if it already exists or inserts it if it is absent.

3.11.7 Moving Columns

We reorder columns across a table using `move`. For instance, to position the `name` column after `size` in an `ls` listing:

```
ls

ls | move name --after size
```

3.11.8 Renaming Columns

Columns can be renamed by passing new header identifiers to `rename`:

```
ls | rename filename filetype filesize filedate
```

3.11.9 Rejecting and Removing Columns

Unwanted attributes can be excluded from a table using `reject`:

```
ls -l

ls -l | reject readonly num_links inode created accessed modified
```

3.12 Navigating and Accessing Structured Data

Given Nushell native support for structured data, navigating and addressing nested structures constitutes one of the most frequent operations in daily data science workflows.

3.12.1 Foundational Concepts

The explanations and examples that follow rely on three foundational structural definitions:

- ***List:*** An ordered sequence of zero or more values of arbitrary types. A list containing zero elements is termed an "empty list."
- ***Record:*** An associative collection of zero or more named keys and their associated values. Values within a record may belong to any Nushell type. A record with zero key-value entries is termed an "empty record."
- ***Nested Data:*** Values stored inside a list, record, or table can be scalar primitives or complex structured data types. Data structures can therefore nest arbitrarily across multiple dimensions:
  - Lists can contain tables, records, and further child lists.
    - ***Table:*** A table is fundamentally a list of records.
  - Records can contain tables, lists, and child records.
    - Consequently, records within a table can themselves hold nested tables, lists, and sub-records.

  Because a table is fundamentally a list of records, any command or syntax valid on lists is also valid on tables. The converse does not hold uniformly: certain commands require tabular column structures and cannot operate on primitive lists.

3.12.2 Cell Paths

A cell path is Nushell primary mechanism for navigating and querying values within structured data. The concept mirrors spreadsheet coordinates, where columns are addressed by name and rows are addressed by numerical index. Segments in a cell path are joined using dot notation (`.`).

3.12.2.1 Records

For a record, a cell path specifies a key name as a `string`.

To query an attribute from a record:

```
let my_record = {
    a: 5
    b: 42
  }

$my_record.b + 5
```

3.12.2.2 Lists

For a list, a cell path specifies the zero-based index of the target element as an `int`:

To query an item from a list (remembering that list indices begin at 0):

```
let scoobies_list = [ Velma Fred Daphne Shaggy Scooby ]

$scoobies_list.3
```

3.12.2.3 Tables

- To access a column, the cell path specifies the column name (`string`).
- To access a row, the cell path specifies the row index (`int`).
- To access a specific cell, combine the column name and row index.

The following examples utilize this sample dataset:

```
let data = [[date                       temps                                  condition ];
            [2022-02-01T14:30:00+05:00,  [ 38.24, 38.50, 37.99, 37.98, 39.10 ],  'summy'   ],
            [2022-02-02T14:30:00+05:00,  [ 35.24, 35.94, 34.91, 35.24, 36.65 ],  'sunny'   ],
            [2022-02-03T14:30:00+05:00,  [ 35.17, 36.67, 34.42, 35.76, 36.52 ],  'cloudy'  ],
            [2022-02-04T14:30:00+05:00,  [ 39.24, 40.94, 39.21, 38.99, 38.80 ],  'rain'    ],
           ]

$data

$data | table -e
```

This table contains meteorological observations across three columns:

1. ***date***: A Nushell `date` timestamp for each observation day.
2. ***temps***: A Nushell `list` of five `float` readings recorded by regional weather sensors.
3. ***condition***: A Nushell `string` describing local atmospheric conditions.

3.12.2.3.1 Example: Accessing a Table Row (Record)

Access the second observation day as an individual record:

```
> $data.1

╭────────────┬──────────────────╮
│ date       │ 2 years ago      │
│            │ ╭───┬────────╮   │
│ temps      │ │ 0 │ 35.24  │   │
│            │ │ 1 │ 35.94  │   │
│            │ │ 2 │ 34.91  │   │
│            │ │ 3 │ 35.24  │   │
│            │ │ 4 │ 36.65  │   │
│            │ ╰───┴────────╯   │
```

```
| condition | sunny         |
└───────────┴───────────────┘
```

3.12.2.3.2 Example: Accessing a Table Column (List)

```
$data.condition
```

3.12.2.3.3 Example: Accessing a Table Cell (Value)

Retrieve the atmospheric condition for the fourth day:

```
$data.condition.3

$data | get condition | get 3

$data | get condition.3
```

3.12.3 Nested Data

Because data structures nest arbitrarily, cell paths can chain multiple names and indices into composite query paths.

3.12.3.1 Example: Accessing Nested Table Data

Retrieve the reading from the second temperature sensor on the third day:

```
$data.temps.2.1
```

Here, index `2` selects the third observation day, and subsequent index `1` isolates the second sensor measurement in the nested list.

3.12.4 Using `get` and `select`

In addition to literal dot notation, Nushell provides specialized pipeline commands that accept cell paths. The two most critical commands are:

- `get`: Functionally equivalent to cell path dot notation, with full support for dynamic variable names and expressions. Like literal cell paths, `get` returns the **underlying value** referenced by the path.
- `select`: Fundamentally different from `get`. Rather than extracting raw scalar values, `select` returns a subset of the **data structure itself**:
  - Applying `select` to a table returns a table of equal or reduced dimensions.

- Applying `select` to a list returns a list of equal or reduced length.
- Applying `select` to a record returns a record with identical or fewer fields.

Continuing with our meteorological dataset:

3.12.4.1 Example: `get` versus `select` on a Table Row

```
$data | get 1

$data | select 1
```

We observe the distinction:

- `get` yields the exact record illustrated in the previous `$data.1` example.
- `select` produces a single-row table complete with column headers and row numbers.

Row indices in a table generated by `select` do not retain their original coordinates; the output table re-indexes its rows beginning at 0.

To preserve original position metadata before slicing, use `enumerate`:

```
$data | enumerate | select 1
```

3.12.4.2 Example: `select` with Multiple Rows and Columns

Because `select` constructs a new table, we can specify multiple columns, row ranges, or combined slices. The following pipeline extracts the date and condition columns from the first two rows:

```
$data | first 2 | select date condition
```

3.12.5 Keys and Column Names with Spaces

When a record key or table column contains spaces or special characters that prevent bareword parsing, enclose the identifier in quotes.

Example:

```
let record_example = {
    "key x":12
    "key y":4
  }

$record_example."key x"

# o
$record_example | get "key x"
```

Quotes are likewise required when key names look like numerical literals.

Example:

```
let record_example = {
    "1": foo
    "2": baz
    "3": far
  }

$record_example."1"
```

In this scenario, do not confuse the string key name with a numeric row index. The first entry has been assigned the string key `"1"`. If this record were transposed into a table using `transpose`, the string key `"1"` would reside at row index `0` (an integer).

3.12.6 Handling Missing Data

3.12.6.1 The Optional Operator

By default, navigating a non-existent cell path triggers a runtime error. To suppress evaluation errors when querying sparse or irregular structures, append `?` to any member of the cell path to mark it as optional:

3.12.6.1.1 Example: The Optional Operator

Using our earlier weather dataset:

```
let cp: cell-path = $.temps?.1

$data | reject temps | get $cp
```

When navigating via an optional cell path, missing attributes evaluate safely to `null` rather than halting pipeline execution.

3.12.6.2 Supplying a Default Value for Missing or Null Data

The `default` command replaces missing or `null` entries with a fallback value across target columns:

```
let missing_value = [{a:1 b:2} {b:1}]

$missing_value

let with_default_value = $missing_value | default 'n/a' a

$with_default_value

$with_default_value.1.a
```

3.12.7 Other Commands for Accessing Structured Data

- `reject`: Acts as the inverse of `select`, dropping specified rows and columns while preserving the remainder.
- `range`: Extracts a continuous slice of rows from a list or table using a range literal.

3.13 In-Terminal File Editing with `ttt`

When working on command-line data science workflows (especially within Docker containers or remote SSH sessions), an ergonomic, terminal-native text editor is essential. The `dsin` container includes **`ttt`** (*Terminal Text Editor*), placed directly in `$env.PATH` and pre-configured with syntax highlighting and intuitive keyboard shortcuts.

We verify the installed version of `ttt` directly from Nushell:

```
ttt -v
```

3.13.1 Basic Usage of `ttt`

To create or edit a Nushell, Python, or R script, or to adjust configuration files inside our working directory (`/data/ch03`), launch `ttt` within an interactive terminal session:

```
ttt mi_script.nu
```

Essential shortcuts and default settings for `ttt` inside `dsin`:

- **`Alt + Q`:** Exit editor.
- **`Ctrl + S`:** Save current file.
- **`Ctrl + F`:** Search text within file.
- **`Ctrl + Z`** / **`Ctrl + Y`:** Undo / Redo edits.

The container environment includes the `black-rain-blur` color theme, 4-space soft tabs, automatic word wrap, line numbering, bracket pair colorization, and hidden-file visibility within the built-in file tree.

Structured scripts can also be authored directly from Nushell pipelines using `save -f`, followed by immediate execution or inspection:

```
"# Script generado en /data/ch03\ndef saludo [nombre: string] {\n  $\"Hola,
 ↪  ($nombre)!\"\n}\nsaludo 'Ciencia de Datos en Nushell'\n" | save -f ejemplo_ttt.nu

nu ejemplo_ttt.nu

nu ejemplo_ttt.nu | save -f saludo_salida.txt
```

```
open saludo_salida.txt
```

3.14 Version Control

This section introduces version control with Git and demonstrates how platforms such as GitHub and GitLab streamline collaboration and reproducibility in data science.

Throughout analytical workflows, code, pipelines, and transformation scripts evolve continuously. Systematically tracking these iterations prevents data loss, simplifies collaboration, and guarantees the scientific reproducibility of empirical findings. This is the domain of **version control**.

A *version control system* records modifications made to a file or repository over time. It allows developers to inspect historical snapshots, compare revisions, identify which change introduced a bug, and restore earlier working states.

Version control provides four critical advantages for data science:

- **Change Tracking:** Exposes the evolution of data pipelines, simplifying bug diagnosis and performance audits.
- **Collaborative Workflows:** Enables multiple analysts to contribute concurrently without overwriting peer contributions.
- **State Restoration:** Allows reverting broken pipelines to verified historical checkpoints.
- **Reproducibility:** Guarantees that analytical code, parameters, and computational artifacts remain consistent over time.

3.14.1 Git: The Essential Tool

Git is a distributed version control system. Unlike centralized predecessors, every clone contains the complete operational history of the project, enabling offline development and rapid branch switching.

Core Git Concepts:

- **Repository:** The directory hosting all project files, configuration assets, and the historical change log (`.git`).
- **Branch:** An independent line of development. Branches allow feature experimentation and bug fixing without compromising the stable `main` branch.
- **Commit:** An immutable snapshot of the project state at a specific point in time, identified by a cryptographic hash and accompanied by an explanatory message.
- **Merge:** The integration of commits from one branch into another.

File Lifecycle States:

- **Modified:** Files containing unsaved local changes not yet staged for inclusion.
- **Staged:** Files marked for inclusion in the upcoming commit snapshot.
- **Committed:** Files safely recorded in the local repository database.

Essential Git Commands:

- **`git init`**: Initializes a new Git repository in the current directory.
- **`git clone <URL>`**: Clones an existing remote repository onto the local machine.
- **`git add <file>`**: Stages modified files for the next commit.
- **`git commit -m "Commit message"`**: Records staged changes as a new commit snapshot.
- **`git push`**: Transmits local commits to a designated remote repository.
- **`git pull`**: Fetches and merges remote changes into the active local branch.
- **`git branch`**: Lists, creates, or deletes development branches.
- **`git checkout <branch>`**: Switches context to the specified branch.
- **`git merge <branch>`**: Combines history from the specified branch into the current branch.

Common Git Workflows:

- **Centralized Workflow:** All collaborators commit directly to a single shared `main` branch. This model is straightforward and suitable for small, tightly-scoped projects.
- **Feature Branch Workflow:** Distinct branches are created for each feature or bug fix. This isolates experimental code and facilitates peer review before integration into `main`.

3.14.2 Cloning and Installing CLI Tools: An Example with ddgr

To examine the real-world cycle of downloading, compiling, and integrating external CLI utilities into the `dsin` container environment, we examine `ddgr`. Written in Python, this utility enables command-line web searches via DuckDuckGo directly from the terminal.

First, navigate to the user home directory (~), create a dedicated source directory, and clone the repository:

```
cd ~

mkdir ~/software

cd ~/software

git clone --depth 1 https://github.com/jarun/ddgr.git

cd ddgr
```

Next, install the tool via `make`. In the `dsin` container, `/home/dsin/bin` is already configured in `$env.PATH`, so we specify `/home/dsin` as the installation prefix:

```
make install PREFIX=/home/dsin

which ddgr
```

Once installed, `ddgr` can be utilized in two complementary ways:

3.14.2.1 1. Interactive Terminal Mode

Run search queries interactively within the console:

```
ddgr como clonar un repositorio github
```

In this mode, navigate result pages using `n` (next page) and `p` (previous page), open links by entering the corresponding number, and exit by pressing `q`.

3.14.2.2 2. Structured Output and Pipeline Integration in Nushell

A primary strength of modern command-line tools is their capacity to emit structured data. Supplying `--json` to `ddgr` allows streaming search results directly into Nushell pipelines for parsing with `from json`:

```
ddgr --json -n 3 "nushell data science" |

  from json |

  select title url
```

Web query results immediately become a structured Nushell table, ready for filtering with `where`, column projection with `select`, or direct serialization into analytical formats.

3.14.3 GitHub and GitLab

GitHub and **GitLab** are cloud-based hosting platforms built on Git that provide issue tracking, code review, continuous integration, and project governance.

Creating a Remote Repository:

1. Register an account on GitHub or GitLab.
2. Initialize a new repository with an appropriate title and description.

Connecting a Local Repository to a Remote Host:

1. In the repository settings on the web platform, copy the remote clone URL.
2. In the terminal, navigate to the root directory of your local project.
3. Link the remote repository via: `git remote add origin <remote_repository_url>`.

Collaborative Mechanisms:

- **Cloning:** Use `git clone <URL>` to pull a complete local copy of an existing project.
- **Forking:** Create an independent copy of an upstream repository under your personal account to test modifications without affecting upstream history.
- **Pull Requests / Merge Requests:** Submit formal review proposals requesting upstream maintainers to integrate your branch changes into their canonical branches.

Platform Features:

- **Issue Tracking:** Organizes bug reports, roadmap milestones, and technical discussions.
- **Wikis:** Dedicated spaces for team documentation and project knowledge bases.
- **Continuous Integration / Continuous Deployment (CI/CD):** Automates test execution, static analysis, container image builds, and deployments on every push.

3.14.4 Best Practices for Data Science Repositories

- **Project Structure:** Organize code, raw datasets, processed data, and documentation into distinct directories.
- **`.gitignore` Rules:** Keep ephemeral artifacts, compiler caches, operating system metadata, and raw datasets out of source control.
- **Meaningful Commit Messages:** Write clear, concise commit messages explaining the intent behind each change.
- **Dependency Declarations:** Document software requirements explicitly using manifest files (`requirements.txt`, `Cargo.toml`, `environment.yml`) or container specifications (`Dockerfile`).

3.14.5 Commit and Push

To publish projects on GitHub or GitLab, register an account. GitHub and GitLab both support public and private repositories.

Authentication is established via an SSH key pair. Generate a new SSH key if one is not already present:

```
ssh-keygen -t rsa
cat ~/.ssh/id_rsa.pub
```

Copy the public key string and register it under the SSH Keys settings in your GitHub or GitLab user profile.

Next, create an empty repository on the web platform and connect your local workspace:

```
git config --global init.defaultBranch main
cd ~/software
mkdir test #o algun nombre seleccionado
cd test
echo "# test" >> README.md
git init
git add README.md
git commit -m "first commit" #mensaje a eleccion
git branch -M main
git remote add origin git@github.com:your_user/your_repo_name.git
git push -u origin main
```

This registers the remote origin and synchronizes local commits.

When files change (for instance, editing `README.md`):

```
nano README.md
```

Stage, commit, and push updates using:

```
git add -A
git status
git commit -am "mensaje apropiado"
git push origin main
```

3.14.6 Contributing to a Repository

To contribute changes to an open-source project via a pull request (PR):

1. **Fork the Upstream Repository:**

- In your web browser, navigate to the target repository.
- Click "Fork" in the top-right corner to establish a personal copy under your account.

2. **Clone the Forked Repository:**

- Copy the SSH clone URL of your fork.
- In the terminal, clone the fork:

```
git clone <copied_https_url>
```

Replace `<copied_https_url>` with the URL copied in the previous step.

3. **Create a Dedicated Feature Branch:**

```
cd <repository_directory>
git checkout -b <your-branch-name>
```

For example: `git checkout -b fix-typo-in-readme`

4. **Implement Changes and Commit:**

- Apply edits to code or documentation.
- Stage and commit changes using `git add` and `git commit`, then push to your fork with `git push origin <your-branch-name>`.

5. **Open the Pull Request:**

- **Navigate to the Upstream Repository:** Open the original repository in your browser.
- **Initiate PR:** Select "Compare & pull request" on the notification banner for your pushed branch, or open "Pull requests" and click "New pull request".
- **Review and Submit:** Review diff changes, provide a concise summary of the modifications in the PR description, and click "Create pull request".

3.14.7 Guidelines for Commit Messages

Conventional commit conventions ensure an informative, structured commit history:

1. Select a conventional prefix:

- *build*: Changes affecting build configurations or external dependencies (e.g., `make`, `npm`, `cargo`).
- *chore*: Maintenance tasks, dependency updates, and configuration adjustments.
- *ci*: Updates to CI/CD pipeline definitions (e.g., GitHub Actions, GitLab CI).
- *docs*: Documentation enhancements and typographical corrections.
- *feat*: Implementation of a new feature or analytical capability.
- *fix*: Bug fixes and corrections.
- *perf*: Performance optimizations.
- *refactor*: Code changes that neither fix a bug nor introduce a feature.
- *style*: Formatting adjustments, whitespace cleanup, and linting fixes.
- *test*: Adding missing test cases or correcting existing test suites.

2. Provide a concise, imperative title.

3. Include a bulleted summary of specific modifications when necessary.

4. Structure the final commit message:

```
<prefix>: <title>

- Item 1
- Item 2
  etc
```

3.14.8 Additional Resources

- **Official Documentation:**
  - Git: https://git-scm.com/docs
  - GitHub: https://docs.github.com/
  - GitLab: https://docs.gitlab.com/
- **Data Science Projects Utilizing Git and GitHub:**
  - TensorFlow
  - scikit-learn
  - Pandas

3.14.9 For Further Exploration

A comprehensive catalogue of command-line tools featured throughout this curriculum is maintained at: https://jeroenjanssens.com/dsatcl/list-of-command-line-tools.

4 Programming and Creating Command-Line Tools

4.1 Programming in Nushell

This section explores Nushell in depth as a general-purpose programming language.

Just as traditional programming languages provide functions for encapsulating reusable logic, Nushell employs custom commands (*custom commands*) for this purpose.

Nushell *aliases* function similarly to aliases in conventional shells, but exist as native constructs within the language rather than mere textual substitutions.

Standard data transformations, from arithmetic evaluation to regular expression matching, are performed using *operators*. Because operations are strictly validated against operand types, Nushell provides immediate syntax and type diagnostics whenever an incompatible operation is attempted.

Intermediate values can be assigned to *variables*. Nushell variables can be immutable (`let`), mutable (`mut`), or parse-time constants (`const`).

Scripts represent the simplest mechanism for organizing code: assemble pipeline statements in a file and execute them directly. Moreover, Nushell scripts can function as standalone executable CLI utilities featuring strongly typed argument signatures via the `main` command.

4.1.1 Operators

Nushell supports the following operators across arithmetic, logic, and string manipulation:

Operator	Description
`+`	Add
`-`	Subtract
`*`	Multiply
`/`	Divide
`//`	Floor division
`mod`	Modulo
`**`	Exponentiation (power)
`==`	Equal
`!=`	Not equal
`<`	Less than
`<=`	Less than or equal to
`>`	Greater than
`>=`	Greater than or equal to
`=~`	Regex match / string contains
`!~`	Inverse regex match / string does not contain
`in`	Value in collection
`not-in`	Value not in collection
`not`	Logical NOT

Operator	Description
`and`	Short-circuit logical AND
`or`	Short-circuit logical OR
`xor`	Logical XOR
`bit-or`	Bitwise OR
`bit-xor`	Bitwise XOR
`bit-and`	Bitwise AND
`bit-shl`	Bitwise shift left
`bit-shr`	Bitwise shift right
`starts-with`	String starts with
`ends-with`	String ends with
`++`	Append lists

Parentheses can be used for subexpression grouping to dictate evaluation precedence or to invoke commands and embed their structured results directly inside an expression.

4.1.1.1 Order of Operations

To inspect operator precedence rules directly from the terminal, run: `help operators | sort-by precedence -r`.

Arranged in descending order of precedence, Nushell evaluates expressions as follows:

- Parentheses (`()`)
- Exponentiation (`**`)
- Multiplication (`*`), Division (`/`), Floor Division (`//`), and Modulo (`mod`)
- Addition (`+`) and Subtraction (`-`)
- Bitwise shifts (`bit-shl`, `bit-shr`)
- Comparison operators (`==`, `!=`, `<`, `>`, `<=`, `>=`), membership tests (`in`, `not-in`, `starts-with`, `ends-with`), regex matching (`=~`, `!~`), and list concatenation (`++`)
- Bitwise AND (`bit-and`)
- Bitwise XOR (`bit-xor`)
- Bitwise OR (`bit-or`)
- Logical AND (`and`)
- Logical XOR (`xor`)
- Logical OR (`or`)
- Assignment operations
- Logical NOT (`not`)

For example:

```
3 * (1 + 2)
```

4.1.1.2 Types

Not all operations make semantic sense across every data type. When attempting an operation between incompatible types, Nushell halts evaluation with an explanatory error diagnostic:

```
> "spam" - 1

Error: nu::parser::unsupported_operation (link)

  × Tipos no coincidentes para la operación.

   ╭─[entry #49:1:1]

 1 | "spam" - 1

   · ──┬── ┬ ┬
   ·   |   | └── int
   ·   |   └── no admite estos valores.
   ·   └── string
   └────

  help: Cambie string o int para que sean los tipos correctos e intente nuevamente.
```

These static typing boundaries may initially seem strict, yet they systematically eliminate unexpected runtime side effects and subtle type coercion bugs common in traditional shells.

4.1.1.3 Regular Expression and Substring Matching Operators

The `=~` and `!~` operators provide an ergonomic syntax for evaluating regular expressions without invoking external utilities. Even without complex regular expressions, they serve as a straightforward mechanism for testing whether a string contains a target substring:

- `string =~ pattern` evaluates to **true** if `string` contains a match for `pattern`, and **false** otherwise.
- `string !~ pattern` evaluates to **false** if `string` contains a match for `pattern`, and **true** otherwise.

For example:

```
"foobarbaz" =~ "bar" # devuelve true

"foobarbaz" !~ "bar" # devuelve false
```

```
ls | where name =~ ^nu # devuelve todos los archivos cuyos nombres comienzan con "nu"
```

4.1.1.4 Case Sensitivity

Nushell operators enforce case sensitivity when comparing text. Several idioms allow performing case-insensitive string evaluations:

1. In regular expression operators, supply the case-insensitive inline flag `(?i)`:

```
"FOO" =~ "foo" # devuelve falso

"FOO" =~ "(?i)foo" # devuelve verdadero
```

2. Pass the `--ignore-case` flag to the `str contains` command:

```
"FOO" | str contains --ignore-case "foo"
```

3. Convert strings to lowercase via `str downcase` prior to evaluation:

```
("FOO" | str downcase) == ("Foo" | str downcase)
```

4.1.1.5 The Spread Operator

Nushell provides a spread operator (`...`) designed to unpack lists and records. It expands compound collections into individual values or key-value attributes wherever multiple arguments or parameters are expected.

The spread operator is valid in three operational contexts:

- Inside list literals.
- Inside record literals.
- Across command invocations.

4.1.1.5.1 In List Literals

Suppose we need to concatenate multiple lists while interspersing individual scalar elements. While this can be achieved by chaining `append` and `prepend`, the spread operator offers significantly cleaner syntax:

```
let dogs = [Spot, Teddy, Tommy]

let cats = ["Mr. Humphrey Montgomery", Kitten]

[
    ...$dogs
    Polly
    ...($cats | each { |elt| $"($elt) \(cat\)" })
```

```
    ...[Porky Bessie]
    ...Nemo
]
```

The following pipeline represents the equivalent transformation constructed with `append`:

```
$dogs |

  append Polly |

  append ($cats | each { |elt| $"($elt) \(cat\)" }) |

  append [Porky Bessie] |

  append ...Nemo
```

We observe that each invocation of `append` allocates a new intermediate list in memory. In this second pipeline, three superfluous intermediate collections are instantiated. The spread operator eliminates these allocations, yielding concrete performance advantages when combining multiple large collections.

Note that the final item in the resulting list above is `"...Nemo"`. Within list literals, the spread operator can only unpack list collections, not bare strings. Consequently, inside list literals, `...` is recognized exclusively before variables (`...$foo`), subexpressions (`...(foo)`), and nested list literals (`...[foo]`).

Furthermore, `...` will not be parsed as a spread operator if whitespace separates the dots from the target expression:

```
[ ... [] ]
```

This parsing constraint prevents ambiguity with standard filesystem paths, such as `mv ... $dir`.

4.1.1.5.2 In Record Literals

Consider a configuration record where additional attributes must be merged dynamically:

```
let my_config = { path: /tmp, limit: 5 }
```

We can derive a new record combining all attributes from `$my_config` alongside new entries using the spread operator. Multiple records can be unpacked simultaneously within a single record literal:

```
{
    ...$my_config,
    users: [alice bob],
    ...{ url: example.com },
```

```
    ...(sys mem)
}
```

Analogous to lists, the spread operator within record literals is recognized exclusively before variables (`...$foo`), subexpressions (`...(foo)`), and nested record literals (`...{foo:bar}`). Whitespace between `...` and the target expression is strictly prohibited.

4.1.1.5.3 In Command Invocations

Arguments can also be unpacked dynamically into command calls, provided the receiving command declares a `rest` parameter or corresponds to an external system binary.

Here is an example custom command featuring a `rest` parameter:

```
def foo [ --flag req opt? ...args ] { [$flag, $req, $opt, $args] | to nuon }
```

This command declares a named flag (`--flag`), a required positional parameter (`req`), an optional positional parameter (`opt?`), and a variable rest parameter (`args`).

To pass an existing list into `args`, spread it just as you would inside a list literal. Identical parsing rules apply: the spread operator is recognized solely before variables, subexpressions, and list literals, with no whitespace permitted:

```
foo "bar" "baz" ...[1 2 3] # Con ..., los números se tratan como argumentos separados

foo "bar" "baz" [1 2 3] # Sin ..., [1 2 3] se trata como un solo argumento
```

A frequent application of the spread operator arises when writing wrapper commands that forward variable arguments directly to an underlying command:

```
def bar [ ...args ] { foo --flag "bar" "baz" ...$args }

bar 1 2 3
```

Multiple collections can be unpacked within a single invocation alongside individual positional arguments:

```
foo "bar" "baz" 1 ...[2 3] 4 5 ...(6..9 | take 2) last
```

Named flags can appear after a spread argument, just as they can follow conventional `rest` parameters:

```
foo "bar" "baz" 1 ...[2 3] --flag 4
```

If a spread argument precedes an optional positional parameter, that optional parameter is treated as omitted:

```
foo "bar" ...[1 2] "not opt"
```

4.1.2 Variables in Scripts and CLI Tools

The conceptual variable model of Nushell, naming conventions, and the architectural distinction between `let`, `mut`, and `const` were established in Chapter 3. When designing reusable CLI tools and functions, the governing principle is to prioritize immutable variables with `let` and continuous pipeline streams, reserving `mut` strictly for imperative algorithms where local state accumulation is indispensable and `const` for static dependencies evaluated during compilation (`source`, `use`).

4.1.3 Custom Commands

Nushell ability to compose extended pipelines provides granular control over data and system state. However, typing lengthy pipelines repeatedly quickly becomes inefficient. Reusable pipelines should be encapsulated into callable routines.

This is where custom commands come in.

Here is a basic custom command definition:

```
def greet [name:string] {
  ['hello' $name]
}
```

This definition declares the command `greet`, accepting a single parameter `name`. Following the parameter signature is the code block executed when the command is invoked. Upon invocation, the argument supplied for `name` is bound to `$name` and made accessible throughout the block.

To execute the custom command, invoke it identically to built-in commands:

```
greet "world"
```

> The value yielded by the final expression of a command automatically becomes its return value. In this case, the command returns a list containing the string `'hello'` and `$name`. To suppress output, pipe to `ignore` or append `null`: `['hello' $name] | ignore`. Note also that most filesystem commands, such as `save` or `cd`, always return `null`.
>
> To produce a formatted string directly, use string interpolation syntax to embed `$name`:

```
def greet [name:string] {
  $"hello ($name)"
}

greet nushell
```

4.1.4 Command Names

In Nushell, command names are plain strings. Valid identifiers include: `greet`, `get-size`, `my-command123`, `my command`, and `emoji_cmd` (including supported Unicode symbols).

> Idiomatic Nushell convention separates words in command names with hyphens (`-`) for enhanced readability. For instance, prefer `get-size` over `getsize` or `get_size`.

4.1.5 Subcommands

Subcommands are defined naturally by separating command components with spaces. For example, to extend `str` with a new subcommand, declare a command whose name begins with `"str "`:‘

```
def "str mycommand" [] {
  "hello"
}
```

The custom command can then be invoked identically to native subcommands of `str`:

```
def "str mycommand" [] { "hello" }

str mycommand
```

Commands containing whitespace in their primary names are declared following the identical syntax:

```
def "custom command" [] {
  "this is a custom command with a space in the name!"
}
```

4.1.6 Parameter Types

When declaring custom commands, parameters can be annotated with explicit static types. For instance, the earlier example can be annotated as:

```
def greet [name: string] {
  $"hello ($name)"
}
```

Parameter type annotations are optional. When omitted, Nushell assigns the type `any`. When a static type is declared, Nushell validates the argument at parse time before execution begins.

For instance, suppose we require an `int` parameter instead:

```
def greet [name: int] {
  $"hello ($name)"
}
```

```
greet world
```

If we attempt to supply an incompatible argument, Nushell halts evaluation with an explicit type mismatch error:

```
error: Type Error
  ┌─ shell:6:7
  │
5 │ greet world
  │       ^^^^^ Expected int
```

This mechanism ensures callers supply data of the expected shape. For a full inventory of supported Nushell types (`string`, `int`, `float`, `filesize`, `datetime`, `table`, etc.), refer to Table 1 in Chapter 3.

4.1.7 Parameters with Default Values

To make a parameter optional while specifying a fallback value, assign a default directly in the parameter list:

```
def greet [age, name = "nushell"] {
  $"hello ($name) + ($age)"
}
```

The command can then be called with or without an explicit argument:

```
def greet [name = "nushell"] { $"hello ($name)" }

greet

greet world
```

Default values can be combined seamlessly with static type annotations:

```
def congratulate [age: int = 18] {
  $"Happy birthday! You are ($age) years old now!"
}
```

When you need to determine whether an optional argument was explicitly provided rather than relying on a static fallback, use optional positional parameters instead.

4.1.8 Optional Positional Parameters

By default, positional parameters are strictly required. Omitting a required positional argument produces a syntax error:

```
  × Missing required positional argument.
```

```
   ╭─[entry #23:1:1]
 1 | greet
   .       ▲
   .       └── missing name
   └────
  help: Usage: greet <name>
```

A positional parameter can be marked optional by appending a question mark (`?`) to its identifier:

```
def greet [name?: string] {
  $"hello ($name)"
}

greet

greet world
```

Marking a parameter optional does not alter its identifier within the command body; it remains accessible via `$name`, without the trailing `?`.

When an optional parameter is omitted by the caller, its value inside the command body resolves to `null`. This allows branching on whether an argument was provided:

```
def greet [name?: string] {
  if ($name == null) {
    "hello, I don't know your name!"
  } else {
    $"hello ($name)"
  }
}

greet
```

If the objective is simply to substitute a fallback value when an argument is missing, declaring a default value is typically more concise.

When combining required and optional positional parameters, required parameters must precede optional ones in the signature.

4.1.9 Flags

In addition to positional arguments, commands can accept named parameters by defining flags.

For example:

```
def greet [
  name: string
  --age: int
] {
  [$name $age]
}
```

In this definition of `greet`, we declare the positional parameter `name` alongside an optional named flag `age`. Callers can optionally provide the `--age` flag during invocation.

The command can be invoked as:

```
greet world --age 10
```

Or:

```
greet --age 10 world
```

Or by omitting the flag entirely:

```
greet world
```

Flags can also define shorthand single-character aliases, allowing both concise interactive use and self-documenting scripts.

We extend the previous command to provide `-a` as a shorthand for `--age`:

```
def greet [
  name: string
  --age (-a): int
] {
  [$name $age]
}
```

> *Note:* Flags are bound inside the command body using their full name; the example above accesses `$age` rather than `$a`.

The updated command can now be invoked using the shorthand flag:

```
greet -a 10 hello
```

Flags can also serve as boolean switches. Their presence or absence indicates a true or false condition. Extending the definition:

```
def greet [
  name: string
  --age (-a): int
```

```
  --twice
] {
  if $twice {
    [$name $name $age $age]
  } else {
    [$name $age]
  }
}
```

The command can now be executed with the switch:

```
greet -a 10 --twice hello
```

Or without it:

```
greet -a 10 hello
```

Boolean values can also be passed explicitly to enable or disable switches:

```
greet -a 10 --twice=false hello

greet -a 10 --twice=true hello
```

Flag names may contain hyphens. When accessed inside the command body, hyphens are translated to underscores:

```
def greet [
  name: string
  --age (-a): int
  --two-times
] {
  if $two_times {
    [$name $name $age $age]
  } else {
    [$name $age]
  }
}
```

4.1.10 Rest Parameters

When a command needs to accept an arbitrary number of positional arguments, declare a `rest` parameter using the `...` syntax:

```
def greet [...name: string] {
  print "hello all:"
  for $n in $name {
    print $n
```

```
  }
}

greet earth mars jupiter venus
```

The `greet` command can now be called with any number of arguments, including none. All supplied values are gathered into `$name` as a typed list.

Rest parameters can follow standard positional parameters:

```
def greet [vip: string, ...name: string] {
  print $"hello to our VIP ($vip)"
  print "and hello to everybody else:"
  for $n in $name {
    print $n
  }
}

greet moon earth mars jupiter venus
```

To pass an existing list into a `rest` parameter, use the spread operator (`...`):

```
let planets = [earth mars jupiter venus]

greet moon ...$planets
```

4.1.11 Documenting a Command

To streamline usage and maintainability, Nushell allows embedding detailed documentation for commands, parameters, and flags directly in the source definition.

Additionally, Nushell provides an annotation system utilizing `@` prefixes placed immediately before `def`:

- **`@category <name>`:** Classifies the command within the system catalog (visible in `help commands`).
- **`@search-terms <terms...>`:** Registers keywords enabling semantic discovery via `help --find`.
- **`@example "description" { code } --result outcome`:** Embeds executable test examples with expected outputs, rendered automatically in the examples section of `help`.

The following listing illustrates a fully documented command definition:

```
# Comando de saludo personalizado que ilustra anotaciones y documentación
@category utilidades

@search-terms saludo bienvenida mensaje
```

```
@example "Saludar indicando la edad" { greet "Carlos" --age 28 } --result ["Carlos", 28]

def greet [
  name: string      # Nombre de la persona a saludar
  --age (-a): int   # Edad opcional de la persona
] {
  [$name $age]
}
```

Once defined, running `help greet` generates an exhaustive technical reference sheet automatically:

```
Comando de saludo personalizado que ilustra anotaciones y documentación

Search terms: saludo, bienvenida, mensaje

Usage:
  > greet <name> {flags}

Flags:
  -a, --age <int>: Edad opcional de la persona
  -h, --help: Muestra este mensaje de ayuda

Parameters:
  name <string>: Nombre de la persona a saludar

Examples:
  Saludar indicando la edad
  > greet "Carlos" --age 28
  ╭───┬────────╮
  │ 0 │ Carlos │
  │ 1 │     28 │
  ╰───┴────────╯
```

Running `help greet` on a standard command similarly provides formatted guidance:

```
A greeting command that can greet the caller

Usage:
  > greet <name> {flags}

Parameters:
  <name> The name of the person to greet

Flags:
  -h, --help: Display this help message
  -a, --age <integer>: The age of the person
```

4.1.12 Closures

A **closure** in Nushell is an anonymous, first-class block of executable code capable of receiving arguments, consuming pipeline input (`$in`), and capturing variables from its enclosing lexical scope. Closures can be stored in variables, passed into higher-order commands, and evaluated dynamically on demand.

4.1.12.1 Syntax and Parameters

Closures are enclosed in curly braces `{ ... }`. When declaring formal parameters, specify them between vertical bars at the beginning of the block (`{ |param1, param2| ... }`), optionally including static type annotations:

```
# Cierre con un parámetro tipado
let duplicar = { |x: int| $x * 2 }

# Cierre con dos parámetros
let ponderar = { |val: float, peso: float| $val * $peso }
```

4.1.12.2 Dynamic Invocation with `do`

To execute a closure stored in a variable, invoke it using the built-in `do` command:

```
do $duplicar 21

do $ponderar 4.5 0.8
```

4.1.12.3 Higher-Order Functions in Pipelines

In data science workflows, closures achieve their primary utility when processing tables and collections via higher-order commands such as `each`, `where`, `filter`, `reduce`, and `par-each`:

```
# Transformación elemento a elemento con each
[1 2 3 4 5] | each { |n| $n ** 2 }

# Filtrado condicional con where
[10 25 3 48 7] | where { |val| $val > 20 }
```

When a closure omits explicit parameters between `|...|`, it can access incoming pipeline records directly via the special variable `$in`:

```
[5 12 18] | each { $in * 10 }
```

4.1.12.4 Lexical Scoping and Immutability

A closure can capture immutable variables (`let` or `const`) from its defining scope. However, by architectural design to ensure thread safety and deterministic concurrency across parallel pipelines, **closures in Nushell are strictly prohibited from capturing and mutating mutable variables (`mut`)**. Any attempt to mutate an exterior mutable variable from within a closure produces an immediate parse error (`nu::shell::capture_of_mutable_var`).

4.1.13 Pipeline Output

Custom commands stream their output downstream just like native commands. For example:

```
ls | get name
```

We wrap `ls` inside a custom command:

```
def my-ls [] { ls }
```

We can consume the output of this custom command identically to `ls`:

```
my-ls | get name
```

This design allows constructing modular command pipelines that pass structured tables between stages. Note that Nushell does not require explicit return statements; instead, pipelines generate continuous data streams that connect cleanly into subsequent pipeline stages.

4.1.14 Pipeline Input

Custom commands can also accept incoming pipeline streams. Incoming data is passed automatically to the command body.

Consider the following command that doubles each element in an incoming list:

```
def double [] {
  each { |elt| 2 * $elt }
}
```

It can be consumed directly from a pipeline:

```
[1 2 3] | double
```

Pipeline input can also be captured explicitly for multi-stage processing using `$in`:

```
def nullify [...cols] {
  let start = $in
  $cols | reduce --fold $start { |col, df|
    $df | upsert $col null
  }
```

```
}
```

4.1.15 Control Flow

Nushell provides robust control flow constructs to govern conditional branching and iterative execution.

4.1.15.1 Conditionals

Conditional expressions execute code blocks based on boolean conditions.

In Nushell, conditional constructs are expressions that evaluate to concrete values, unlike statement-based control flow in conventional shells. This means conditionals can be embedded directly within pipelines:

```
'foo' | if $in == 'foo' { 1 } else { 0 } | $in + 2
```

4.1.15.1.1 `if`

`if` evaluates conditional branches, executing code based on the truth value of one or more expressions. For example:

```
let x = 5

if $x > 0 { 'positivo' }
```

This returns `'positive'` when the condition is `true` (`$x > 0`) and `null` when `false` (`$x <= 0`).

An `else` branch can be appended to provide an explicit fallback value when the condition evaluates to `false`:

```
if $x > 0 { 'positivo' } else { 'no-positivo' }
```

This returns `'positive'` when `$x > 0` and `'non-positive'` otherwise.

Multiple conditions can be chained using `else if`:

```
if $x > 0 { 'positive' } else if $x == 0 { 'zero' } else { "negative" }
```

Or formatted across multiple lines:

```
if $x > 0 {
   'positive'
} else if $x == 0 {
   'zero'
} else {
   "negativo"
```

```
}
```

When the first condition is `true` (`$x > 0`), it returns `'positive'`; when the second condition holds (`$x == 0`), it returns `'zero'`; otherwise, it returns `'negative'`.

4.1.15.1.2 `match`

`match` performs pattern matching against an incoming value, branching execution to the matching arm. It can unpack structured compound types such as lists and records.

In its simplest form, `match` functions analogously to a `switch` statement in other programming languages. It compares the target value against each arm before `=>`, executing the block corresponding to the first match:

```
match 2 {
    1 => 'one',
    2 => {
        let w = 'w'
        't' + $w + 'o'
    },
    3 => 'three',
    4 => 'four'
}
```

Arms can return single values or full expression blocks.

Wildcard Arm

A wildcard arm (_) matches any value not caught by earlier arms, serving as a catch-all branch:

```
let foo = match 7 {
    1 => 'one',
    2 => 'two',
    3 => 'three',
    _ => 'other number'
}

$foo
```

Because `match` is an expression, its result can be assigned directly to a variable.

Pattern Matching

Compound values such as records and lists can be destructured using pattern matching. Variable bindings in the pattern capture corresponding components for use within the arm expression:

```
let foo = { name: 'bar', count: 7 }
```

```
match $foo {
    { name: 'bar', count: $it } => ($it + 3),
    { name: _, count: $it } => ($it + 7),
    _ => 1
}
```

The wildcard _ in the second arm matches any record containing fields `name` and `count`, regardless of the value stored in `name`.

Guards

Guards add boolean conditions to pattern arms using `if`, ensuring an arm matches only when both the structural pattern and the guard condition are satisfied:

```
let foo = { name: 'bar', count: 7 }

match $foo {
    { name: 'bar', count: $it } if $it < 5 => ($it + 3),
    { name: 'bar', count: $it } if $it >= 5 => ($it + 7),
    _ => 1
}
```

4.1.15.2 Loops

Loop constructs repeatedly execute a block of code.

4.1.15.2.1 Loops versus Functional Iterators

Loop commands share functionality with higher-order iteration commands like `each` and `where`. Often, identical transformations can be accomplished using either paradigm:

```
mut result = []

for $it in [1 2 3] {
   $result = ($result | append ($it + 1))
}

$result

[1 2 3] | each { $in + 1 }
```

While imperative loops may feel familiar to programmers arriving from other languages, the idiomatic Nushell standard is to favor functional pipeline commands that apply closures. This preference stems from a fundamental architectural limitation of loops.

Disadvantages of Loops

The primary disadvantage of loops is that they are statements rather than expressions. Commands like `each` are expressions that invariably evaluate to structured output, whereas loop statements return nothing.

Consequently, loops cannot be used with immutable variables. Without a mutable accumulator variable initialized prior to the loop, it is impossible for `for` to return an accumulated collection.

Furthermore, loop statements cannot be embedded inside Nushell pipelines. Attempting to pipe into or out of a loop triggers a syntax error:

```
[1 2 3] | for x in $in { $x + 1 } | $in ++ [5 6 7]

Error: nu::parser::unexpected_keyword

  × Statement used in pipeline.

   ╭─[entry #5:1:1]

 1 │ [1 2 3] | for x in $in { $x + 1 } | $in ++ [5 6 7]

   .           ─┬─

   .            └── not allowed in pipeline

   ╰────

  help: 'for' keyword is not allowed in pipeline. Use 'for' by itself, outside of a pipeline.
```

Because Nushell is fundamentally pipeline-centric, expression commands like `each` integrate far more naturally into data transformations.

When Loops are Appropriate

Why do loops exist if expression commands are preferred? One reason is that closures, such as those passed to `each`, cannot mutate variables in their enclosing scope. Attempting to modify an exterior mutable variable from within a closure produces an error:

```
mut foo = []

[1 2 3] | each { $foo = ($foo | append ($in + 1)) }

Error: nu::parser::expected_keyword

  × Capture of mutable variable.

   ╭─[entry #8:1:1]

 1 │ [1 2 3] | each { $foo = ($foo | append ($in + 1)) }
```

```
  .                ──┬─
  .                  └── capture of mutable variable
  └────
```

While a closure can modify environment variables locally within its own execution scope, those changes do not persist outside. Loops, however, execute standard blocks rather than closures, allowing them to mutate exterior state or update environment variables across the broader scope:

```
mut result = []

for $it in [1 2 3] { $result = ($result | append ($it + 1)) }

$result
```

4.1.15.2.2 `for`

`for` iterates over collections, such as ranges, lists, or tables:

```
for x in [1 2 3] { $x * $x | print }
```

Expression Alternatives to `for`

- `each`
- `par-each`
- `where`
- `reduce`

4.1.15.2.3 `while`

`while` repeatedly executes a block as long as its condition evaluates to `true`:

```
mut x = 0;

while $x < 10 { $x = $x + 1 }

$x
```

Expression Alternatives to `while`

- `take until`
- `take while`
- `skip until`
- `skip while`

4.1.15.2.4 `loop`

`loop` repeats a block indefinitely. It is typically controlled via `break` and is useful for continuous monitoring routines or interactive prompts:

```
mut x = 0; loop { if $x > 10 { break }; $x = $x + 1 }; $x
```

4.1.15.2.5 `break`

`break` terminates loop execution immediately, transferring control to the statement following the loop block:

```
for x in 1..10 { if $x > 3 { break }; print $x }
```

4.1.15.2.6 `continue`

`continue` halts the current iteration, skipping remaining statements in the loop body and proceeding to the next iteration:

```
mut x = -1

while $x <= 6 {
    $x = $x + 1;
    if $x mod 3 == 0 {
        continue
    }
    print $x
}
```

4.1.15.2.7 Errors

`error make`

`error make` raises a runtime error that halts execution up through the call stack until caught by a `try` block or terminating the script with a diagnostic report:

```
error make { msg: 'Some error info' }; print 'unprinted'

Error:   × Some error info

   ╭─[entry #9:1:1]

 1 │ print 'printed'; error make { msg: 'Some error info' }; print 'unprinted'

   .                  ──────┬─────

   .                        └── originates from here
```

```
  └──┘
```

The record passed to `error make` provides metadata for upstream error handlers or terminal diagnostics.

`try`

`try` catches runtime errors occurring within its block, resuming execution cleanly after the block:

```
try { error make { msg: 'Some error info' }}; print 'Resuming'
```

This includes intercepting built-in evaluation errors such as division by zero:

```
try { 1 / 0 }; print 'Resuming'
```

The resulting value is `nothing` if an error occurs, or the return value of the block if execution succeeds.

An optional `catch` block executes custom recovery logic when an error is trapped:

```
try { 1 / 0 } catch { 'An error happened!' } | $in ++ ' And now I am resuming.'
```

The `catch` block is bypassed entirely when execution succeeds without errors.

```
try { 1 / 0 } catch { |e| $e.msg }
```

4.1.15.2.8 Other Statements

`return`

`return` terminates a custom command or closure early, returning the specified value immediately. While rarely required because the final expression in a block automatically forms its return value, `return` is useful for early exit conditions:

```
def 'positive-check' [it] {
    if $it > 0 {
        return 'positive'
    };

    'non-positive'
}
```

```
positive-check 3

positive-check (-3)

let positive_check = {|elt| if $elt > 0 { return 'positive' }; 'non-positive' }
```

```
do $positive_check 3

do $positive_check (-3)
```

4.2 Converting One-Liners into Command-Line Tools

Throughout this textbook, we construct numerous compact, single-line pipelines (*one-liners*). When a one-liner is needed repeatedly across different files or with variable arguments, it is best converted into an **executable command-line tool**.

In `/data/ch04`, we have the complete unabridged text of *Alice's Adventures in Wonderland* (`alice.txt`) alongside a stopword list (`stopwords`):

```
ls
```

First, we demonstrate how to compute the 10 most frequent words from `alice.txt` in Nushell (both unfiltered and filtering stopwords), rendering the output to the terminal and saving it as a CSV file:

```
open --raw alice.txt |

  str lowercase |

  parse --regex '(?P<word>[a-z]{2,})' |

  get word |

  uniq -c |

  rename word count |

  sort-by count --reverse |

  first 10
```

Next, we filter out stopwords using `stopwords` and save the resulting table both to the console and to `top10_alice.csv`:

```
let sw = open --raw stopwords | lines | each { str trim | str lowercase }

let top10 = open --raw alice.txt
    | str lowercase
    | parse --regex '(?P<word>[a-z]{2,})'
    | get word
    | where { |w| not ($w in $sw) }
```

```
    | uniq -c
    | rename word count
    | sort-by count --reverse
    | first 10

$top10

$top10 | save -f top10_alice.csv
```

To convert this one-liner into a reusable, parameterized command-line tool in Nushell, we follow **six systematic steps**:

1. Create the script file (`top-words-1.nu`).
2. Grant execution permissions (`chmod u+x`).
3. Define the shebang directive (`#!/usr/bin/env -S nu --stdin`).
4. Decouple fixed input to read dynamically from standard input (`$in`).
5. Add typed arguments and flags via `def main [...]`.
6. Place the binary in `$env.PATH`.

4.2.1 Step 1: Create the File

In `/data/ch04`, we save the initial pipeline within `top-words-1.nu` (which can be edited interactively via `ttt top-words-1.nu`):

```
# Código de top-words-1.nu
let sw = if ("stopwords" | path exists) {
    open --raw stopwords | lines | each { str trim | str lowercase }
} else {
    []
}

open --raw alice.txt
| str lowercase
| parse --regex "(?P<word>[a-z]{2,})"
| get word
| where { |w| not ($w in $sw) }
| uniq -c
| rename word count
| sort-by count --reverse
| first 10
```

To inspect script source code directly from the terminal, Nushell prints raw text via `open --raw <file>`. Additionally, for syntax highlighting and line numbers, the environment includes **`bat`** (`bat top-words-1.nu`). For custom Nushell commands and functions loaded in memory (such as commands from `dsin_tools.nu`), the environment provides **`view-code`** (`view-code <command>`),

which displays syntax-highlighted source code directly from the shell runtime. We utilize `bat` for script inspections and `view-code` for custom command inspections throughout this text:

```
open --raw top-words-1.nu

bat top-words-1.nu

nu top-words-1.nu

nu top-words-1.nu | save -f salida_paso1.csv
```

4.2.2 Step 2: Grant Execution Permissions

We copy `top-words-1.nu` to `top-words-2.nu` and grant execution permissions to the owner via `chmod u+x`:

```
# Código de top-words-2.nu (idéntico en contenido a top-words-1.nu)
let sw = if ("stopwords" | path exists) {
    open --raw stopwords | lines | each { str trim | str lowercase }
} else {
    []
}

open --raw alice.txt
| str lowercase
| parse --regex "(?P<word>[a-z]{2,})"
| get word
| where { |w| not ($w in $sw) }
| uniq -c
| rename word count
| sort-by count --reverse
| first 10
```

We inspect the updated permissions using `chmod` and `ls`:

```
cp top-words-1.nu top-words-2.nu

chmod u+x top-words-2.nu

ls -l top-words-1.nu top-words-2.nu | select name mode size
```

In the `mode` column (e.g., `-rwxr-xr-x`), the leading `-` denotes a regular file, followed by permission triads for read (`r`), write (`w`), and execute (`x`) across user (`u`), group (`g`), and others (`o`).

4.2.3 Step 3: Define the Shebang

The shebang directive (`#!/usr/bin/env -S nu --stdin`) on the first line instructs the operating system kernel which interpreter to invoke when calling the script directly as an executable binary (`./top-words-3.nu`):

```
# Código de top-words-3.nu
#!/usr/bin/env -S nu --stdin
let sw = if ("stopwords" | path exists) {
    open --raw stopwords | lines | each { str trim | str lowercase }
} else {
    []
}

open --raw alice.txt
| str lowercase
| parse --regex "(?P<word>[a-z]{2,})"
| get word
| where { |w| not ($w in $sw) }
| uniq -c
| rename word count
| sort-by count --reverse
| first 10
```

We inspect the script with `bat` and execute it directly:

```
bat top-words-3.nu

./top-words-3.nu

./top-words-3.nu | save -f salida_paso3.json
```

4.2.4 Step 4: Remove Fixed Input

By replacing the hardcoded reference to `alice.txt` with `$in` inside a `main` function in `top-words-4.nu`, the tool decouples from specific files and processes any stream received via a pipeline (`|`):

```
# Código de top-words-4.nu
#!/usr/bin/env -S nu --stdin
def main [] {
    $in
    | str lowercase
    | parse --regex "(?P<word>[a-z]{2,})"
    | get word
    | uniq -c
    | sort-by count --reverse
    | first 10
```

```
}
```

Inspect with `bat` and test pipeline streaming from the terminal:

```
bat top-words-4.nu

open --raw alice.txt | ./top-words-4.nu

open --raw alice.txt | ./top-words-4.nu | save -f salida_paso4.json
```

4.2.5 Step 5: Add Typed Arguments and Options (`def main`)

By declaring formal parameters in `def main [num_words: int = 10, --stopwords-file (-s): path = "stopwords"]` inside `top-words-5.nu`, Nushell enforces static type checking, assigns default values, and auto-generates the `--help` manual:

```
# Código de top-words-5.nu
#!/usr/bin/env -S nu --stdin

# Extrae las palabras más frecuentes de un texto excluyendo stopwords
def main [
    num_words: int = 10 # Número de palabras frecuentes a retornar
    --stopwords-file (-s): path = "stopwords" # Archivo de palabras vacías
] {
    let input_text = $in
    let sw = if ($stopwords_file | path exists) {
        open --raw $stopwords_file | lines | each { str trim | str lowercase }
    } else {
        []
    }
    $input_text
    | str lowercase
    | parse --regex "(?P<word>[a-z]{2,})"
    | get word
    | where { |w| not ($w in $sw) }
    | uniq -c
    | rename word count
    | sort-by count --reverse
    | first $num_words
}
```

Inspect with `bat` and execute while passing positional parameters:

```
bat top-words-5.nu

open --raw alice.txt | ./top-words-5.nu 8
```

```
open --raw alice.txt | ./top-words-5.nu 8 | save -f top8_alice.json
```

4.2.6 Step 6: Extend `$env.PATH`

To invoke the utility from any directory without specifying `./`, install the script into a directory registered in `$env.PATH` (such as `/home/dsin/bin`):

```
cp top-words-5.nu /home/dsin/bin/top-words-nu

chmod +x /home/dsin/bin/top-words-nu

open --raw alice.txt | top-words-nu 5

open --raw alice.txt | top-words-nu 5 | save -f top5_alice.json
```

4.3 Creating Command-Line Tools with Python and R

Because the command line is language-agnostic, scripts written in **Python** (`#!/usr/bin/env python3`) and **R** (`#!/usr/bin/env Rscript`) integrate smoothly into Nushell pipelines. Following our established methodology, we inspect the `.py` or `.R` script inside the chapter directory, invoke it from Nushell, parse its standard output into a structured table via `from csv`, and persist the results.

4.3.1 Porting the Tool to Python and R

In `/data/ch04`, we have complete implementations in Python (`top-words.py`) and R (`top-words.R`). First, inspect `top-words.py` and run it from Nushell:

```
bat top-words.py

open --raw alice.txt | python3 top-words.py 5 | from csv

open --raw alice.txt | python3 top-words.py 5 | from csv | save -f top_words_py.csv
```

Now inspect the R implementation (`top-words.R`), execute it against `alice.txt`, parse its output into a Nushell table, and save it to `top_words_r.csv`:

```
bat top-words.R

open --raw alice.txt | Rscript top-words.R 5 | from csv

open --raw alice.txt | Rscript top-words.R 5 | from csv | save -f top_words_r.csv
```

4.3.2 Streaming Data from `stdin`

When input arrives as a line-by-line stream, both Python and R can process lines incrementally as they arrive over `stdin`. In `/data/ch04`, `fizzbuzz.py` and `fizzbuzz.R` implement the classic FizzBuzz challenge (printing `"fizz"` for multiples of 3, `"buzz"` for multiples of 5, and `"fizzbuzz"` for multiples of 15):

```
bat fizzbuzz.py

1..15 | to text | python3 fizzbuzz.py | lines | wrap resultado_py

1..15 | to text | python3 fizzbuzz.py | lines | wrap resultado_py | save -f fizzbuzz_py.csv
```

And identically with the R script `fizzbuzz.R`:

```
bat fizzbuzz.R

1..15 | to text | Rscript fizzbuzz.R | lines | wrap resultado_r

1..15 | to text | Rscript fizzbuzz.R | lines | wrap resultado_r | save -f fizzbuzz_r.csv
```

5 Data Acquisition

This chapter addresses the initial phase of the OSEMN framework: **data acquisition** (*Obtain*). In computational data science, data originates across diverse media: local filesystem artifacts, HTTP and HTTPS endpoints, compressed tarballs (`.tar.gz`, `.zip`), Microsoft Excel workbooks (`.xlsx`), embedded relational SQLite databases (`.db`), RESTful web services, and continuous real-time event streams.

We begin by navigating into `/data/ch05`, which is mounted by default within the `dsin` container:

```
cd /data/ch05
```

From this point forward, commands in this chapter execute within `/data/ch05`. Let us inspect the available files:

```
ls | sort-by name
```

5.1 Copying Local Files into the Docker Container

While the `dsin` container image bundles all textbook datasets within `/data/ch03` through `/data/ch09`, project work frequently requires transferring external files between the host workstation and the container:

- **Mounting a local directory when launching the container:** Execute `docker run --rm -it -v "$(pwd):/data/my_data" ghcr.io/kurokirasama/dsin:latest`.
- **Copying files into or out of a running container:** Execute `docker cp local_file.csv <container_id>:/data/ch05/` from the host operating system.

5.2 Downloading Data from the Internet with `http get`

Nushell incorporates a native HTTP client suite (`http get`, `http post`, `http head`, `http put`, `http delete`), eliminating dependencies on legacy utilities such as `curl` or `wget`. A major advantage of `http get` is that whenever the remote server returns structured payloads (such as JSON), **Nushell automatically parses the response into an navigable table or record**.

5.2.1 Fetching JSON Endpoints and Web Pages

In `/data/ch05`, we have `urls.json` alongside sample JSON publications. When querying a REST endpoint or reading local JSON responses like `posts_sample.json`, we can manipulate the resulting table and write it directly to disk via `save -f`:

```
let response = open posts_sample.json

$response | first 3 | select userId id title

$response | save -f posts.json
```

```
open posts.json | first 3 | select id title
```

When fetching or inspecting HTML documents (such as `MolinosFriesland.html` in `/data/ch05`), we can review leading lines using `open --raw` and persist a processed extract:

```
open --raw MolinosFriesland.html | lines | first 10

open --raw MolinosFriesland.html | lines | first 10 | to text | save -f molinos_encabezado.txt
```

5.2.2 Fetching from Multiple URLs and Dispatching `http post` Requests

The `/data/ch05/urls.json` file contains a collection of target endpoints under the `urls` key:

```
open urls.json

open urls.json | get urls

open urls.json | get urls | to text | save -f lista_urls.txt
```

In environments with active Internet connectivity, we can iterate over multiple URLs concurrently with `each { |u| http get $u }` or transmit structured JSON payloads to a remote server using `http post`:

```
open urls.json | get urls | each { |u| http get $u } | save -f respuestas_api.json

{
  my_payload: {
    title: "foo"
    body: "bar"
    userId: 1
  }
} | get my_payload | to json | http post --content-type "application/json"
 ↪  https://jsonplaceholder.typicode.com/posts $in
```

5.2.3 Following Redirects and Inspecting HTTP Headers

When resolving shortened URLs (such as `bit.ly`) or diagnosing web services, it is helpful to inspect solely HTTP headers and status codes (such as `200 OK`, `301 Moved Permanently`, `303 See Other`, `403 Forbidden`, `404 Not Found`) using `http head` or the `--full` flag on `http get`:

```
http head https://www.wikipedia.org | first 8
```

5.3 Decompressing and Extracting Archives

When datasets involve substantial volume or bundle hundreds of disparate files, they are routinely distributed as compressed archives such as `.tar.gz`, `.zip`, or `.7z`.

The `/data/ch05` directory provides the compressed archive `logs.tar.gz`. Before proceeding with extraction, it is prudent practice to inspect its table of contents. This can be accomplished using standard `tar` (`-tzf`) or the universal **`7z`** utility (passing the `l` sub-command to list contents):

```
tar -tzf logs.tar.gz | lines | first 10

tar -tzf logs.tar.gz | lines | save -f indice_logs.txt

7z l logs.tar.gz
```

Because the files packaged within `logs.tar.gz` lack an enclosing root directory, we create an extraction target directory (`logs_extraidos`) prior to unpacking. First, extraction using `tar` with `-C`:

```
rm -rf logs_extraidos

mkdir logs_extraidos

tar -xzf logs.tar.gz -C logs_extraidos
```

Equivalently, archive extraction can be performed using **`7z`** (specifying `-o` to direct output to the target folder):

```
rm -rf logs_7z

mkdir logs_7z

7z x logs.tar.gz -so | 7z x -si -ttar -o./logs_7z
```

Once extraction finishes, we consolidate file sizes and line counts across all extracted log files into a structured Nushell table, writing the resulting summary to `resumen_logs.csv`:

```
let resumen_logs = ls logs_extraidos
    | each { |f|
        {
            archivo: ($f.name | path basename)
            bytes: $f.size
            lineas: (open --raw $f.name | lines | length)
        }
    }
    | sort-by archivo

$resumen_logs | first 5
```

```
$resumen_logs | save -f resumen_logs.csv
```

5.4 Working with Spreadsheets (Microsoft Excel) and CSV Files

The CSV (*Comma-Separated Values*) format stores tabular records separated by newlines with column fields delimited by commas. The `/data/ch05` directory includes three variations of a Teenage Mutant Ninja Turtles dataset (`tmnt-with-header.csv`, `tmnt-basic.csv`, and `tmnt-missing-newline.csv`).

In Nushell, `open` automatically detects headers in well-formed files such as `tmnt-with-header.csv`:

```
open tmnt-with-header.csv

open tmnt-with-header.csv | save -f tmnt_con_encabezado.json
```

When a CSV file **lacks a header row** (as in `tmnt-missing-newline.csv`), we invoke `open --raw` paired with `from csv --noheaders`, assigning descriptive column names using `rename` and writing the normalized table to `tmnt_normalizado.csv`:

```
open --raw tmnt-missing-newline.csv | from csv --noheaders | rename nombre apodo color arma

open --raw tmnt-missing-newline.csv | from csv --noheaders | rename nombre apodo color arma |
 ↪  save -f tmnt_normalizado.csv
```

For working with **Microsoft Excel spreadsheets (`.xlsx` and `.xls`)**, Nushell requires no external translation utilities: `open` parses the workbook directly, returning a record whose keys correspond to individual worksheets. Let us inspect `cars.xlsx` in `/data/ch05`:

```
open cars.xlsx | columns
```

We observe that `cars.xlsx` contains the worksheet `autos`. We extract this sheet via `get autos`, inspect its leading rows, and export it as a CSV file (`cars_exportado.csv`):

```
open cars.xlsx | get autos | first 5 | select make "body.style" horsepower "city.mpg" price

open cars.xlsx | get autos | save -f cars.csv

open cars.xlsx | get autos | save -f cars_exportado.csv
```

To filter records matching brand `toyota` using a regular expression anchor (`^toyota$`), we project key attributes and persist the filtered subset to `cars_toyota.csv`:

```
open cars.csv | where make =~ '^toyota$' | select make "body.style" "engine.size" horsepower
 ↪  price | first 8
```

```
open cars.csv | where make =~ '^toyota$' | save -f cars_toyota.csv
```

5.5 Querying Relational Databases (SQLite)

In `/data/ch05`, we have an embedded SQLite relational database (`r-datasets.db`) storing classic R benchmark datasets. In Nushell, database tables can be audited directly using `open r-datasets.db | columns`:

```
open r-datasets.db | columns | first 10
```

To execute SQL `SELECT` queries directly against `r-datasets.db`, we employ the native **`query db`** command, rendering the result as a Nushell table and saving it to `mtcars_ordenado.csv`:

```
open r-datasets.db | query db "SELECT row_names, mpg, cyl, hp, wt FROM mtcars ORDER BY mpg DESC"
 ↪  | first 8

open r-datasets.db | query db "SELECT row_names, mpg, cyl, hp, wt FROM mtcars ORDER BY mpg DESC"
 ↪  | save -f mtcars_ordenado.csv
```

We can likewise execute SQL aggregate calculations directly via `query db`, saving the grouped summary to `mtcars_por_cilindros.csv`:

```
let resumen_cyl = open r-datasets.db
    | query db "SELECT cyl, COUNT(*) AS n_autos, ROUND(AVG(mpg), 2) AS mpg_promedio,
     ↪  ROUND(AVG(hp), 1) AS hp_promedio FROM mtcars GROUP BY cyl ORDER BY cyl"

$resumen_cyl

$resumen_cyl | save -f mtcars_por_cilindros.csv
```

5.6 Consuming Web APIs

Web APIs (*Application Programming Interfaces*) return structured data (predominantly in JSON format) ready for immediate ingestion and transformation within Nushell pipelines. In contemporary data science, analytical inputs rarely reside exclusively on local disks; they are routinely queried from remote platforms via authenticated HTTP requests or real-time event streams.

Below, we examine key authentication and four practical case studies in Nushell, structured as reusable commands and modules ready for pipeline integration:

1. **Authenticated URL Construction with News API:** Secure endpoint generation using structured records and `url join`.
2. **Geocoding with the Google Maps API:** Bidirectional translation between postal addresses and geographic coordinates (latitude and longitude).

3. **Automated Translation with Google Translate:** Dynamic text translation within analytical pipelines.
4. **Structured Web Search with Ollama Search:** Live querying and ingestion of web search results directly into Nushell tables.
5. **Real-Time Event Streams (*Streaming*):** Continuous event processing using Wikimedia EventStreams.

All modules and example scripts in this section reside in `/data/ch05` (with duplicate references in `/data/ch04`):

```
cd /data/ch05
```

5.6.1 API Key Authentication and URL Construction

Many web APIs (such as *News API* or *Google Maps*) require an authentication key stored securely in a local credential file (for example, `.secret/newsapi.org_apikey` in `/data/ch05`). In Nushell, we read the secret key using `open --raw` and construct the authenticated endpoint cleanly with `url join`:

```
let api_key = open --raw .secret/newsapi.org_apikey | str trim

let news_url = {
    scheme: "https"
    host: "newsapi.org"
    path: "/v2/everything"
    params: {
        q: "linux"
        apiKey: $api_key
    }
} | url join

$news_url

$news_url | save -f news_endpoint.txt
```

5.6.2 Geocoding with the Google Maps API

According to the official Google Maps Geocoding API documentation, the service converts postal addresses into geographic coordinates (forward geocoding) or translates latitude and longitude coordinates into human-readable formatted addresses (reverse geocoding) via HTTP GET requests.

5.6.2.1 Obtaining an API Access Key

Using Google Maps Platform services requires an *API Key* provisioned through the Google Cloud Console:

1. Sign in to Google Cloud Console with your Google account.
2. Create a new project or select an existing project from the top navigation bar.
3. Navigate to **APIs & Services** → **Library**.
4. Search for **Geocoding API** and click **Enable**.
5. Navigate to **APIs & Services** → **Credentials**.
6. Click **Create Credentials** → **API Key**. Copy the generated key and store it in a secure location.

5.6.2.2 Anatomy of the Request and url join

To query the API, we construct a URL specifying scheme `https`, host `maps.googleapis.com`, path `/maps/api/geocode/json`, and query parameters (`address` or `latlng`, alongside `key`).

Rather than concatenating strings manually (which introduces escaping errors with whitespace and special symbols), Nushell provides **`url join`**, which composes and URL-encodes a clean request string from a structured record:

```
let mapsAPIkey = "the_api_key"

let address = "Paseo Yugoslavo 176, Valparaiso, Chile"

let url_record = {
    scheme: "https"
    host: "maps.googleapis.com"
    path: "/maps/api/geocode/json"
    params: {
        address: $address
        key: $mapsAPIkey
    }
}

$url_record | url join

$url_record | save -f consulta_maps.json
```

Equivalently, the inverse command **`url parse`** deconstructs any existing URL string into a structured Nushell record:

```
open consulta_maps.json | url join | url parse
```

Dispatching the request with `http get $url` returns the parsed JSON payload, from which we can extract coordinates via `results.geometry.location`, returning `lat` and `lng`:

```
# Estructura de respuesta devuelta por Google Geocoding API
let sample_response = {
    status: "OK"
    results: [
```

```
        {
            formatted_address: "Paseo Yugoslavo 176, Valparaíso, Región de Valparaíso, Chile"
            geometry: {
                location: { lat: -33.0400281, lng: -71.6289843 }
            }
        }
    ]
}

$sample_response.results | select formatted_address

$sample_response.results.geometry.location

$sample_response.results.geometry.location | save -f coordenadas.json
```

5.6.2.3 Implementing the maps.nu Module

We encapsulate these operations in the module `/data/ch05/maps.nu` (also accessible in `/data/ch04/maps.nu`). Its complete source code is as follows:

```
# Módulo de geocodificación con la API de Google Maps

export def loc-from-address [
    address: string
    --api-key (-k): string = "the_api_key"
] {
    let url = {
        scheme: "https"
        host: "maps.googleapis.com"
        path: "/maps/api/geocode/json"
        params: {
            address: $address
            key: $api_key
        }
    } | url join

    http get $url | get results.geometry.location
}

export def address-from-loc [
    latitude: number
    longitude: number
    --api-key (-k): string = "the_api_key"
] {
    let url = {
        scheme: "https"
```

```
        host: "maps.googleapis.com"
        path: "/maps/api/geocode/json"
        params: {
            latlng: $"($latitude),($longitude)"
            key: $api_key
        }
    } | url join

    let response = http get $url
    if $response.status != "OK" {
        error make -u {msg: "address not found!"}
    }
    $response.results.0.formatted_address
}
```

To utilize the module, import it into your session via `use maps.nu *`:

```
use maps.nu *

loc-from-address "Paseo Yugoslavo 176, Valparaiso, Chile"

address-from-loc -33.0400281 -71.6289843
```

5.6.3 Automated Translation with Google Translate

Our second case study addresses translating text attributes and labels within data pipelines. Following the Google Cloud Translation API documentation, the service dynamically translates text across dozens of languages via HTTP requests.

5.6.3.1 Obtaining an API Access Key

1. Inside the Google Cloud Console, navigate to your project.
2. Go to **APIs & Services** → **Library**.
3. Locate **Cloud Translation API** and click **Enable**.
4. You may reuse the API key created for Google Maps or provision a dedicated key in **APIs & Services** → **Credentials** → **Create Credentials** → **API Key**.

5.6.3.2 Implementing the translate.nu Module

The `/data/ch05/translate.nu` module accepts text either via pipeline input (`$in`) or as a positional argument (`text`), offering optional parameters for source language (`--source`, defaulting to auto-detection) and destination language (`--destination`, defaulting to Spanish). If a commercial API key is not supplied, the script falls back to Google Translate's direct web endpoint, enabling immediate local experimentation:

```
# Módulo de traducción con Google Translate

export def trans [
    text?: string                       # Texto a traducir (o desde el pipeline $in)
    --source (-s): string = "auto"      # Idioma de origen (por defecto auto-detección)
    --destination (-d): string = "es"   # Idioma de destino (por defecto español)
    --api-key (-k): string = ""         # Clave opcional de Google Cloud Translation API
] {
    let input_text = if ($text | is-empty) { $in } else { $text }
    if ($input_text | is-empty) {
        error make -u { msg: "Debe proporcionar un texto para traducir (como argumento o mediante
        ↪  tubería)." }
    }

    if ($api_key | is-not-empty) {
        let url = {
            scheme: "https"
            host: "translation.googleapis.com"
            path: "/language/translate/v2"
            params: {
                key: $api_key
                q: $input_text
                source: (if $source == "auto" { "" } else { $source })
                target: $destination
                format: "text"
            }
        } | url join

        let response = http post $url {}
        $response.data.translations.0.translatedText
    } else {
        let url = {
            scheme: "https"
            host: "translate.googleapis.com"
            path: "/translate_a/single"
            params: {
                client: "gtx"
                sl: $source
                tl: $destination
                dt: "t"
                q: $input_text
            }
        } | url join

        http get $url | get 0.0.0
    }
```

```
}
```

Import the `trans` command and apply it within your pipelines:

```
use translate.nu *

"Data science workflows at the command line" | trans

trans "Structured data pipelines in Nushell" --destination "fr"
```

5.6.4 Structured Web Search and Ingestion with the Ollama Search API

Our third case study integrates live web searches directly into typed Nushell tables using the Ollama web search API. This enables analytical pipelines to enrich datasets with live internet information without resorting to graphical web browsers or unstructured HTML dumps.

5.6.4.1 Obtaining an Access Token (API Key)

1. Register or sign in at ollama.com.
2. Navigate to your account settings at ollama.com/settings/keys.
3. Generate and copy your personal API token (*Ollama API Key*).
4. In Nushell, export the environment variable `$env.OLLAMA_API_KEY = "your_key"` in your configuration or pass it explicitly via `--api-key` (`-k`).

5.6.4.2 Implementing the ollama_search.nu Module

The `/data/ch05/ollama_search.nu` module issues an HTTP POST request with the `Authorization: Bearer <token>` header, receives the structured JSON response, and returns search results directly as a typed Nushell table (or as Markdown text when passing `--md`):

```
# Módulo de búsqueda web estructurada mediante Ollama Search API

export def ollama-search [
    query: string                       # Consulta de búsqueda web
    --max-results (-n): int = 5         # Cantidad máxima de resultados a devolver
    --api-key (-k): string = ""         # Token de autenticación de Ollama
    --md (-m)                           # Formatear la salida como texto Markdown
    --verbose (-v)                      # Mostrar mensajes informativos
] {
    let token = if ($api_key | is-not-empty) {
        $api_key
    } else {
        try { $env.OLLAMA_API_KEY } catch { "" }
    }
```

```
if ($token | is-empty) {
    error make -u { msg: "Se requiere un token de Ollama. Use --api-key (-k) o defina
     ↪  $env.OLLAMA_API_KEY." }
}

let url = "https://ollama.com/api/web_search"
let body = { query: $query, max_results: $max_results }
let headers = {
    Authorization: $"Bearer ($token)"
    "Content-Type": "application/json"
}

if $verbose { print $"Consultando la web mediante Ollama Search API: ($query)" }

let response = try {
    http post -t application/json $url -H $headers $body -e
} catch { |e|
    error make -u { msg: $"Error de conexión con Ollama Search API: ($e.msg)" }
}

let data = if ($response | describe) == "string" {
    $response | from json
} else {
    $response
}

if ($data | get -o error | is-not-empty) {
    error make -u { msg: $"Ollama API Error: ($data.error)" }
}

if $md {
    return (
        $data.results
        | default []
        | each { |item|
            [
                $"# ($item.title)"
                $"link: ($item.url)"
                ""
                $item.content
            ] | str join (char nl)
        }
        | str join $"(char nl)(char nl)"
    )
}
```

```
    $data.results | default []
}
```

We can use this tool to feed downstream pipelines with commands like `select`, `where`, or `save`:

```
use ollama_search.nu *

ollama-search "nushell data science pipelines" -n 3 | select title url

ollama-search "command line data science" --md | save -f resultados_busqueda.md
```

5.6.5 Real-Time Streaming APIs

Certain web services stream real-time events continuously over persistent connections, such as the Wikimedia *EventStreams* endpoint (`https://stream.wikimedia.org/v2/stream/recentchange`). In `/data/ch05`, we provide a captured stream sample in `wikimedia-stream-sample`.

In Nushell, we filter lines prefixed with `data:`, strip the prefix via `str replace`, parse each line into a record with `from json`, and isolate revisions originating on English Wikipedia (`en.wikipedia.org`), displaying the table on screen and saving it to `wikimedia_edits_en.csv`:

```
let edits_en = open --raw wikimedia-stream-sample
    | lines
    | where ($it | str starts-with "data: ")
    | each { |line| $line | str replace "^data: " "" | from json }
    | where ($it.type? == "edit") and ($it.server_name? == "en.wikipedia.org")
    | select -o title user bot timestamp

$edits_en | first 10

$edits_en | save -f wikimedia_edits_en.csv
```

6 Data Cleaning and Transformation

This chapter explores the second stage of the OSEMN framework: **data cleaning and transformation** (*Scrub*). Raw data acquired from files, databases, or web endpoints is rarely organized for immediate modeling. It frequently contains missing entries, extraneous delimiters, inconsistent character encodings, unstructured free text, or unnormalized schemas.

We begin by entering the `/data/ch06` working directory:

```
cd /data/ch06
```

From this point forward, all transformations in this chapter execute within `/data/ch06`. Let us inspect the available files:

```
ls | sort-by name
```

6.1 Transformations in a Data Pipeline

A central advantage of command-line data science is the ability to stream transformations progressively through chained pipes (`|`). In Nushell, transformations operate on strongly typed in-memory values rather than unparsed byte streams.

Consider generating a FizzBuzz evaluation sequence from 1 to 100, restructuring it into a typed table, and persisting it to `fb.csv`:

```
1..100 | to text | python3 fizzbuzz.py | save -f fb.seq

let conteo_fb = open --raw fb.seq
    | lines
    | where ($it =~ "fizz|buzz")
    | uniq -c
    | rename value count
    | sort-by count --reverse

$conteo_fb

$conteo_fb | save -f fb.csv
```

6.2 Plain Text

When analytical inputs arrive as unstructured raw text (such as raw log entries, exported documents, or scraped book transcripts), Nushell provides specialized operators to filter lines, extract token fields, and normalize text without needing legacy tools like `sed` or `awk`.

6.2.1 Filtering Lines

The file `alice.txt` contains the full text of *Alice's Adventures in Wonderland*. We can inspect and partition text subsets into files:

```
open --raw lines | lines
```

- **First 3 lines (`first 3`) and last 3 lines (`last 3`):**

```
open --raw lines | lines | first 3

open --raw lines | lines | last 3

open --raw lines | lines | first 3 | to text | save -f lines_primeras3.txt
```

- **Skip the first 3 lines (`skip 3`), and slice lines 4, 5, and 6 (`slice 3..5`):**

```
open --raw lines | lines | skip 3

open --raw lines | lines | drop 3

open --raw lines | lines | slice 3..5

open --raw lines | lines | slice 3..5 | to text | save -f lines_4_a_6.txt
```

- **Extract strictly even- or odd-positioned lines by pairing `enumerate` with the modulo operator (`mod`):**

```
open --raw lines | lines | enumerate | where ($it.index mod 2) == 0 | get item

open --raw lines | lines | enumerate | where ($it.index mod 2) == 0 | get item | to text | save
↪  -f lines_impares.txt
```

6.2.1.1 Pattern-Based Filtering

To isolate lines containing specific syntactic patterns (such as roman-numeral chapter headers `CHAPTER I`, `CHAPTER II`), we filter lines with regular expression anchors (`=~ '^CHAPTER [IVXLCDM]+'`). To remove empty lines, we exclude blank strings via `str trim` and `where`:

```
open --raw alice.txt | lines | where ($it =~ '(?i)^chapter')

open --raw alice.txt | lines | where ($it =~ '^CHAPTER (.*)\. The')

open --raw alice.txt | lines | where ($it =~ '^CHAPTER (.*)\. The') | to text | save -f
↪  capitulos_the.txt

open --raw alice.txt | lines | where ($it | str trim | is-not-empty) | length
```

6.2.1.2 Random Sampling

When analyzing large text corpora, inspecting a random sample is often preferable to reading sequential head lines. We shuffle line arrays using **`shuffle`**, combined with **`first`**:

```
1..1000 | each { |i| $"Line ($i)" } | shuffle | first 10

1..1000 | each { |i| $"Line ($i)" } | shuffle | first 10 | to text | save -f muestra_10_lineas.txt
```

6.2.2 Extracting Values

To extract structured attributes from unstructured text, the **`parse --regex`** command applies named capturing groups (`(?P<group_name>pattern)`), converting matching patterns directly into typed table columns.

Here, we parse roman numeral chapter numbers and chapter titles from `alice.txt`, persisting the structured catalog to `capitulos_alice.csv`:

```
let caps = open --raw alice.txt
    | lines
    | where ($it =~ '^CHAPTER ')
    | parse --regex '^CHAPTER (?P<numero>[IVXLCDM]+)\.\s+(?P<titulo>.*)$'

$caps

$caps | save -f capitulos_alice.csv
```

Similarly, to construct a dataset of words containing both the letter “a” and the letter “e”, we tokenize words via regular expressions, filter by length and character membership, deduplicate counts, and save the result to `palabras_a_e.csv`:

```
let palabras_ae = open --raw alice.txt
    | str lowercase
    | parse --regex '(?P<palabra>[a-z]{2,})'
    | get palabra
    | where ($it =~ '^a.*e$')
    | uniq
    | sort
    | wrap palabra

$palabras_ae | first 15

$palabras_ae | save -f palabras_a_e.csv
```

6.2.3 Replacing and Stripping Values

For character-level and substring replacements, Nushell provides `str replace`. When batch character transliteration is required (such as converting between case variants or translating specific character sets), the native Linux utility **`tr`** can be invoked directly on byte streams:

```
"hello world!" | str replace -a " " "_"

"hello world!" | str replace -a --regex '[ !]' ''

"hallo wêreld!" | str uppercase

"helló világ" | str uppercase

"  hello     world!  " | str trim | str replace -a --regex '\s+' ' ' | str replace "hello" "bye"

"  hello     world!  " | str trim | str replace -a --regex '\s+' ' ' | str replace "hello" "bye"
 ↪  | save -f saludo_limpio.txt
```

6.3 Parsing Command Output

Many command-line utilities emit tabular information as formatted plain text rather than structured objects (for example, `df -h` or `docker ps`). Nushell provides three complementary strategies to parse text streams into structured, typed tables:

1. **`detect columns`:** Automatically identifies column boundaries based on spacing between text headings.
2. **`from ssv`:** Parses space-separated values (*space-separated values*), with support for multi-space alignment.
3. **`parse`:** Extracts column fields according to an explicit template string.

Let us examine the output of `df -h`:

```
df -h

df -h | str replace "Mounted on" Mounted_On | detect columns
```

For command output formatted with tabular column headers like `df`, `detect columns` is typically the most direct parsing strategy:

```
df -h | str replace "Mounted on" Mounted_On | from ssv --aligned-columns --minimum-spaces 1
```

Alternatively, **`from ssv --aligned-columns`** parses columns where values are aligned beneath space-padded headers:

```
'nu = "0.116.1" # A new type of shell
```

```
ttt = "1.8.0" # Terminal text editor' | lines | parse '{crate_name} = "{version}" #{description}'
 ↪  | update description { str trim }
```

Finally, when output contains irregular spacing or complex formatting, **parse** provides explicit template extraction:

```
open tips.csv | update day { str uppercase } | first 5

open tips.csv | update day { str uppercase } | save -f tips_day_upper.csv
```

6.4 CSV and Structured Tables

The CSV (*Comma-Separated Values*) format is the lingua franca of tabular datasets. In `/data/ch06`, we have `tips.csv`, which documents restaurant transactions, customer gratuities, party sizes, and customer demographics.

6.4.1 Headers, Bodies, and Columns

When opening `tips.csv` with `open`, Nushell parses column types and headers automatically. We can transform column attributes using `upsert` or `update`, such as capitalizing day names and saving to `tips_day_upper.csv`:

```
rm -f tips_temp.db

open tips.csv | into sqlite tips_temp.db

open tips_temp.db | query db "SELECT bill, tip, day, time, size FROM main WHERE bill > 40.0 AND
 ↪  day LIKE 'S%'"

open tips_temp.db | query db "SELECT bill, tip, day, time, size FROM main WHERE bill > 40.0 AND
 ↪  day LIKE 'S%'" | save -f tips_sql_resultado.csv
```

6.4.2 Executing SQL Queries on CSV

Nushell allows running SQL queries directly on tabular CSV streams using **query db**. Below, we aggregate tips by customer sex and smoking preference, saving the summary to `tips_sql_resultado.csv`:

```
open iris.csv | select sepal_length petal_length sepal_width petal_width | first 5

open iris.csv | reject species | first 5

open iris.csv | select sepal_length petal_length sepal_width petal_width | save -f
 ↪  iris_columnas_reordenadas.csv
```

6.4.3 Extracting and Reordering Columns

To project specific variables into an analytical dataset, we pass target columns to `select`. When we need to remove unwanted variables while retaining all others, we use `reject`:

```
open tips.csv | where size >= 5 | first 6

open tips.csv | where bill > 40.0 and day =~ '^S'

open tips.csv | where bill > 40.0 and day =~ '^S' | save -f tips_altos_finde.csv
```

6.4.4 Filtering Rows

Filtering rows in Nushell is performed with `where`, passing boolean conditions evaluated per row:

```
open names.csv

open names-comma.csv
```

6.4.5 Combining Columns and Quoted Delimiter Robustness

We can concatenate text columns into composite identifiers (such as combining `first_name` and `last_name` into `full_name`) using `insert` with string interpolation:

```
let nombres_limpios = open names-comma.csv
    | insert full_name { |row| $"($row.first_name) ($row.last_name)" }
    | select id full_name born

$nombres_limpios

$nombres_limpios | save -f names_completos.csv
```

6.4.6 Combining Multiple CSV Files

6.4.6.1 Horizontal Merging (`merge`)

To join two tables by appending columns horizontally on a row-by-row index, use `merge`. For example, from `tips.csv`, we generate two sub-tables (`bills.csv` and `tips.csv`) and merge them column-wise:

```
open tips.csv | select bill tip | save -f bills.csv

open tips.csv | select day time | save -f datetime.csv

open tips.csv | select sex smoker size | save -f customers.csv

let unido_horizontal = open bills.csv | merge (open customers.csv) | merge (open datetime.csv)
```

```
$unido_horizontal | first 5

$unido_horizontal | save -f tips_horizontal.csv
```

6.4.6.2 Relational Joins (`join`)

When datasets share a key identifier column, relational joins are performed using the **`join`** command.

In `/data/ch06`, two related CSV files illustrate relational joins: `iris.csv` (150 morphological observations referencing `species_id`) and `irismeta.csv` (taxonomic metadata mapping `species_id` to `species_name`):

```
open irismeta.csv

let iris_enriquecido = open iris.csv
    | join (open irismeta.csv) species
    | select sepal_length sepal_width species usda_id

$iris_enriquecido | slice 48..53

$iris_enriquecido | save -f iris_usda.csv
```

6.5 Working with XML/HTML and JSON

In `/data/ch06`, we examine `users.json` alongside Wikipedia extracts `wiki.html`, `table.html`, and `table.json`.

Rather than attempting to manipulate structured JSON using stream text tools like `sed` (which risks corrupting nested data or inadvertently altering key names), Nushell loads `users.json` directly into typed memory structures. We select desired attributes from the `results` table, rename `gender` to `sex`, and persist the cleaned dataset to both JSON and CSV:

```
let usuarios = open users.json
    | get results
    | select gender email nat phone
    | rename sex email nat phone

$usuarios | first 5

$usuarios | save -f users_sex.csv
```

Similarly, from the HTML table converted to JSON in `table.json` (listing national boundary-to-area ratios from Wikipedia), we extract tabular rows and persist the structured dataset to

`countries_from_json.csv`:

```
open countries.csv | first 10

open countries.csv | first 10 | save -f countries_top10.csv
```

6.6 The Nushell Polars Plugin

Polars is a lightning-fast DataFrame library implemented in Rust and built upon the Apache Arrow columnar memory format. It provides a declarative query engine that optimizes computation graphs automatically, enabling high-throughput data processing via lazy evaluation and multi-core parallelism.

6.6.1 Use Cases

- Processing datasets that exceed available RAM or make in-memory native shell table manipulation impractical.
- High-performance analytical operations on large Apache Parquet archives.
- Vectorized, parallel execution of complex grouping, filtering, and aggregation pipelines.

6.6.2 Foundational Concepts

6.6.2.1 Polars DataFrames Versus Nushell Tables

- **Nushell Tables:** Strongly typed, in-memory records designed for interactive terminal exploration. Users manipulate rows and columns using commands such as `select`, `where`, and `update`.
- **Polars DataFrames:** Columnar, highly optimized structures backed by Apache Arrow memory buffers. They support lazy evaluation query plans, SIMD vectorization, and multi-threaded execution. Nushell tables convert seamlessly to Polars DataFrames via `polars into-df`, and back to native Nushell tables using `polars into-nu`.

6.6.3 Basic Usage

To use `nu_plugin_polars`, data must first be ingested into a `DataFrame`. The Polars plugin supports diverse file formats, including CSV, JSON, Parquet, and IPC.

6.6.3.1 Data Ingestion

In this walkthrough, we analyze video game sales records from `vgsales.csv`. We verify the local dataset in `/data/ch06`:

```
ls vgsales.csv
```

Source: https://raw.githubusercontent.com/rfordatascience/tidytuesday/master/data/2019/2019-07-30/video_games.csv

We ingest `vgsales.csv` into an in-memory Polars DataFrame:

```
let df = open vgsales.csv | polars into-df

$df | polars first 5 | polars collect

$df | describe
```

This creates a Polars `DataFrame` ready for high-performance analytics.

> If `polars open` encounters path resolution issues in certain environments, load the file into a DataFrame using `open vgsales.csv | polars into-df`.

6.6.3.2 Data Exploration

Once loaded, we inspect DataFrame properties using Polars exploration commands. To examine column names and inferred data types, query its schema:

```
$df | polars schema
```

Notice that string attributes default to `object` type, and missing values appear as typed `null` entries in fields like `metascore`.

6.6.3.3 Type Casting and Schema Enforcement

Polars infers types from initial rows, but analytical workflows frequently require strict schema definitions to avoid downstream type errors. To explicitly cast `number` to `i64` and `price` to `f64` during ingestion, pass a schema definition via `--schema`:

```
let df = open vgsales.csv
  | polars into-df --schema {
      number: i64,
      game: str,
      release_date: str,
      price: str,
      owners: str,
      developer: str,
      publisher: str,
      average_playtime: str,
      median_playtime: str,
      metascore: str
    }
```

> When missing values (`NA`) are present in raw data, define numerical columns temporarily as `string` and cast them to numeric types following cleansing steps.

```
let df = $df
  | collect
  | polars with-column [
    (polars col price
        | polars str-replace-all --pattern "Free" --replace "0"
        | polars cast f64
    ),
    (polars col average_playtime | polars cast i64),
    (polars col median_playtime | polars cast i64),
    (polars col metascore | polars cast i64)
  ]

$df | polars schema

$df | polars summary | polars collect

$df | polars summary | polars select descriptor "price (f64)" "average_playtime (i64)"
 ↪  "median_playtime (i64)" "metascore (i64)" | polars collect

$df | polars first 5 | polars collect
```

In this transformation:

- `collect`: Materializes an internal stream into an in-memory DataFrame before evaluating expressions.
- `with-column`: Appends derived columns or modifies existing series using vectorized Polars expressions.
- `str-slice`: Extracts text substrings.
- `as-date`: Parses formatted text strings into temporal date values.

6.6.4 Data Manipulation

6.6.4.1 Column Selection

To project specific columns from a DataFrame, use `polars select`. For example, to isolate `game`, `price`, and `metascore`:

```
$df | polars select game price metascore
```

Polars outputs the following evaluation structure:

```
┌────────────────┬──────────────────────────────────────────────────────┐
| plan           | SELECT [col("game"), col("price"), col("metascore")] |
|                |   DF ["number", "game", "release_date", "price",     |
|                |   ...]; PROJECT */10 COLUMNS                         |
| optimized_plan | DF ["number", "game", "release_date", "price", ...]; |
|                |   PROJECT["game", "price", "metascore"] 3/10 COLUMNS |
```

```
└──────────────────┴──────────────────────────────────────────────────────────┘
```

This output represents the Polars **query plan**, illustrating how the query optimizer structures execution to maximize computational efficiency. To materialize the actual dataset records, invoke `polars collect`:

```
$df | polars select game price metascore | polars collect
```

6.6.4.2 Row Filtering

To filter rows based on boolean conditions, use `polars filter-with`. For example, to filter rows where the price is greater than 10 and the metascore exceeds 80:

```
#búsqueda Insensible a Mayúsculas/Minúsculas
let filter_condition = polars col developer | polars contains "(?i)Action"

$df | polars filter $filter_condition | polars collect

#búsqueda Sensible a Mayúsculas/Minúsculas
let filter_condition = polars col developer | polars contains "Action"

$df | polars filter $filter_condition | polars collect

#eliminar na in price
let filter_condition = polars col price | polars is-not-null

$df | polars filter $filter_condition | polars collect
```

Alternatively, to drop all rows containing null values across any column, use `polars drop-nulls`:

```
let df_filtered = $df | polars drop-nulls

$df_filtered | polars collect
```

6.6.4.3 Sorting Data

To order a DataFrame by column values, use `polars sort-by`. For example, to sort records in descending order of `price`:

```
$df_filtered | polars sort-by price --reverse [true]
```

6.6.4.4 Grouping and Aggregation

Polars aggregates data across categories using `polars group-by` combined with `polars agg`. For example, to compute mean price and mean metascore grouped by game developer:

```
$df_filtered |

  polars group-by developer |

  polars agg (polars col game | polars count) |

  polars sort-by game --reverse [true] |

  polars collect
```

- For aggregated statistics grouped by `developer`:

```
$df_filtered |

  polars group-by developer |

  polars agg [
    (polars col price | polars mean | polars as "price_mean"),
    (polars col game  | polars count | polars as "n_games")
  ] |

  polars sort-by developer |

  polars collect
```

- For aggregated statistics grouped by `release_date` publication year:

```
#se mira la columna release_date
$df_filtered | polars select release_date | polars collect

#primero se extrae año, luego se agrupa
let data_by_year = $df_filtered |

  polars with-column [
    (polars col release_date
       | polars str-replace-all --pattern ".*,\\s*(\\d{4})$" --replace "$1" # extrae el año
       | polars str-replace-all --pattern "^\\s+|\\s+$" --replace ""        # trim
       | polars cast i64
       | polars as "year")
   ] |

  polars group-by year |

  polars agg [
    (polars col price | polars mean | polars as "price_mean"),
    (polars col game  | polars count | polars as "n_games")
```

```
  ] |

  polars sort-by year |

  polars collect

#se visualiza
$data_by_year

$data_by_year | describe

$data_by_year | polars into-nu #tabla nu

$data_by_year | polars into-nu | describe
```

6.6.5 File Formats and Persistence

To export a DataFrame to disk, use `polars save`. The plugin natively writes to Apache Parquet, CSV, and Arrow IPC binary formats:

```
$df_filtered | polars collect | polars save vgsales_filtered.parquet

$df_filtered | polars collect | polars save vgsales_filtered.csv

$data_by_year | polars save data_by_year.csv
```

When saving in certain pipeline environments, specify an absolute target path:

```
$df_filtered | polars collect | polars save ($env.PWD | path join vgsales_filtered.csv)
```

Or save directly to standard CSV format:

```
$df_filtered | polars collect | polars save vgsales_filtered.csv

$data_by_year | polars save data_by_year.csv
```

6.6.6 Column Pivoting

The `polars pivot` command reshapes data from long format to wide format by transposing unique categorical values into distinct columns while aggregating cell values.

To illustrate pivoting with the video game dataset, we first aggregate mean prices by `developer` and `year`:

```
let df_aggregated = $df_filtered |

```

```
  polars with-column [
    (polars col release_date
      | polars str-replace-all --pattern ".*,\\s*(\\d{4})$" --replace "$1"
      | polars str-replace-all --pattern "^\\s+|\\s+$" --replace ""
      | polars cast i64
      | polars as "year"
    )
  ] |

  polars select [
    (polars col developer)
    (polars col price)
    (polars col year)
  ] |

  polars group-by [developer year] |

  polars agg [(polars col price | polars mean | polars as price_mean)]

$df_aggregated | polars collect

$df_aggregated | polars sort-by year | polars collect

$df_aggregated | polars sort-by developer | polars collect
```

We then perform the relational pivot operation:

```
let df_pivoted = $df_aggregated
  | polars sort-by year
  | polars collect
  | polars pivot --index [developer] --on [year] --values [price_mean]
    --stable
  | polars collect
  | polars sort-by developer
  | polars collect

$df_pivoted | polars first 5 | polars into-nu

$df_pivoted | polars into-nu | save -f vgsales_pivot.csv
```

6.6.7 Best Practices and Recommendations

- **Name Columns Explicitly:** When applying aggregations or creating derived columns, assign unambiguous column names using descriptive aliases.
- **Leverage Lazy Evaluation:** Build execution plans with `polars into-lazy` before call-

ing `polars collect`, allowing the optimizer to prune unused columns and minimize RAM overhead.
- **Enforce Schemas During Ingestion:** Defining explicit schemas with `--schema` prevents misinferred column types and eliminates runtime type coercion issues.
- **Prefer Parquet for Storage:** For medium to large datasets, persist files in Apache Parquet rather than CSV to reduce storage footprint and achieve orders-of-magnitude faster ingestion.

6.6.8 Summary

The Nushell Polars plugin delivers enterprise-grade analytical capabilities for large-scale structured data, uniting the raw performance of Rust with Nushell's ergonomic pipeline syntax. By mastering data ingestion, vectorized filtering, multi-level aggregation, and pivoting in Polars, data engineers can run high-throughput workflows directly from the terminal with production-grade efficiency.

7 Exploratory Data Analysis and Visualization

This chapter focuses on the third phase of the OSEMN framework: **exploratory data analysis and visualization** (*Explore*). Before fitting predictive models or performing formal hypothesis testing, we must inspect the distribution of our variables, identify anomalous observations, compute descriptive statistics, and visualize multidimensional relationships. As John W. Tukey emphasized in his classic *Exploratory Data Analysis* (1977), exploratory data analysis (EDA) allows us to discover unexpected patterns, detect outliers, verify assumptions, and formulate informed hypotheses.

In Nushell, exploratory data analysis combines native functional pipelines (`describe`, `math`, `polars summary`) with polyglot scripts in R (`ggplot2`) and Python (`seaborn`, `matplotlib`). Graphics rendered by these scripts can be previewed directly inside the terminal using **`timg`** without leaving the shell.

We begin by navigating into the chapter directory `/data/ch07`:

```
cd /data/ch07
```

From this point forward, all commands in this chapter execute within `/data/ch07`. Let us inspect the available files:

```
ls
```

7.1 Inspecting Data and Its Properties

When approaching a new dataset, our initial questions are fundamental: Does it include a header row? How many observations and features are present? What data types are inferred for each column? Are there duplicate rows or invariant columns?

In this section, we examine `vehicles.csv` in `/data/ch07`:

```
open --raw venture.csv | lines | first 5
```

Opening the dataset with `open vehicles.csv` reveals its row count and schema inventory via `columns` and `enumerate`:

```
open venture.csv | length

open venture.csv | columns | enumerate | first 15

open venture.csv | columns | enumerate | wrap col_info | flatten | save -f venture_columns.csv
```

7.1.1 Inspecting Data Types and Schema Inference

Nushell automatically infers types upon ingestion (integers, floating-point numbers, booleans, datetimes, and strings):

```
open datatypes.csv

open datatypes.csv | save -f datatypes_copy.json
```

To inspect exact column types and nullability inferred by Polars, query `polars schema`:

```
open datatypes.csv | polars into-df | polars schema | transpose column dtype

open datatypes.csv | polars into-df | polars schema | transpose column dtype | save -f
↪  datatypes_schema.csv
```

7.1.2 Identifying Unique Values and Removing Invariant Columns

Columns containing a single constant value across all observations provide zero statistical variance for machine learning algorithms. We detect distinct values per column using `get` and `uniq`:

```
let df = open venture.csv

let non_const_cols = $df | columns | where {|c| ($df | get $c | uniq | length) > 1 }

$df | select ...$non_const_cols | first 5

$df | select ...$non_const_cols | save -f venture_clean.csv
```

Notice how cleanly Nushell pipelines compose: we isolate invariant columns with `where count == 1`, extract their identifiers, and discard them in a single functional pass via `reject`.

7.2 Computing Descriptive Statistics

Summary statistics provide quantitative characterizations of central tendency, spread, and skewness across numerical distributions. In this section, we explore `/data/ch07/tips.csv`:

```
open tips.csv | first 10

open tips.csv | first 10 | save -f tips_sample.csv
```

7.2.1 Summary Statistics with `math` and `polars summary` in Nushell

Nushell includes a native `math` command family (`math avg`, `math min`, `math max`, `math stddev`, `math median`) that operates directly on numeric columns:

```
open tips.csv | select bill tip size | math avg

open tips.csv | select bill tip size | math stddev

open tips.csv | select bill tip size | math median | save -f tips_median.json
```

To compute record counts, mean, standard deviation, quartiles, and range in a single vectorized pass, we invoke `polars summary`:

```
open tips.csv | polars into-df | polars summary | polars collect | polars into-nu

open tips.csv | polars into-df | polars summary | polars collect | polars into-nu | save -f
↪  tips_summary.csv
```

For categorical variables (such as `day` or `time`), pairing `uniq -c` provides immediate frequency distributions with in-terminal ASCII counts:

```
open tips.csv | histogram day

open tips.csv | histogram day | save -f tips_day_freq.csv
```

7.2.2 Transformations and Statistical Summaries via R and Python Scripts

When analytical tasks demand specialized statistical transformations, Nushell invokes external scripts written in R or Python, consuming their structured tabular output directly from standard output:

```
bat mutate_tips.R
```

Executing `Rscript mutate_tips.R` and piping its standard output into `from csv` allows sorting, projecting, and saving the transformed dataset entirely within Nushell:

```
Rscript mutate_tips.R | from csv | first 10

Rscript mutate_tips.R | from csv | save -f tips_with_percent.csv
```

Similarly, `summary_tips.R` groups observations by dining time and customer sex, computing average party sizes and gratuities:

```
bat summary_tips.R

Rscript summary_tips.R | from csv | sort-by mean_tip -r

Rscript summary_tips.R | from csv | sort-by mean_tip -r | save -f tips_grouped_summary.csv
```

With Python (`corr_tips.py`), we compute a Pearson correlation matrix and stream it directly back to Nushell:

```
bat corr_tips.py

python3 corr_tips.py | from csv

python3 corr_tips.py | from csv | save -f tips_correlations.csv
```

7.3 Creating Statistical Visualizations from the Command Line

Visualizations allow analysts to detect nonlinear relationships, multimodality, and outliers that summary metrics frequently conceal. In `/data/ch07`, we generate faceted plots of gratuities across meal times and days (`tips_faceted.png`):

```
bat plot_faceted.R

Rscript plot_faceted.R

ls tips_faceted.png
```

7.3.1 Viewing Images Directly in the Terminal with `timg`

To inspect graphical artifacts without leaving the terminal session or launching a graphical desktop, the `dsin` container includes **`timg`** (*Terminal Image Viewer*).

> [!NOTE] **Graphical fidelity and native resolution in the terminal:** When using a modern terminal emulator supporting bitmap graphics protocols—such as the Kitty Graphics Protocol or Sixel, built natively into emulators like **Ghostty**, Kitty, or WezTerm—`timg` automatically detects terminal capabilities and displays images in their **original pixel-by-pixel resolution**, without degrading them to text character blocks. On standard terminals or SSH connections lacking this support, `timg` falls

back automatically to high-density half-block Unicode characters (`-g 80x25`), ensuring immediate visual inspection in any terminal environment.

To render an image in `/data/ch07`, invoke:

```
timg -g 80x25 tips_faceted.png
```

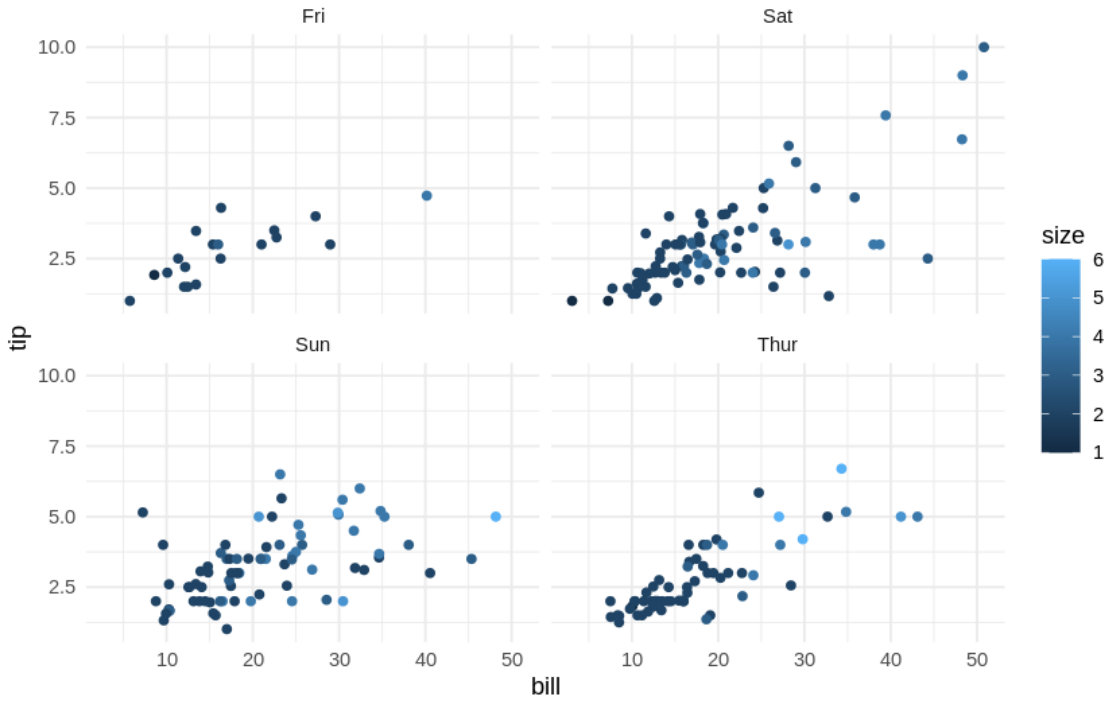


Figure 7: Faceted scatter plot of total bills and tips across days of the week

Multiple figures can also be arranged and displayed as a composite terminal gallery using `--grid`:

```
Rscript plot_histogram.R

Rscript plot_density.R

timg --grid=2x1 -g 80x25 tips_histogram.png tips_density.png
```

7.3.2 Remote Alternatives: Copying Graphics via `scp`

When working over SSH connections where high-resolution terminal graphics are unsupported, image files can be transferred to a local workstation using standard `scp` (*secure copy*):

```
scp usuario@servidor:/data/ch07/tips_faceted.png .
```

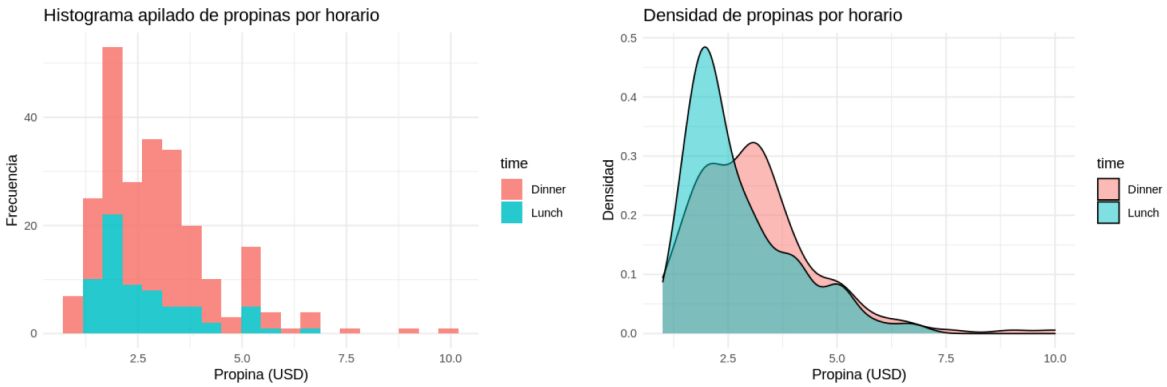


Figure 8: Composite terminal gallery of histogram and kernel density estimation

7.4 Catalog of Statistical Graphics via R and Python Scripts

This section provides a reference gallery of standard statistical visualizations authored in R (`ggplot2`) and Python (`seaborn`), ready for CLI execution.

7.4.1 Bar Charts

To visualize discrete distributions across categorical groups, execute `plot_bar.R`:

```
bat plot_bar.R

Rscript plot_bar.R

timg -g 80x25 tips_bar.png
```

7.4.2 Histograms

To examine the continuous empirical distribution of bill totals, execute `plot_histogram.R`:

```
bat plot_histogram.R

Rscript plot_histogram.R

timg -g 80x25 tips_histogram.png
```

7.4.3 Density Plots

Kernel density estimation provides a smooth approximation of probability density functions, demonstrated in `plot_density.R`:

```
bat plot_density.R
```

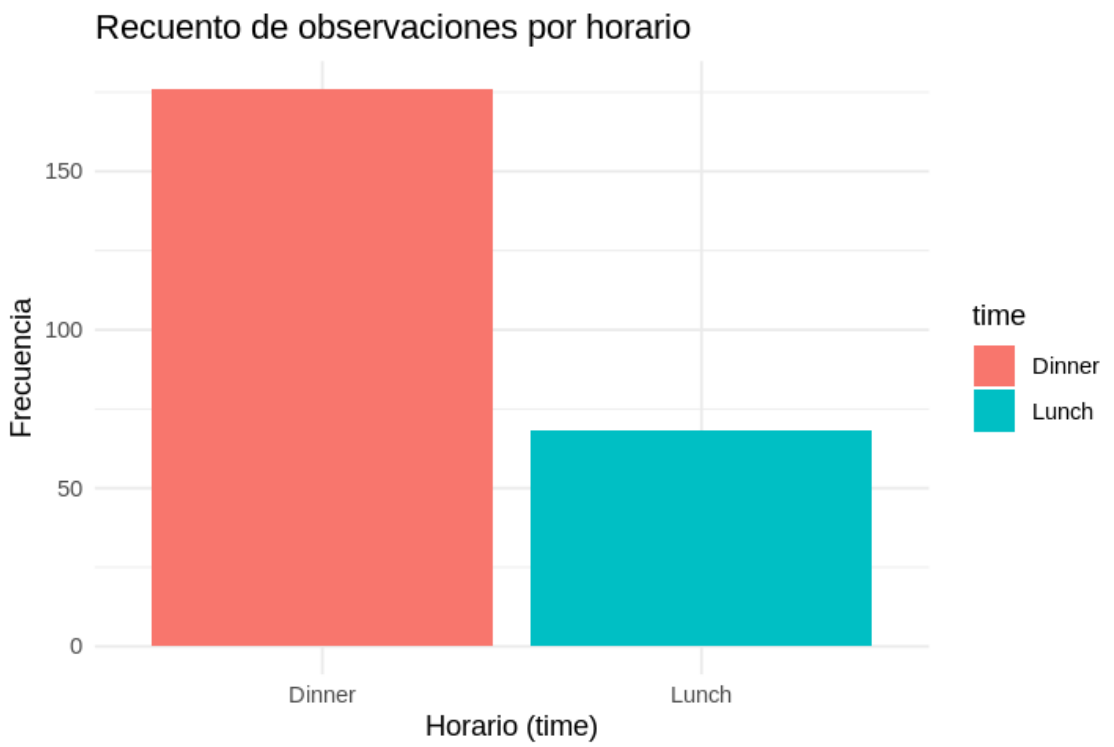


Figure 9: Bar chart generated with ggplot2

```
Rscript plot_density.R

timg -g 80x25 tips_density.png
```

7.4.4 Line Charts for Time Series

To examine longitudinal patterns and chronological trends, execute `plot_accident.R`:

```
open Fatalities.csv | first 5

open Fatalities.csv | first 5 | save -f fatalities_head.csv

bat plot_accident.R

Rscript plot_accident.R

timg -g 80x25 fatalities_line.png
```

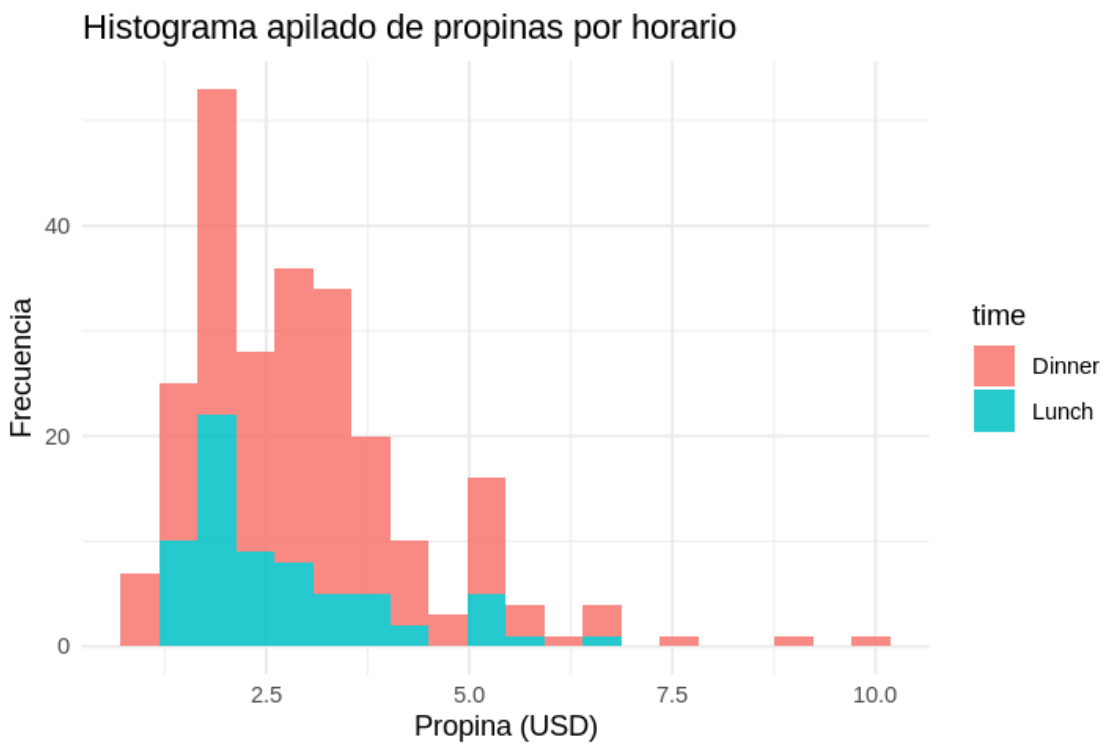


Figure 10: Histogram of total bill values

7.4.5 Scatter Plots and Trend Curves

To assess bivariate associations between bill size and tip amount, execute `plot_scatter.R` and `plot_trend.R`:

```
bat plot_scatter.R

Rscript plot_scatter.R

bat plot_trend.R

Rscript plot_trend.R

timg --grid=2x1 -g 80x25 tips_scatter.png tips_trend.png
```

7.4.6 Box Plots and Violin Plots

Box plots highlight quartiles and potential outliers across groups, while violin plots overlay kernel densities:

```
bat plot_boxplot.R
```

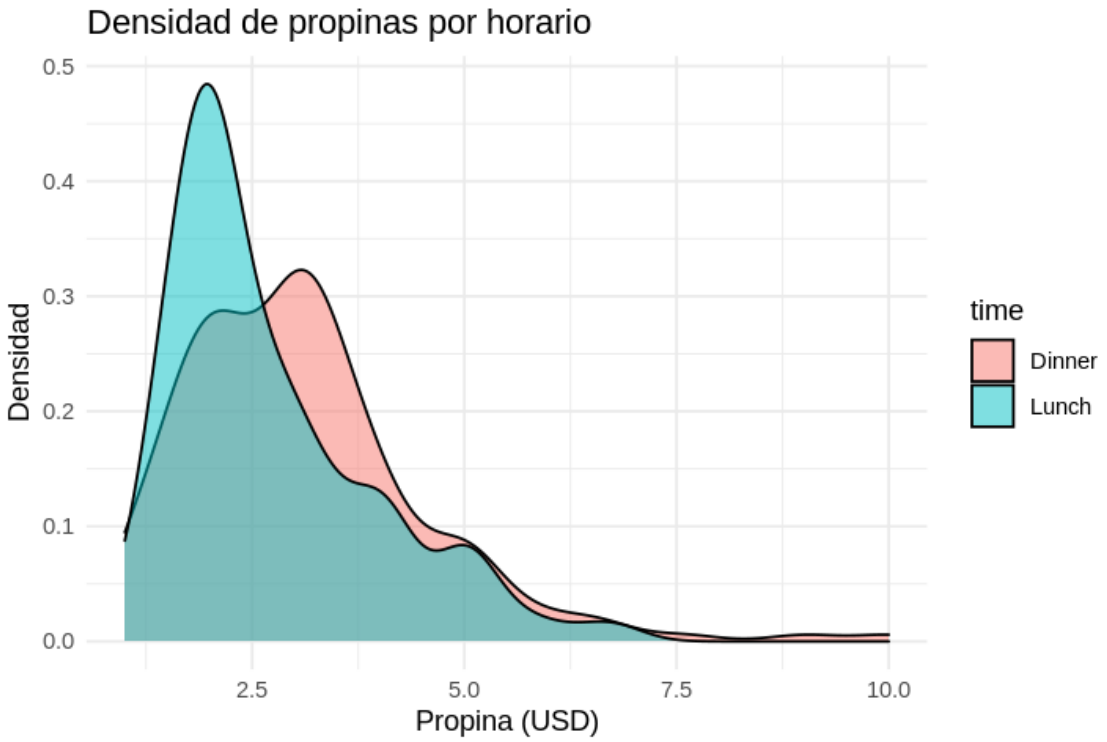


Figure 11: Kernel density estimation

```
Rscript plot_boxplot.R

bat plot_violin.R

Rscript plot_violin.R

timg --grid=2x1 -g 80x25 tips_boxplot.png tips_violin.png
```

7.4.7 Equivalent Statistical Visualization in Python

Equivalent visualization pipelines can be constructed in Python. The script `plot_iris_py.py` computes pair-wise feature relationships across Iris measurements, outputting summary tables to Nushell:

```
bat plot_tips_py.py

python3 plot_tips_py.py | from json

python3 plot_tips_py.py | from json | save -f tips_py_summary.json

timg -g 80x25 tips_python.png
```

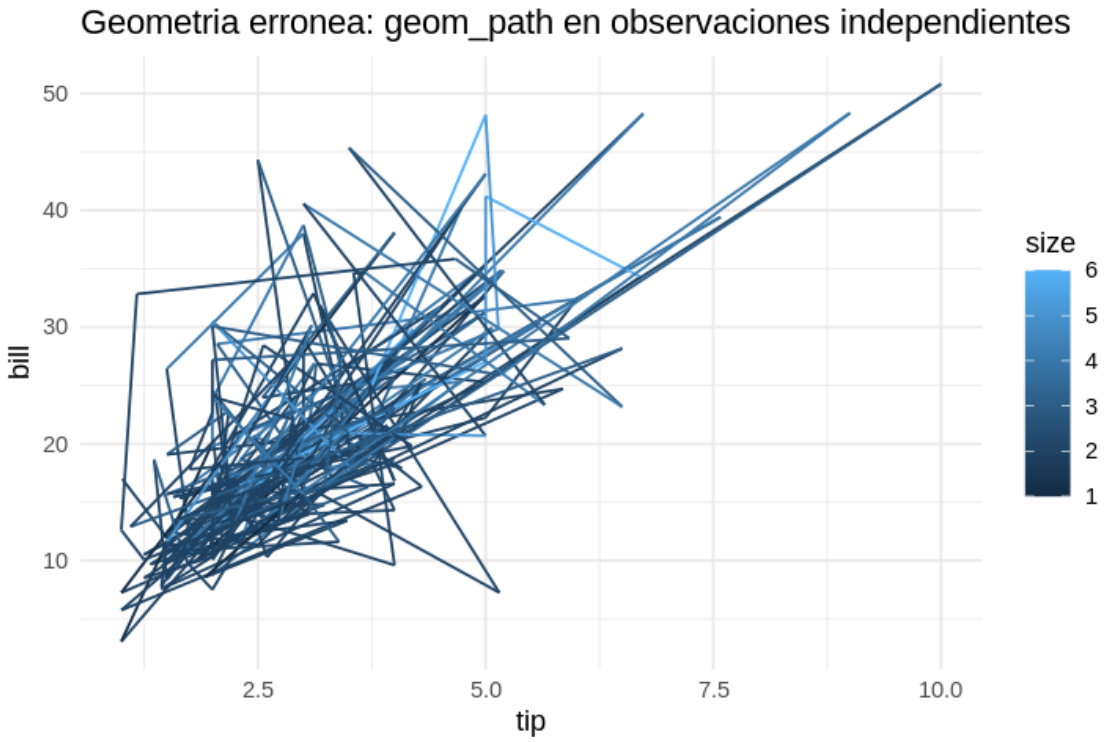


Figure 12: Time series of traffic accidents

7.5 Summary

Exploratory data analysis in Nushell establishes a unified bridge between command-line speed and rigorous statistical graphics. Native commands like `math`, `describe`, and `polars summary` provide immediate quantitative profiling of distributions. When advanced graphics are required, polyglot scripts in R and Python render high-resolution figures that can be previewed directly in the terminal via `timg`.

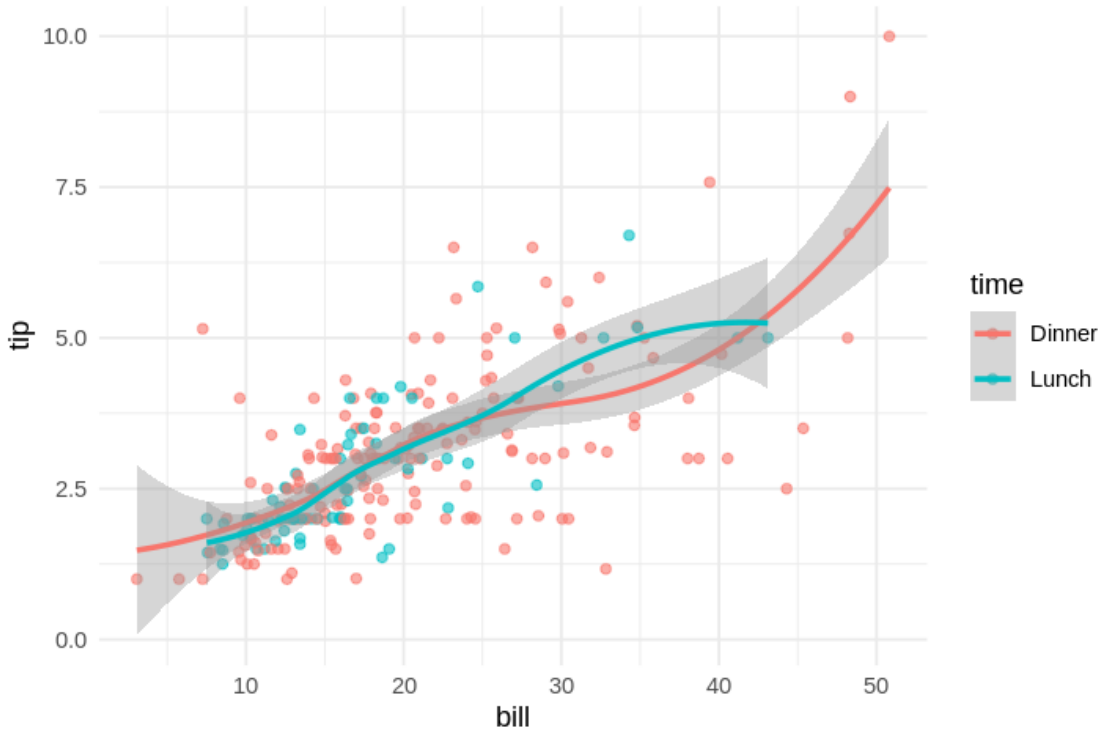


Figure 13: Scatter plot with LOESS trend curve

8 Serial and Parallel Data Pipelines

Computational data science pipelines routinely handle computationally intensive tasks: iterating over thousands of URLs, parsing gigabytes of server logs, computing cross-validated machine learning benchmarks, or applying image processing algorithms across thousands of images.

When tasks are executed sequentially (in series), throughput is bound by a single CPU core, leaving the remaining cores of modern multicore processors idle. In this chapter, we explore how to parallelize pipelines using Nushell native multi-threaded iterator **`par-each`** as well as the external process parallelizer **GNU `parallel`**.

We begin by navigating into `/data/ch08`:

```
cd /data/ch08
```

From this point forward, commands in this chapter execute within `/data/ch08`. Let us inspect the available files:

```
ls
```

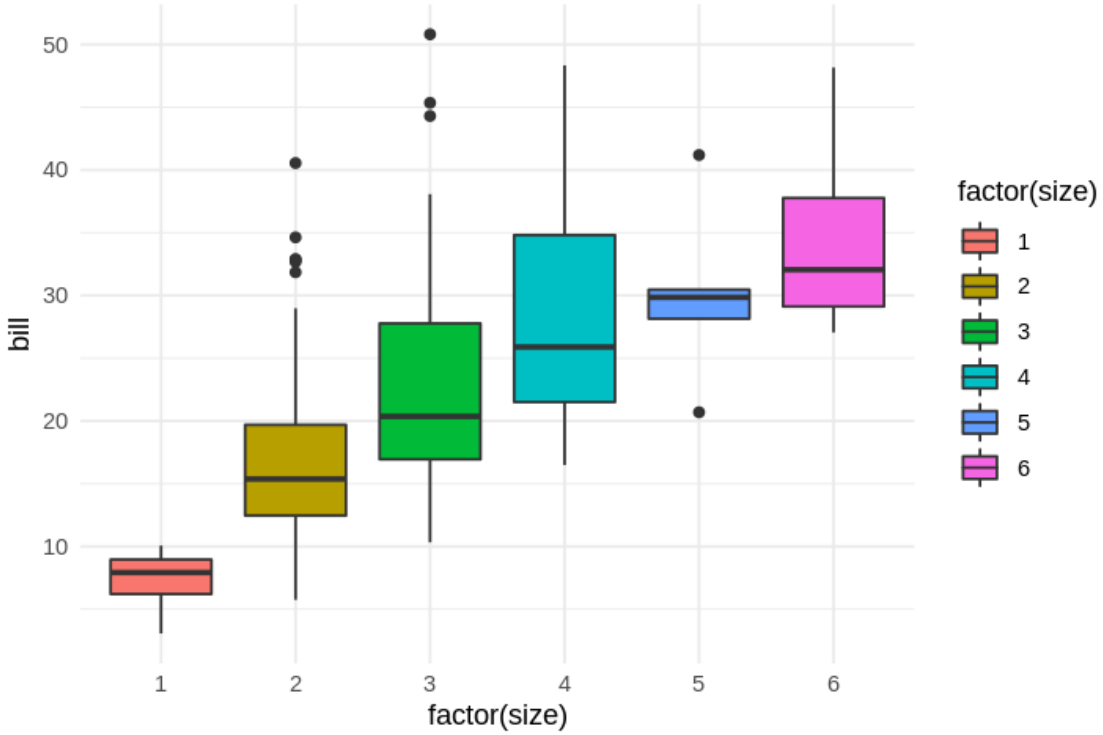


Figure 14: Box plot of bills by day

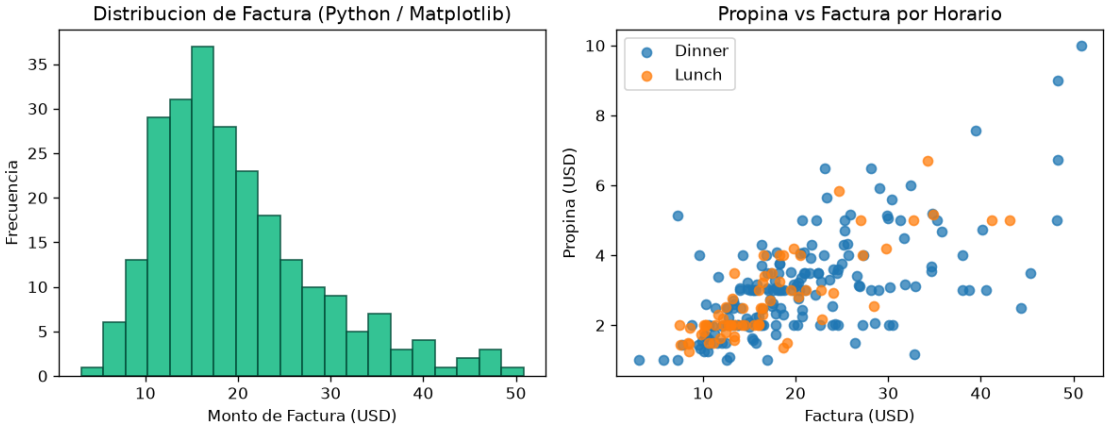


Figure 15: Dual-panel empirical distribution and scatter plot generated via Python matplotlib

8.1 Serial Processing: Iterating Over Data with `each`

The `each` command iterates sequentially over elements in a collection, list, or table, evaluating a closure for each element and emitting results in original input order.

Consider evaluating an integer sequence from 1 to 10 to identify multiples of 3:

```
1..10 | each {|x| {n: $x, cuadrado: ($x * $x), es_mult_3: (($x mod 3) == 0)} }

1..10 | each {|x| {n: $x, cuadrado: ($x * $x), es_mult_3: (($x mod 3) == 0)} } | save -f
 ↪  squares_table.csv
```

For high-precision mathematical operations, external utilities like **`bc -l`** can be invoked from within closures:

```
1..5 | to text | pbc "{1}^2" | lines | each {|v| $v | into int }
```

8.1.1 Iterating Over Lines and Rows

When iterating over structured datasets, `each` receives entire records, granting access to individual fields via record dot notation. Here, we iterate over `input.csv`:

```
open --raw movies.txt | lines | enumerate | each {|row| {idx: $row.index, movie: ($row.item | str
 ↪  trim), chars: ($row.item | str length)} }

open --raw movies.txt | lines | enumerate | each {|row| {idx: $row.index, movie: ($row.item | str
 ↪  trim), chars: ($row.item | str length)} } | save -f movies_summary.csv
```

8.1.2 Iterating Over Files

To process collections of files sequentially, we pipeline directory listings from `ls` into `each`, transforming and persisting each artifact:

```
ls *.csv | each {|f| {archivo: $f.name, bytes: $f.size, filas: (open $f.name | length)} }

ls *.csv | each {|f| {archivo: $f.name, bytes: $f.size, filas: (open $f.name | length)} } | save
 ↪  -f csv_inventory.csv
```

8.2 Parallel Processing in Nushell with `par-each`

While `each` processes items sequentially, **`par-each`** distributes closures across an asynchronous thread pool, processing multiple items concurrently across available CPU cores.

To quantify performance differences between serial and parallel execution, we use `timeit`:

```
1..4 | par-each --threads 4 {|i| {job: $i, msg: $"Tarea ($i) completada"} }
```

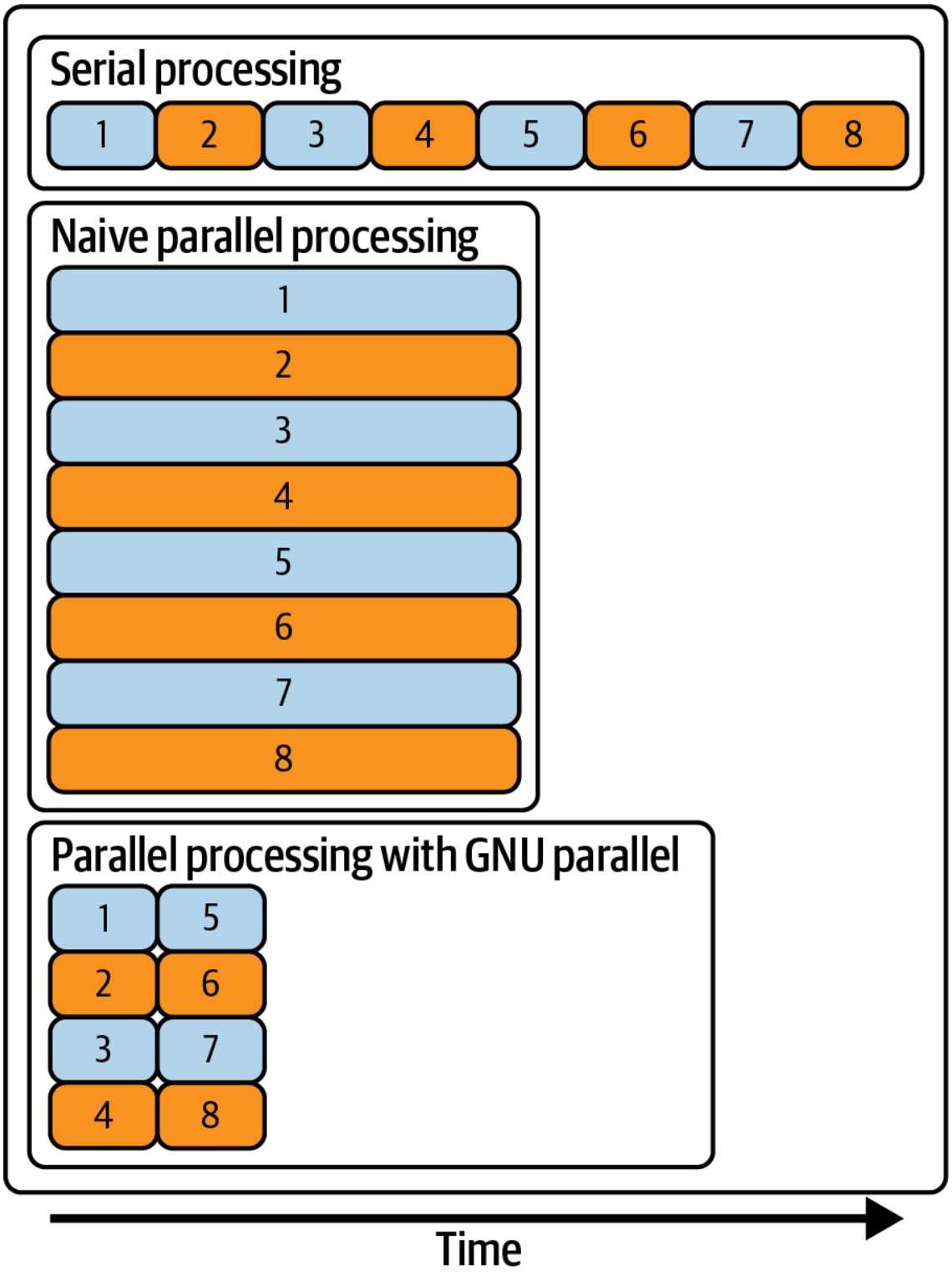


Figure 16: Conceptual comparison between sequential and parallel execution

```
1..4 | par-each --threads 4 --keep-order {|i| {job: $i, cuadrado: ($i * $i)} } | save -f
↪  parallel_jobs.csv
```

8.2.1 Concurrent Execution and Thread Control

Two essential flags govern `par-each` behavior:

- **`--threads` (`-t`):** Restricts the maximum number of concurrent threads allocated to the execution pool.
- **`--keep-order` (`-k`):** Guarantees that outputs appear in the exact order of the input collection, even if worker threads complete out of sequence.

8.3 Process Parallelization with GNU `parallel` from Nushell

When parallelization involves coordinating external CLI binaries, legacy shell scripts, or distributing work across remote networked clusters via SSH, **GNU `parallel`** provides an industrial-grade workload manager.

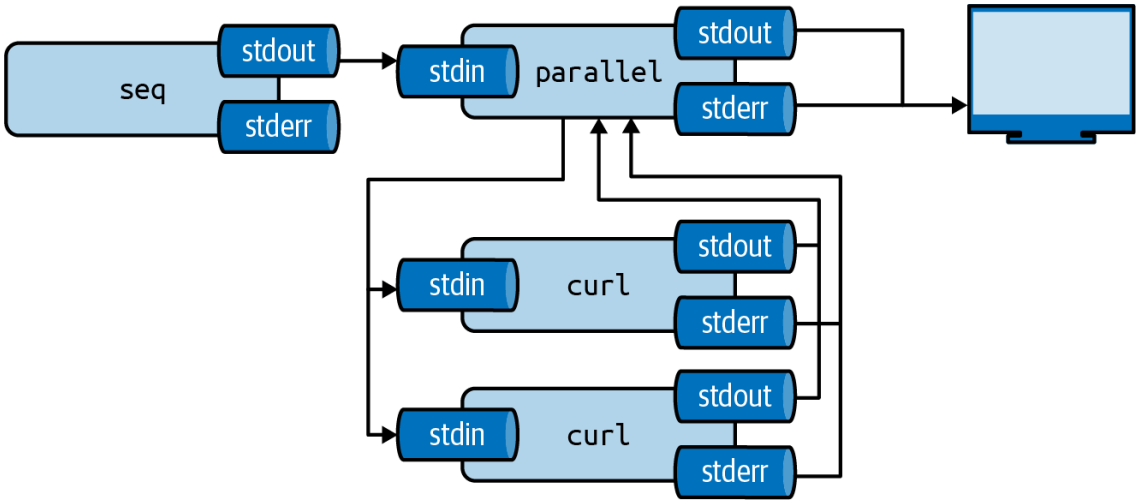


Figure 17: Execution flow of GNU parallel distributing arguments across multiple cores

8.3.1 Specifying Arguments and Replacement Tokens

GNU `parallel` utilizes positional token placeholders to inject arguments into target commands:

- **`{}`:** Complete input argument string.
- **`{/}`:** Filename basename with directory components stripped.
- **`{.}`:** Filename stem with directory intact but final extension stripped.
- **`{/.}`:** Basename with both parent directory and extension stripped.
- **`{#}`:** Incremental job sequence number.

We can invoke `parallel` directly from Nushell, piping its output into a structured table:

```
open input.csv

open --raw input.csv | parallel --colsep ',' --header : "echo {name}:{quantity}" | lines | where
 ↪  ($it | str contains ":") | split column ":" nombre cantidad | collect | save -f
 ↪  input_parallel.csv

open input_parallel.csv
```

8.3.2 Controlling Concurrent Jobs

By default, GNU `parallel` spawns one worker job per CPU core. The `--jobs` (`-j`) flag adjusts concurrency explicitly (e.g., `-j 4`, `-j 200%`, or `-j +2`):

```
1..4 | to text | parallel -j 4 "sh slow.sh {}" | lines
```

8.3.3 Output Logging and Tagging

When multiple background processes write to standard output concurrently, their emissions can become interleaved.

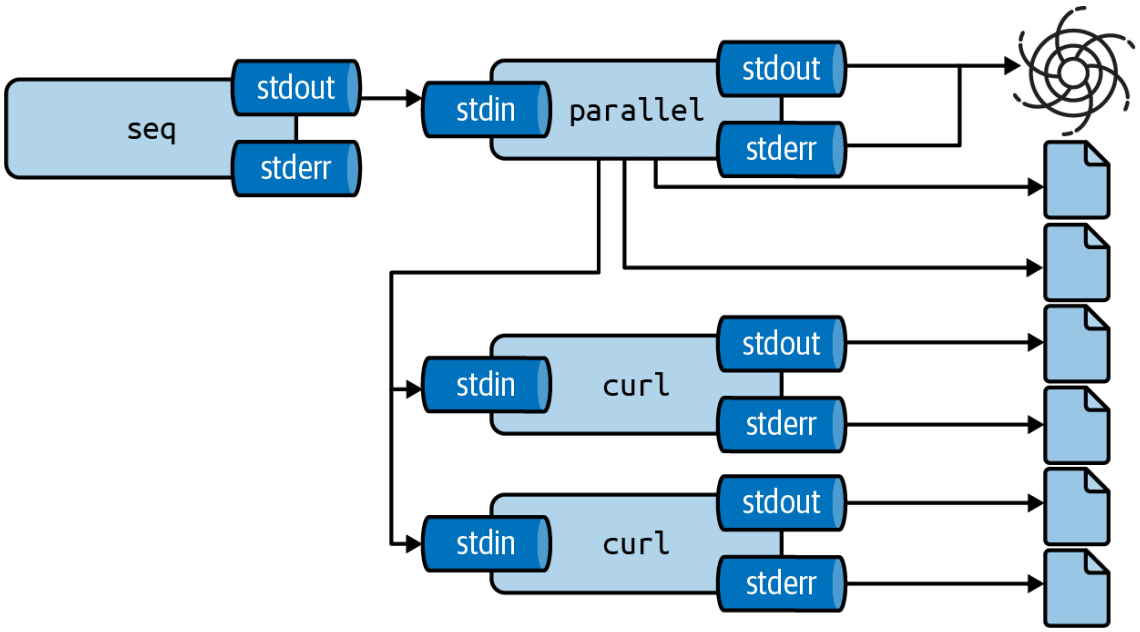


Figure 18: Interleaved outputs when multiple processes write concurrently

GNU `parallel` prevents interleaved corruption by capturing process buffers and emitting complete outputs per job. Useful control flags include:

- **`--keep-order` (`-k`):** Emits output in input order.

- **`--tag`:** Prefixes each output line with its input argument.
- **`--results <dir>`:** Directs stdout, stderr, and exit codes into a structured directory hierarchy.

We run `parallel` with `--results salida_parallel` and parse the generated records directly in Nushell:

```
1..5 | to text | parallel -k --tag "echo $(( {} * 10 ))" | from tsv --noheaders | rename arg
 ↪  resultado

1..5 | to text | parallel -k --tag "echo $(( {} * 10 ))" | from tsv --noheaders | rename arg
 ↪  resultado | save -f tagged_output.csv
```

We can likewise combine `parallel` with utilities like **`seq`**, **`bc`**, or **`jq`**:

```
["Nushell", "Polars"] | str join "\n" | parallel -k "cowsay {}"
```

8.3.4 Parallel Tools and Block Processing

When processing massive single files (such as a multi-gigabyte CSV or JSON stream), parallelizing line-by-line is inefficient. The **`--pipe`** flag partitions input streams into multi-megabyte blocks, feeding each block to a separate worker process:

```
open --raw movies.txt | parallel --pipe -N 3 "wc -l" | lines | each {|x| $x | str trim | into int
 ↪  } | math sum
```

8.4 Distributed Computing Across Multiple Nodes

Beyond single-machine multicore execution, GNU `parallel` can distribute computations across clusters of remote servers using SSH.

8.4.1 Discovering Active Cluster Nodes

Cluster node hostnames or IP addresses can be enumerated in a configuration file (`~/.parallel/cluster_hosts` or `/data/ch08/servers.txt`).

8.4.2 Running Commands Across All Cluster Nodes

To audit operating system metrics or check software versions across all cluster nodes:

```
parallel --nonall --slf hosts.txt "nu --version"
```

8.4.3 Distributing Local Data Streams Across Cluster Nodes

To stream local data chunks across cluster nodes for distributed processing:

```
cat gran_dataset.csv | parallel --pipe --header : --block 50M --slf hosts.txt "nu -c \"from csv |
↪  where amount > 100 | to csv\"" > filtrado_distribuido.csv
```

8.4.4 Processing Files on Remote Nodes

GNU `parallel` can automatically upload input files to remote nodes via SCP/rsync, execute analytical workloads remotely, and transfer result files back to the local coordinator:

```
ls *.csv | parallel --slf hosts.txt --transferfile {} --return {.}_clean.csv "nu limpiar_lote.nu
↪  {} {.}_clean.csv"
```

8.5 Summary

Parallel processing accelerates computationally intensive data science pipelines by fully utilizing modern multicore and networked hardware. Nushell native **`par-each`** provides zero-overhead in-memory concurrency for internal table and collection transformations. For orchestrating external binaries, batch file transformations, or distributed cluster computing across SSH nodes, **GNU `parallel`** pairs seamlessly with Nushell pipelines.

9 Polyglot Data Modeling

The fourth phase of the OSEMN framework is **data modeling** (*Model*), paired directly with empirical evaluation (*iNterpret*). At this stage, our data has been acquired (Chapter 5), cleaned and transformed (Chapter 6), and thoroughly explored (Chapter 7). We now construct predictive, descriptive, and inferential models across both unsupervised and supervised machine learning domains.

Rather than forcing complex mathematical modeling into custom shell scripts, modern command-line data science adopts a **polyglot architecture**:

1. **Standalone CLI modeling tools:** High-performance, compiled command-line utilities engineered specifically for rapid iteration (**Tapkee** for dimensionality reduction, **Vowpal Wabbit** for online out-of-core learning, and **SKLL** for declarative model benchmarking).
2. **Polyglot scripting with R and Python:** Direct invocation of R scripts (`Rscript`) for classical statistical inference and Python scripts (`python3`) for algorithmic machine learning via scikit-learn.
3. **Nushell as universal orchestrator:** Nushell handles end-to-end data pipelines, formats input feature streams, launches modeling processes concurrently, parses tabular outputs, and computes comparative performance benchmarks.

We begin by entering the chapter working directory `/data/ch09`:

```
cd /data/ch09
```

From this point forward, all code examples and modeling pipelines execute within `/data/ch09`.

9.1 Data Preparation and Exploration of the Wine Dataset

First, check the dataset files available inside `/data/ch09`:

```
ls
```

The files `wine-red.csv` (1,599 red wines) and `wine-white.csv` (4,898 white wines) document 11 physicochemical properties (acidity, residual sugar, chlorides, sulfur dioxide, pH, sulphates, alcohol) and subjective sensory quality ratings (0 to 10) from the classic vinho verde dataset (Cortez et al., 2009).

We concatenate both datasets into a single unified table (`wine-both.csv`), creating a categorical `type` feature and saving to disk:

```
let red = open wine-red.csv | insert type "red"

let white = open wine-white.csv | insert type "white"

let both = $red | append $white

```

```
$both | first 5

$both | save -f wine-both-generated.csv
```

Next, we check the class balance (`red` versus `white`) with `histogram`:

```
open wine-both.csv | histogram type

open wine-both.csv
| histogram type
| save -f wine_type_distribution.csv
```

Before fitting any statistical model, we plot the empirical distributions of sensory quality and alcohol concentration across wine varieties using `plot_wine_eda.R`. Display its source code in the terminal:

```
bat plot_wine_eda.R
```

```
#!/usr/bin/env Rscript
suppressPackageStartupMessages(library(ggplot2))
df <- read.csv("wine.csv")

p1 <- ggplot(df, aes(x = quality, fill = type)) +
  geom_density(adjust = 3, alpha = 0.5) +
  labs(
    title = "Quality Distribution by Wine Type",
    x = "Sensory Quality", y = "Density"
  ) +
  theme_minimal()
ggsave(
  "wine-quality.png",
  plot = p1, width = 6, height = 4, dpi = 120
)
ggsave(
  "wine_eda.png",
  plot = p1, width = 6, height = 4, dpi = 120
)

p2 <- ggplot(df, aes(x = alcohol, y = quality, color = type)) +
  geom_smooth(method = "loess", formula = y ~ x) +
  labs(
    title = "Alcohol vs. Quality by Wine Type",
    x = "Alcohol (%)", y = "Sensory Quality"
  ) +
  theme_minimal()
ggsave(
```

```
  "wine-alcohol-vs-quality.png",
  plot = p2, width = 6, height = 4, dpi = 120
)
```

Three building blocks structure `plot_wine_eda.R`:

1. **Silent package loading and ingestion:** `suppressPackageStartupMessages` keeps standard output free of startup banners before `read.csv("wine.csv")` loads all 6,497 observations.
2. **Smoothed class-conditional density (`p1`):** `geom_density(adjust = 3, alpha = 0.5)` widens the Gaussian kernel bandwidth to smooth discrete integer quality scores across `red` and `white` wines.
3. **Non-parametric trend estimation (`p2`):** `geom_smooth(method = "loess", formula = y ~ x)` fits local polynomial regressions of sensory quality on alcohol percentage for each wine variety, exporting crisp raster figures via `ggsave`.

We execute the script and render `wine_eda.png` directly inside the terminal with `timg`:

```
Rscript plot_wine_eda.R

timg -g 80x25 wine_eda.png
```



Figure 19: Class-conditional density of sensory quality across red and white wines

9.2 Dimensionality Reduction (PCA and t-SNE)

High-dimensional feature tables often suffer from strong collinearity and geometric sparsity. Dimensionality reduction maps p correlated predictors into a low-dimensional subspace (typically two or three coordinates) that retains either global variance or local neighborhood structure:

- **Principal Component Analysis (PCA):** Orthogonal linear projection aligned with directions of maximum sample variance.
- **t-Distributed Stochastic Neighbor Embedding (t-SNE):** Non-linear manifold embedding that converts pairwise Euclidean distances into conditional probabilities to preserve local neighborhoods.

9.2.1 Dimensionality Reduction with CLI Tools

The `dsin` container includes **Tapkee** (Lisitsyn et al., 2013), a compiled C++ template library for fast spectral and manifold dimensionality reduction. Like most C++ numerical binaries, `tapkee` knows nothing about Nushell schemas, column names, or string labels. It expects a headerless file of floating-point numbers separated by a declared delimiter (`--delimiter ","`) and writes unlabeled coordinate rows to an output file.

Running `tapkee` directly therefore requires an explicit three-stage pipeline in Nushell: preprocessing the input matrix, invoking the native binary, and post-processing the raw coordinate stream. First, we slice 200 observations from `wine-both.csv`, drop `type` and `quality` so only the 11 continuous physicochemical measurements remain, serialize a headerless CSV matrix, and call `tapkee`:

```
let sample_w = open wine-both.csv | first 200

$sample_w
  | reject type quality
  | to csv --noheaders
  | save -f wine_matrix.csv

let tk_cmd = [
  "-i" "wine_matrix.csv" "-o" "wine_pca_raw.csv"
  "-m" "pca" "-td" "2" "--delimiter" ","
]

tapkee ...$tk_cmd

open --raw wine_pca_raw.csv | lines | first 5
```

Inspecting the raw output shows plain comma-separated coordinate pairs (`-1.67715,-16.4492`) with neither headers nor row identifiers. Because `tapkee` preserves input row order, we parse the headerless file back into Nushell, assign descriptive column names (`dim_1`, `dim_2`), round each coordinate to four decimal places, and merge the projection horizontally onto `$sample_w`:

```
let coords = open --raw wine_pca_raw.csv
  | from csv --noheaders
  | rename dim_1 dim_2
  | update dim_1 { into float | math round -p 4 }
  | update dim_2 { into float | math round -p 4 }

let direct_pca = $sample_w
  | merge $coords
  | select type quality dim_1 dim_2

$direct_pca | first 5

rm -f wine_matrix.csv wine_pca_raw.csv
```

Repeating that manual sequence (feature isolation, temporary CSV creation, external binary execution, float parsing, horizontal merge, and file cleanup) every time we explore a projection quickly clutters interactive sessions. In `dsin_tools.nu`, the custom command **`tapkee-reduce`** encapsulates those exact operations behind a typed `table -> table` signature. To inspect its implementation directly in the terminal, we execute `view-code tapkee-reduce`:

```
view-code tapkee-reduce
```

```
# Run Tapkee dimensionality reduction on a numeric table.
export def tapkee-reduce [
  --method(-m): string = "pca"
  --target-dim(-d): int = 2
  --label-col(-l): string
  --cols(-c): list<string>
]: table -> table {
  let tbl = $in
  let norm_m = if $method == "tsne" { "t-sne" } else { $method }
  let has_cols = ($cols != null) and (($cols | length) > 0)
  let has_lbl = if ($label_col != null) {
    $label_col in ($tbl | columns)
  } else { false }
  let num_tbl = if $has_cols {
    $tbl | select ...$cols
  } else if $has_lbl {
    $tbl | reject $label_col
  } else {
    let non_num = ["type"] | where { |c| $c in ($tbl | columns) }
    if ($non_num | is-not-empty) {
      $tbl | reject ...$non_num
    } else {
      $tbl
    }
```

```
  }

  let tmp_in = mktemp --suffix ".csv"
  let tmp_out = mktemp --suffix ".tsv"
  $num_tbl | to csv --noheaders | save -f $tmp_in

  let tk_args = [
    "-i" $tmp_in "-o" $tmp_out
    "-m" $norm_m "-td" ($target_dim | into string)
    "--delimiter" ","
  ]
  tapkee ...$tk_args
  let raw_coords = open --raw $tmp_out
    | lines
    | where ($it | str trim | is-not-empty)
  rm -f $tmp_in $tmp_out

  let coords_tbl = $raw_coords | each { |line|
    let parts = $line
      | split row ","
      | each { |v| $v | str trim | into float | math round -p 4 }
    mut rec = {}
    for idx in 0..<($parts | length) {
      let val = $parts | get $idx
      let k1 = $"dim($idx + 1)"
      let k2 = $"dim_($idx + 1)"
      $rec = $rec | insert $k1 $val | insert $k2 $val
    }
    $rec
  }

  $tbl | merge $coords_tbl
}
```

Four design choices turn the raw C++ binary into a composable shell operator:

1. **Flexible schema filtering:** Callers can pass an explicit feature list (`--cols`), exclude a target label (`--label-col`), or rely on automatic removal of categorical `type` labels while retaining every original column in `$tbl` for the final merge.
2. **Alias normalization:** Translates `--method tsne` into Tapkee's canonical `"t-sne"` identifier so interactive typos do not abort pipelines.
3. **Ephemeral file hygiene:** Allocates scratch files via `mktemp` and deletes them immediately after reading `$tmp_out`, preventing leftover `.csv` artifacts in `/data/ch09`.
4. **General d-dimensional reconstruction:** Dynamically builds `dim_1`, `dim_2`, …, `dim_d` records for arbitrary `--target-dim` values before executing `$tbl | merge $coords_tbl`.

We now repeat the exact same 200-row PCA projection using `tapkee-reduce`, obtaining identical coordinates in a single pipeline:

```
let features = [
  "fixed_acidity" "volatile_acidity" "citric_acid"
  "residual_sugar" "chlorides" "free_sulfur_dioxide"
  "total_sulfur_dioxide" "density" "ph" "sulphates" "alcohol"
]

let pca_res = open wine-both.csv
  | first 200
  | tapkee-reduce --method pca --cols $features
  | select type quality dim_1 dim_2

$pca_res | first 5

$pca_res | save -f wine_pca_tapkee.csv
```

Switching `--method pca` to `--method t-sne` computes a two-dimensional non-linear manifold embedding through the exact same interface:

```
let tsne_res = open wine-both.csv
  | first 150
  | tapkee-reduce --method t-sne --cols $features
  | select type quality dim_1 dim_2

$tsne_res | first 5

$tsne_res | save -f wine_tsne_tapkee.csv
```

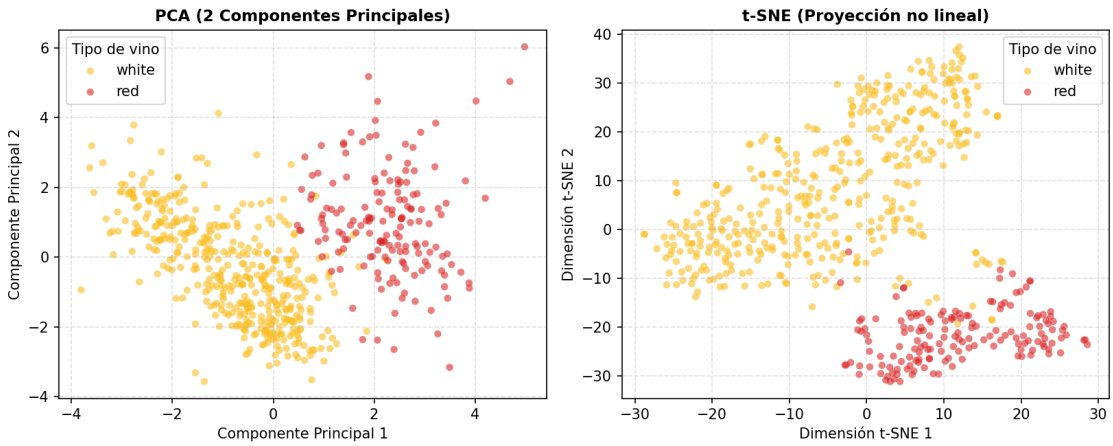


Figure 20: Bidimensional projections obtained via dimensionality reduction

9.2.2 Dimensionality Reduction with `Rscript`

While `tapkee` excels at rapid coordinate projection, statistical diagnostics often require inspecting eigenvalues and cumulative explained variance across components. The script `pca_wine.R` performs standardized singular value decomposition in R (`prcomp`) and streams variance summaries directly to standard output:

```
bat pca_wine.R
```

```
#!/usr/bin/env Rscript
df <- read.csv("wine.csv", stringsAsFactors = FALSE)
num_cols <- setdiff(names(df), c("type", "quality"))
pca <- prcomp(df[, num_cols], center = TRUE, scale. = TRUE)
var_exp <- (pca$sdev^2) / sum(pca$sdev^2) * 100

# Save loadings summary to stdout as CSV
out <- data.frame(
  component = paste0("PC", 1:5),
  std_dev = round(pca$sdev[1:5], 4),
  pct_variance = round(var_exp[1:5], 2),
  cum_variance = round(cumsum(var_exp)[1:5], 2)
)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)

# Also save 2D coordinates to wine_pca_r.csv
coords <- data.frame(
  pc1 = round(pca$x[, 1], 4),
  pc2 = round(pca$x[, 2], 4),
  type = df$type,
  quality = df$quality
)
write.csv(coords, "wine_pca_r.csv", row.names = FALSE, quote = FALSE)
```

Notice the three statistical and I/O stages in `pca_wine.R`:

1. **Feature isolation and z-score standardization:** `setdiff` removes `type` and `quality`, while `prcomp(..., center = TRUE, scale. = TRUE)` standardizes all 11 predictors to unit variance so variables with large magnitudes (such as `total_sulfur_dioxide`) do not dominate `PC1`.
2. **Spectral variance decomposition:** Squaring singular values (`pca$sdev^2`) yields component eigenvalues, from which marginal (`var_exp`) and cumulative (`cumsum(var_exp)`) variance percentages are derived.
3. **Dual tabular output:** `write.csv(out, stdout(), ...)` emits a clean 5-row variance summary to standard output for Nushell's `from csv` parser, while writing full 2D scores (`pca$x[, 1:2]`) to `wine_pca_r.csv`.

Executing `Rscript pca_wine.R | from csv` converts R's text stream into a typed Nushell table:

```
let pca_r = Rscript pca_wine.R | from csv

$pca_r

$pca_r | save -f wine_pca_summary_r.csv
```

The first four principal components account for 73.37% of total physicochemical variance across the 11 predictors (`PC1` alone captures 27.54%, driven by the contrast between sulfur dioxide levels in white wines and volatile acidity in red wines).

9.2.3 Dimensionality Reduction with Python

The companion script `pca_wine.py` reproduces the same orthogonal decomposition using `scikit-learn`:

```
bat pca_wine.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("wine.csv")
feat_cols = [c for c in df.columns if c not in ("type", "quality")]
X = StandardScaler().fit_transform(df[feat_cols])

pca = PCA(n_components=5, random_state=42)
coords = pca.fit_transform(X)

proj = pd.DataFrame({
    "pc1": coords[:, 0].round(4),
    "pc2": coords[:, 1].round(4),
    "type": df["type"],
    "quality": df["quality"],
})
proj.to_csv("wine_pca_py.csv", index=False)

pct = pca.explained_variance_ratio_ * 100.0
summary = pd.DataFrame({
    "component": [f"PC{i + 1}" for i in range(5)],
    "explained_variance": pca.explained_variance_.round(4),
    "pct_variance": pct.round(2),
    "cum_variance": pct.cumsum().round(2),
})
summary.to_csv(sys.stdout, index=False)
```

Three components align `pca_wine.py` with its R counterpart:

1. **Explicit matrix scaling:** Unlike R's `prcomp(scale. = TRUE)`, `scikit-learn`'s `PCA` centers features without scaling them; applying `StandardScaler().fit_transform` first ensures identical correlation-matrix eigenstructure.
2. **Truncated SVD projection:** `PCA(n_components=5, random_state=42)` extracts the top five orthogonal components and stores row projections (`pc1`, `pc2`) in `wine_pca_py.csv`.
3. **Standard-output serialization:** Writing `summary.to_csv(sys.stdout, index=False)` lets Nushell ingest `explained_variance`, `pct_variance`, and `cum_variance` directly via `from csv`.

We execute `pca_wine.py` and verify that Python and R agree to two decimal places on both marginal and cumulative variance:

```
let pca_py = python3 pca_wine.py | from csv

$pca_py
```

9.2.4 Multicollinearity Diagnostics (VIF) and Principal Component Regression (PCR)

Severe correlation among predictors inflates the variance of Ordinary Least Squares (OLS) slope estimates, making individual coefficients unstable under small sample perturbations. For each predictor X_j, the **Variance Inflation Factor** is defined as $\text{VIF}_j = 1/(1 - R_j^2)$, where R_j^2 is the coefficient of determination obtained by regressing X_j on all remaining predictors. Values above 10 signal acute collinearity.

Using morphological measurements of 159 fish specimens in `Fish.csv`, the script `vif_pcr_fish.py` computes exact VIF scores and contrasts full 5-predictor OLS against **Principal Component Regression (PCR)** on the first two orthogonal components:

```
bat vif_pcr_fish.py
```

```
#!/usr/bin/env python3
import sys
import numpy as np
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error, r2_score

df = pd.read_csv("Fish.csv")
pred_cols = ["Length1", "Length2", "Length3", "Height", "Width"]
X = df[pred_cols]
y = df["Weight"]
```

```
vif_rows = []
for col in pred_cols:
    X_other = X.drop(columns=[col])
    reg = LinearRegression().fit(X_other, X[col])
    r2_i = reg.score(X_other, X[col])
    vif = 1.0 / (1.0 - r2_i) if r2_i < 0.99999 else 9999.0
    vif_rows.append({"feature": col, "vif": round(float(vif), 2)})

pd.DataFrame(vif_rows).to_csv("fish_vif_py.csv", index=False)

ols = LinearRegression().fit(X, y)
ols_p = ols.predict(X)
ols_r2 = r2_score(y, ols_p)
ols_rmse = np.sqrt(mean_squared_error(y, ols_p))

X_pca = PCA(n_components=2).fit_transform(X)
pcr = LinearRegression().fit(X_pca, y)
pcr_p = pcr.predict(X_pca)
pcr_r2 = r2_score(y, pcr_p)
pcr_rmse = np.sqrt(mean_squared_error(y, pcr_p))

comp_df = pd.DataFrame([
    {
        "model": "Python_OLS_Multiple_Regression",
        "components": 5,
        "r2": round(float(ols_r2), 4),
        "rmse": round(float(ols_rmse), 2),
    },
    {
        "model": "Python_PCR_Principal_Components",
        "components": 2,
        "r2": round(float(pcr_r2), 4),
        "rmse": round(float(pcr_rmse), 2),
    },
])
comp_df.to_csv("fish_pcr_py.csv", index=False)
comp_df.to_csv(sys.stdout, index=False)
```

Three algorithmic blocks drive `vif_pcr_fish.py`:

1. **Auxiliary predictor regressions (`vif_rows`):** Loops over the five morphometric predictors, regressing each column `X[col]` against `X_other` to compute $1/(1 - R_i^2)$ and exporting the diagnostic table to `fish_vif_py.csv`.
2. **Baseline OLS fit:** Estimates unconstrained multiple linear regression on all five correlated predictors, recording in-sample R^2 and root mean squared error (`ols_rmse`).
3. **Orthogonal PCR reduction:** Projects `X` onto two uncorrelated principal axes (`PCA(n_-`

components=2)) before fitting LinearRegression, writing comparative metrics to fish_pcr_py.csv and sys.stdout.

Running the script in Nushell reveals extreme collinearity among the three length measurements (Length1, Length2, Length3 all exceed VIF $> 1{,}200$), whereas two orthogonal components preserve $R^2 \approx 0.87$:

```
python3 vif_pcr_fish.py | from csv

open fish_vif_py.csv | sort-by vif -r
```

We cross-validate the same PCR comparison in R via vif_pcr_fish.R:

```
bat vif_pcr_fish.R
```

```
#!/usr/bin/env Rscript
df <- read.csv("Fish.csv")
pred_cols <- c("Length1", "Length2", "Length3", "Height", "Width")
X <- df[, pred_cols]
y <- df$Weight

fit_ols <- lm(Weight ~ ., data = df[, c(pred_cols, "Weight")])
r2_ols <- summary(fit_ols)$r.squared
rmse_ols <- sqrt(mean(residuals(fit_ols)^2))

pca <- prcomp(X, scale. = TRUE)
fit_pcr <- lm(y ~ pca$x[, 1:2])
r2_pcr <- summary(fit_pcr)$r.squared
rmse_pcr <- sqrt(mean(residuals(fit_pcr)^2))

out_comp <- data.frame(
  model = c(
    "R_OLS_Multiple_Regression", "R_PCR_Principal_Components"
  ),
  components = c(5, 2),
  r2 = round(c(r2_ols, r2_pcr), 4),
  rmse = round(c(rmse_ols, rmse_pcr), 2)
)
write.csv(
  out_comp, "fish_pcr_r.csv", row.names = FALSE, quote = FALSE
)
write.csv(out_comp, stdout(), row.names = FALSE, quote = FALSE)
```

In vif_pcr_fish.R, lm(Weight ~ .) fits the full five-variable linear model, prcomp(X, scale. = TRUE) computes standardized principal components (pca$x[, 1:2]), and lm(y ~ pca$x[, 1:2]) regresses fish weight onto those two orthogonal scores before streaming the comparison table to stdout():

```
Rscript vif_pcr_fish.R | from csv
```

9.3 Clustering

Clustering partitions unlabeled observations into cohesive groups so that records within the same cluster share similar feature profiles while records across clusters remain distinct. Because no response label guides the optimization, geometric scaling and post-hoc validation determine whether discovered partitions reflect genuine structure or numerical artifacts.

This section covers four complementary clustering families:

- **K-Means Clustering:** Iterative centroid relocation minimizing within-cluster sum of squares (WCSS).
- **Agglomerative Hierarchical Clustering:** Bottom-up dendrogram construction under Ward's minimum-variance linkage.
- **Multi-Metric and Visual Cluster Validation:** Selecting K jointly via inertia elbow curves, Silhouette coefficients, Calinski-Harabasz variance ratios, and Davies-Bouldin scores.
- **Density and Medoid Methods (DBSCAN and PAM):** Isolating morphological outliers as explicit noise points and replacing sensitive arithmetic means with actual sample medoids.

9.3.1 K-Means Clustering

Unlike `tapkee`, which ships as a standalone C++ binary, our tabular K-Means pipeline delegates numerical optimization to a compact, stream-aware Python helper (`kmeans_cluster.py`) built on `scikit-learn`. Keeping the numerical kernel in an explicit `.py` script rather than hiding inline Python strings inside shell functions lets us inspect, test, and invoke the clustering logic independently before wrapping it in Nushell.

We open `kmeans_cluster.py` in `/data/ch09` with `bat`:

```
bat kmeans_cluster.py
```

```
#!/usr/bin/env python3
"""Partition numeric columns via standardized K-Means clustering."""
import argparse
import sys
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler


def main():
    p = argparse.ArgumentParser(description="K-Means CLI helper.")
    p.add_argument("--input", "-i", default="-")
    p.add_argument("--k", "-k", type=int, default=2)
```

```
    p.add_argument("--exclude", "-e", default="type")
    p.add_argument("--cols", "-c", default="")
    args = p.parse_args()

    src = sys.stdin if args.input == "-" else args.input
    df = pd.read_csv(src)

    if args.cols:
        num_cols = [
            c for c in args.cols.split(",") if c in df.columns
        ]
    else:
        excl = set(
            x.strip() for x in args.exclude.split(",") if x.strip()
        )
        excl.add("type")
        num_cols = [
            c for c in df.columns
            if c not in excl and pd.api.types.is_numeric_dtype(df[c])
        ]

    X = StandardScaler().fit_transform(df[num_cols])
    km = KMeans(n_clusters=args.k, n_init=10, random_state=42)
    labels = km.fit_predict(X)
    df["cluster"] = [f"cluster_{c + 1}" for c in labels]
    df.to_csv(sys.stdout, index=False)


if __name__ == "__main__":
    main()
```

Three design decisions make `kmeans_cluster.py` behave like a native Unix filter:

1. **Stream or file ingestion (`--input -`):** Reading `sys.stdin` when `--input` defaults to `"-"` allows Nushell tables serialized with `to csv` to flow straight into Python without manual intermediate files, while `--cols` and `--exclude` prevent response labels (`quality`, `type`) from leaking into the feature matrix.
2. **Mandatory z-score standardization:** Euclidean distance is not scale-invariant; without `StandardScaler()`, `total_sulfur_dioxide` (ranging from 6 to 440 mg/L) would overwhelm `chlorides` (ranging from 0.009 to 0.611 g/L).
3. **Deterministic multi-start Lloyd iterations:** `KMeans(n_clusters=args.k, n_init=10, random_state=42)` runs ten independent centroid initializations (`k-means++`), retains the lowest-inertia solution, appends human-readable labels (`cluster_1`, …, `cluster_K`), and writes CSV output to `sys.stdout`.

Before introducing any Nushell wrapper, we run `kmeans_cluster.py` directly in a raw pipeline:

Nushell serializes `wine-both.csv` to CSV text on standard output, Python computes two standardized clusters excluding `type` and `quality`, and Nushell parses the returned stream via `from csv` to cross-tabulate clusters against true wine varieties in Polars:

```
let raw_km = open wine-both.csv
  | to csv
  | python3 kmeans_cluster.py --k 2 --exclude "type,quality"
  | from csv

$raw_km
  | select type quality alcohol volatile_acidity cluster
  | first 5

$raw_km
  | polars into-df
  | polars group-by cluster type
  | polars agg [
      (polars col quality | polars count | polars as n_wines)
      (polars col alcohol | polars mean | polars as mean_alcohol)
    ]
  | polars collect
  | polars into-nu
  | sort-by cluster type
```

Look at that contingency table: without ever seeing the `type` column, unsupervised Euclidean partitioning in standardized 11-dimensional chemical space places over 98.5% of red wines into one cluster and over 98.5% of white wines into the other.

To eliminate manual `to csv` and `from csv` plumbing and provide native Nushell tab-completion, `dsin_tools.nu` defines the wrapper **`kmeans-cluster`**. We inspect its implementation in the terminal via `view-code kmeans-cluster`:

```
view-code kmeans-cluster
```

```
# Partition numeric columns via standardized K-Means clustering.
export def kmeans-cluster [
  --k(-k): int = 2
  --clusters: int = 0
  --label-col(-l): string
  --cols(-c): list<string>
]: table -> table {
  let tbl = $in
  let n_k = if $clusters > 0 { $clusters } else { $k }
  let tmp_in = mktemp --suffix ".csv"
  $tbl | to csv | save -f $tmp_in

  let lbl = $label_col | default "type"
```

```
  let cols_flag = if ($cols != null) and (($cols | length) > 0) {
    ["--cols" ($cols | str join ",")]
  } else { [] }

  let bin_p = "/usr/local/bin/kmeans-cluster-py"
  let py_km = if ($bin_p | path exists) {
    $bin_p
  } else if ("/data/ch09/kmeans_cluster.py" | path exists) {
    "/data/ch09/kmeans_cluster.py"
  } else {
    "kmeans_cluster.py"
  }

  let km_args = [
    "--input" $tmp_in
    "--k" ($n_k | into string)
    "--exclude" $lbl
    ...$cols_flag
  ]
  let res = python3 $py_km ...$km_args | from csv
  rm -f $tmp_in
  $res
}
```

Inside `kmeans-cluster`, Nushell converts `--cols` from a typed `list<string>` into a comma-delimited CLI flag via list spreading (`...$cols_flag`), resolves the installed helper path (`/usr/local/bin/kmeans-cluster-py`) so the command works from any working directory, and cleans up `$tmp_in` before returning the enriched table.

Repeating the exact same analysis through `kmeans-cluster` reduces the entire workflow to a concise, strongly typed pipeline:

```
let features = [
  "fixed_acidity" "volatile_acidity" "citric_acid"
  "residual_sugar" "chlorides" "free_sulfur_dioxide"
  "total_sulfur_dioxide" "density" "ph" "sulphates" "alcohol"
]

let clustered = open wine-both.csv
  | kmeans-cluster --k 2 --cols $features

$clustered
  | select type quality alcohol volatile_acidity cluster
  | first 5

$clustered
```

```
  | select type quality alcohol volatile_acidity cluster
  | save -f wine_kmeans_cli.csv

$clustered
  | polars into-df
  | polars group-by cluster type
  | polars agg [
      (polars col quality | polars count | polars as n_vinos)
      (polars col alcohol | polars mean | polars as mean_alcohol)
    ]
  | polars collect
  | polars into-nu
  | sort-by cluster type
  | tee { save -f wine_kmeans_contingency.csv }
```

9.3.2 K-Means Diagnostics in R and Python

When reporting cluster solutions in statistical studies, we also need within-cluster sums of squares (`withinss`) and convergence diagnostics across candidate partitions. In R, `cluster_wine.R` fits a 20-start K-Means model (`nstart = 20`) and cross-tabulates cluster purity directly:

```
bat cluster_wine.R
```

```
#!/usr/bin/env Rscript
set.seed(42)
df <- read.csv("wine.csv", stringsAsFactors = FALSE)
num_cols <- setdiff(names(df), c("type", "quality"))
X <- scale(df[, num_cols])

km <- kmeans(X, centers = 2, nstart = 20)
df$cluster <- paste0("cluster_", km$cluster)

keep_cols <- c(
  "type", "quality", "alcohol", "volatile_acidity", "cluster"
)
write.csv(
  df[, keep_cols], "wine_kmeans_r.csv",
  row.names = FALSE, quote = FALSE
)

tab <- as.data.frame.matrix(table(df$cluster, df$type))
out <- data.frame(
  cluster = rownames(tab),
  size = as.integer(km$size),
  withinss = round(km$withinss, 2),
```

```
  red_count = tab$red,
  white_count = tab$white,
  dominant_type = ifelse(tab$red > tab$white, "red", "white")
)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

Notice how `cluster_wine.R` standardizes predictors via `scale()`, runs Hartigan-Wong optimization across 20 random starts (`kmeans(X, centers = 2, nstart = 20)`), and combines cluster sizes (`km$size`), within-cluster dispersion (`km$withinss`), and contingency counts (`table(df$cluster, df$type)`) into a two-row CSV summary emitted to `stdout()`:

```
let km_r = Rscript cluster_wine.R | from csv

$km_r

$km_r | save -f wine_clusters_r.csv
```

In Python, `cluster_wine.py` sweeps $K \in \{2, 3, 4, 5\}$ to track how inertia, mean silhouette width, and iteration counts evolve as partitions subdivide:

```
bat cluster_wine.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("wine.csv")
feat_cols = [c for c in df.columns if c not in ("type", "quality")]
X = StandardScaler().fit_transform(df[feat_cols])

rows = []
for k in [2, 3, 4, 5]:
    km = KMeans(n_clusters=k, n_init=10, random_state=42)
    labels = km.fit_predict(X)
    sil = silhouette_score(
        X, labels, sample_size=1500, random_state=42
    )
    if k == 2:
        keep = ["type", "quality", "alcohol", "volatile_acidity"]
        df_out = df[keep].copy()
        df_out["cluster"] = [f"cluster_{c + 1}" for c in labels]
        df_out.to_csv("wine_kmeans_py.csv", index=False)
    rows.append({
```

```
        "k_clusters": k,
        "inertia": round(float(km.inertia_), 2),
        "silhouette": round(float(sil), 4),
        "iterations": int(km.n_iter_),
    })

pd.DataFrame(rows).to_csv(sys.stdout, index=False)
```

Inside the loop, `silhouette_score(..., sample_size=1500, random_state=42)` estimates average cohesion-separation contrast in $O(m^2)$ time on a fixed subsample rather than incurring quadratic $6{,}497^2$ pairwise distance overhead, exporting row labels at $K = 2$ and streaming the 4-row metric table to `sys.stdout`:

```
let km_py = python3 cluster_wine.py | from csv

$km_py

$km_py | save -f wine_clusters_py.csv
```

9.3.3 Agglomerative Hierarchical Clustering

Whereas K-Means commits to a fixed K upfront, agglomerative hierarchical clustering builds a complete binary merge tree (dendrogram) from n singleton leaves up to a single root partition. Ward's minimum-variance criterion (`ward.D2`) merges at each step the pair of clusters that induces the smallest increase in total within-cluster sum of squares.

We start with the R implementation in `hclust_fish.R`:

```
bat hclust_fish.R
```

```
#!/usr/bin/env Rscript
df <- read.csv("Fish.csv", stringsAsFactors = FALSE)
num_cols <- c(
  "Weight", "Length1", "Length2", "Length3", "Height", "Width"
)
X <- scale(df[, num_cols])

d <- dist(X, method = "euclidean")
hc <- hclust(d, method = "ward.D2")
df$cluster <- paste0("cluster_", cutree(hc, k = 3))

keep <- c("Species", "Weight", "Height", "Width", "cluster")
write.csv(
  df[, keep], "fish_hclust_r.csv", row.names = FALSE, quote = FALSE
)
```

```
tab <- as.data.frame.matrix(table(df$cluster, df$Species))
dom_idx <- max.col(tab, ties.method = "first")
out <- data.frame(
  cluster = rownames(tab),
  total_samples = as.integer(rowSums(tab)),
  dominant_species = colnames(tab)[dom_idx],
  dominant_count = apply(tab, 1, max)
)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

Three operations define `hclust_fish.R`: `dist(X, method = "euclidean")` computes the 159×159 pairwise distance matrix across standardized morphometric variables, `hclust(d, method = "ward.D2")` constructs the variance-minimizing dendrogram, and `cutree(hc, k = 3)` slices the tree into three morphological groups whose dominant biological species are summarized via `max.col(tab)`:

```
let hc_r = Rscript hclust_fish.R | from csv

$hc_r

$hc_r | save -f fish_hclust_r.csv
```

The companion script `hclust_fish.py` evaluates Silhouette widths across tree cuts $K \in \{2, 3, 4, 5\}$ using `AgglomerativeClustering`:

```
bat hclust_fish.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.cluster import AgglomerativeClustering
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("Fish.csv")
num_cols = [
    "Weight", "Length1", "Length2", "Length3", "Height", "Width"
]
X = StandardScaler().fit_transform(df[num_cols])

rows = []
for k in [2, 3, 4, 5]:
    model = AgglomerativeClustering(n_clusters=k, linkage="ward")
    labels = model.fit_predict(X)
    sil = silhouette_score(X, labels)
    if k == 3:
```

```
        df_out = df[["Species", "Weight", "Height", "Width"]].copy()
        df_out["cluster"] = [f"cluster_{c + 1}" for c in labels]
        df_out.to_csv("fish_hclust_py.csv", index=False)
    rows.append({
        "linkage": "ward",
        "n_clusters": k,
        "silhouette": round(float(sil), 4),
    })

pd.DataFrame(rows).to_csv(sys.stdout, index=False)
```

After z-score scaling (`StandardScaler`), `AgglomerativeClustering(n_clusters=k, linkage="ward")` fits deterministic Ward merges, persists specimen assignments at $K = 3$, and streams Silhouette scores for every cut level to `sys.stdout`:

```
let hc_py = python3 hclust_fish.py | from csv

$hc_py
```

9.3.4 Multi-Metric Validation and Visual Elbow Diagnostics

Choosing the number of clusters K solely from monotone-decreasing WCSS inertia can be subjective when the elbow bends gradually. Rigorous unsupervised validation combines inertia with internal separation criteria:

- **Silhouette Score ($[-1, 1]$, higher is better):** Contrasts mean intra-cluster distance $a(i)$ against nearest-neighbor cluster distance $b(i)$ via $(b(i) - a(i))/\max\{a(i), b(i)\}$.
- **Calinski-Harabasz Index (higher is better):** Ratio of between-cluster dispersion to within-cluster dispersion adjusted for degrees of freedom.
- **Davies-Bouldin Index (lower is better):** Average worst-case similarity ratio between each cluster and its closest competitor.

On the retail banking dataset `banco.csv`, `cluster_eval.py` computes all four quantities across $K \in \{2, \ldots, 6\}$ over customer age, income, credit-card spending, and mortgage balance:

```
bat cluster_eval.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import (
    calinski_harabasz_score,
    davies_bouldin_score,
    silhouette_score,
)
```

```
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("banco.csv")
features = ["edad", "ingreso", "gasto_tarjeta", "hipoteca"]
X = StandardScaler().fit_transform(df[features])

rows = []
for k in range(2, 7):
    km = KMeans(n_clusters=k, n_init=10, random_state=42)
    labels = km.fit_predict(X)
    sil = silhouette_score(
        X, labels, sample_size=1000, random_state=42
    )
    ch = calinski_harabasz_score(X, labels)
    db = davies_bouldin_score(X, labels)
    rows.append({
        "k": k,
        "inertia": round(float(km.inertia_), 2),
        "silhouette": round(float(sil), 4),
        "calinski_harabasz": round(float(ch), 2),
        "davies_bouldin": round(float(db), 4),
    })

pd.DataFrame(rows).to_csv(sys.stdout, index=False)
```

In `cluster_eval.py`, `StandardScaler` equalizes currency and age units, the `range(2, 7)` loop evaluates each fitted `KMeans` partition on all three internal indices simultaneously, and `pd.DataFrame(rows).to_csv(sys.stdout, index=False)` emits a tidy 5-row evaluation table that we rank directly in Nushell:

```
let eval_b = python3 cluster_eval.py | from csv

$eval_b | save -f banco_cluster_eval.csv

$eval_b | sort-by silhouette -r
```

To inspect inertia curvature and silhouette peaks visually across a wider sweep ($K = 2 \ldots 10$) on `wine-both.csv`, we use the diagnostic plotting script `plot_kmeans_elbow.py`:

```
bat plot_kmeans_elbow.py
```

```
#!/opt/modern-ml/bin/python
"""Render K-Means inertia and silhouette curves (k = 2..10)."""
import os
import sys
import warnings
```

```
os.environ.setdefault("MPLCONFIGDIR", "/tmp/matplotlib_cache")
warnings.filterwarnings("ignore")

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score
from sklearn.preprocessing import StandardScaler


def main():
    csv_in = sys.argv[1] if len(sys.argv) > 1 else "wine-both.csv"
    png_out = (
        sys.argv[2] if len(sys.argv) > 2 else "kmeans_elbow.png"
    )

    df = pd.read_csv(csv_in)
    drop_c = [c for c in ("type", "quality") if c in df.columns]
    num_df = df.drop(columns=drop_c).select_dtypes(
        include=[np.number]
    )
    X = StandardScaler().fit_transform(num_df)

    ks = list(range(2, 11))
    inertias, sils = [], []
    for k in ks:
        km = KMeans(n_clusters=k, n_init=10, random_state=42)
        labels = km.fit_predict(X)
        inertias.append(km.inertia_)
        sil = silhouette_score(
            X, labels, sample_size=1500, random_state=42
        )
        sils.append(sil)

    best_k = ks[int(np.argmax(sils))]
    fig, (ax1, ax2) = plt.subplots(
        1, 2, figsize=(10.5, 4.2), dpi=130
    )

    ax1.plot(ks, inertias, "o-", color="#0284c7", lw=2, ms=6)
    ax1.axvline(best_k, color="#be123c", ls="--", lw=1.2)
    ax1.set_title("Within-Cluster Inertia", fontweight="bold")
```

```
    ax1.set_xlabel("Number of Clusters (k)")
    ax1.set_ylabel("Inertia (WCSS)")
    ax1.set_xticks(ks)
    ax1.grid(True, ls="--", lw=0.4, alpha=0.5)

    ax2.plot(ks, sils, "s-", color="#0f766e", lw=2, ms=6)
    ax2.axvline(
        best_k, color="#be123c", ls="--", lw=1.2,
        label=f"Optimal k = {best_k}"
    )
    ax2.set_title("Mean Silhouette Score", fontweight="bold")
    ax2.set_xlabel("Number of Clusters (k)")
    ax2.set_ylabel("Silhouette Score")
    ax2.set_xticks(ks)
    ax2.legend(loc="best", frameon=True)
    ax2.grid(True, ls="--", lw=0.4, alpha=0.5)

    fig.suptitle(
        f"K-Means Cluster Validation ({csv_in})",
        fontsize=11.5, fontweight="bold", y=1.01
    )
    fig.tight_layout()
    fig.savefig(png_out, bbox_inches="tight", dpi=130)
    print(f"Saved {png_out} (optimal k={best_k})")


if __name__ == "__main__":
    main()
```

Three components coordinate `plot_kmeans_elbow.py`: CLI positional defaults (`wine-both.csv` and `kmeans_elbow.png`) allow zero-argument execution or reuse on any numeric table, the `ks = range(2, 11)` sweep records both `km.inertia_` and subsampled `silhouette_score` values while locating `best_k = ks[np.argmax(sils)]`, and headless `matplotlib` (`Agg`) pairs the WCSS elbow curve (`ax1`) with the silhouette profile (`ax2`) marked at `best_k`.

We generate `kmeans_elbow.png` and inspect the dual-panel chart directly inside the terminal via `timg`:

```
python plot_kmeans_elbow.py

timg -g 80x25 kmeans_elbow.png
```

Both diagnostics tell a consistent story: WCSS drops steepest between $k = 2$ and $k = 4$, while mean silhouette peaks sharply at $k = 2$ (`0.273`, capturing the primary red-versus-white split) with a secondary shoulder at $k = 4$ corresponding to dry versus residual-sugar white wine styles.

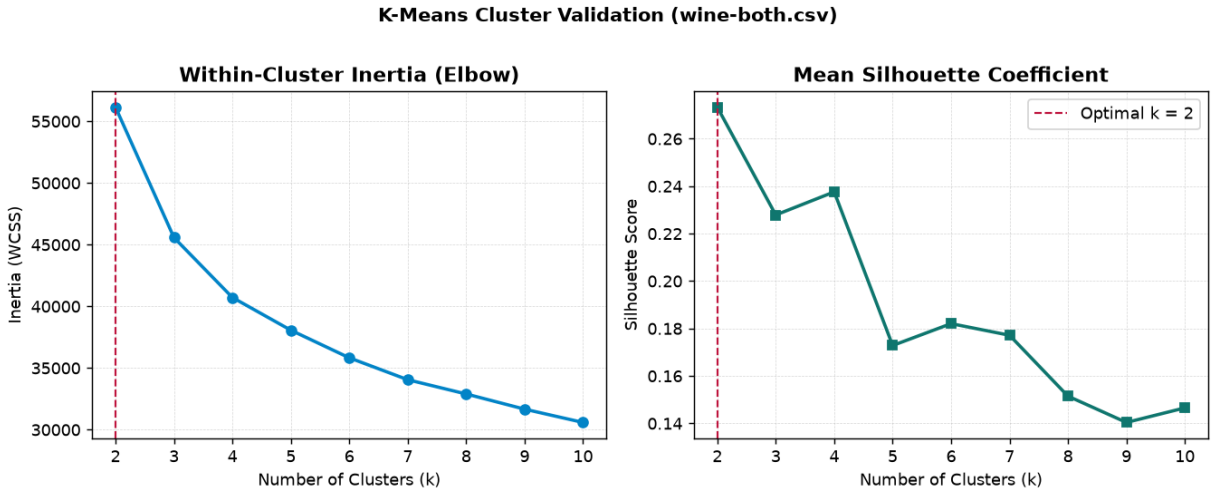


Figure 21: Dual-panel K-Means cluster validation showing within-cluster inertia and mean silhouette score across k = 2 to 10 on wine-both.csv

9.3.5 Density-Based Clustering with DBSCAN and K-Medoids with PAM

Centroid methods force every observation, no matter how extreme, into the nearest Voronoi cell, pulling cluster means toward outliers. **DBSCAN** (*Density-Based Spatial Clustering of Applications with Noise*) instead defines clusters as connected regions containing at least `min_samples` neighbors within radius ε (`eps`), labeling isolated observations in sparse regions as `-1` (`Noise / Outliers`).

Inspect `dbscan_fish.py` to see how neighborhood density separates core points from anomalies:

```
bat dbscan_fish.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.cluster import DBSCAN
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("Fish.csv")
num_cols = [
    "Weight", "Length1", "Length2", "Length3", "Height", "Width"
]
X_scaled = StandardScaler().fit_transform(df[num_cols])

db = DBSCAN(eps=0.9, min_samples=4)
df["cluster"] = db.fit_predict(X_scaled)

grouped = df.groupby("cluster").agg(
```

```
    n_samples=("Species", "count"),
    avg_weight=("Weight", lambda x: round(float(x.mean()), 1)),
    avg_length=("Length1", lambda x: round(float(x.mean()), 1)),
    avg_height=("Height", lambda x: round(float(x.mean()), 1)),
    avg_width=("Width", lambda x: round(float(x.mean()), 1)),
).reset_index()

grouped["cluster_type"] = grouped["cluster"].apply(
    lambda c: "Noise / Outliers" if c == -1 else f"Cluster {c}"
)
cols_out = [
    "cluster", "cluster_type", "n_samples",
    "avg_weight", "avg_length", "avg_height", "avg_width",
]
out = grouped[cols_out]
out.to_csv("fish_dbscan_py.csv", index=False)
out.to_csv(sys.stdout, index=False)
```

In `dbscan_fish.py`, `StandardScaler` normalizes the six morphometric dimensions so a uniform spherical radius `eps=0.9` applies equally across weight and length, `DBSCAN(eps=0.9, min_samples=4).fit_predict` assigns core/border points to non-negative cluster IDs and anomalies to `-1`, and `groupby("cluster").agg(...)` computes group-level morphological averages streamed to `sys.stdout`:

```
python3 dbscan_fish.py | from csv
```

Notice how heavy pike and perch outliers (`avg_weight` above 1,000 g) are cleanly quarantined into cluster `-1` (`Noise / Outliers`) rather than distorting the main body-plan group.

When a fixed number of partitions K is desired alongside outlier resistance, **Partitioning Around Medoids (PAM)** restricts cluster centers to actual observed specimens (*medoids*) and minimizes pairwise dissimilarities instead of squared deviations. Its R implementation lives in `pam_fish.R`:

```
bat pam_fish.R
```

```
#!/usr/bin/env Rscript
suppressPackageStartupMessages(library(cluster))
df <- read.csv("Fish.csv")
num_cols <- c(
  "Weight", "Length1", "Length2", "Length3", "Height", "Width"
)
X <- scale(df[, num_cols])

fit_pam <- pam(X, k = 3, metric = "euclidean")
sil <- silhouette(fit_pam)
avg_sil <- mean(sil[, 3])
```

```r
out <- data.frame(
  model = "R_cluster_pam_kmedoids",
  k_clusters = 3,
  avg_silhouette = round(avg_sil, 4),
  cluster_1_size = fit_pam$clusinfo[1, 1],
  cluster_2_size = fit_pam$clusinfo[2, 1],
  cluster_3_size = fit_pam$clusinfo[3, 1]
)
write.csv(
  out, "fish_pam_r.csv", row.names = FALSE, quote = FALSE
)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

Using R's `cluster` library, `pam(X, k = 3, metric = "euclidean")` runs the BUILD and SWAP medoid phases on standardized features, `silhouette(fit_pam)` extracts observation-level silhouette widths (`sil[, 3]`), and `fit_pam$clusinfo[, 1]` reports exact cluster cardinalities to standard output:

```
Rscript pam_fish.R | from csv
```

9.4 Regression: Predicting Wine Quality

Supervised regression learns a functional mapping $f : \mathcal{X} \to \mathbb{R}$ that predicts a continuous response from observed covariates. To estimate sensory wine quality ($y \in [0, 10]$) on the 4,898 white wine samples in `wine-white.csv`, we compare three distinct computational engines:

1. **Vowpal Wabbit (`vw`):** Out-of-core online linear optimization driven by stochastic gradient descent and feature hashing.
2. **Multiple Linear Regression in R (`regress_wine.R`):** Exact Ordinary Least Squares (OLS) estimation with parametric t-tests and p-values.
3. **Penalized Linear and Ensemble Regression in Python (`regress_wine.py`):** Standardized L_2 Ridge regression benchmarked against non-linear Random Forest ensembles.

9.4.1 Online Learning with Vowpal Wabbit

In-memory statistical packages allocate dense design matrices in RAM before computing a single gradient step. **Vowpal Wabbit** (Langford et al., 2007) reverses that architecture: it streams observations one line at a time from standard input or a binary cache file, mapping arbitrary feature names into a fixed-size weight vector via the hashing trick and updating weights on the fly with online stochastic gradient descent (SGD).

Every input record in Vowpal Wabbit obeys a plain-text wire grammar:

```
[label] [weight] [tag]|Namespace feat1:val1 feat2:val2 ...
```

A white wine record with sensory quality `6`, for instance, serializes its eleven physicochemical predictors behind a namespace marker `|f` (split across two lines here for readability, though stored as one line on disk):

```
6 |f fixed_acidity:7 volatile_acidity:0.27 citric_acid:0.36
  residual_sugar:20.7 chlorides:0.045 alcohol:8.8 ...
```

Three syntactic tokens govern how `vw` parses each record:

- `6`: Continuous target response (`quality`).
- `|f`: Vertical bar separating response metadata from predictor namespaces, where `f` names the feature group (enabling fast namespace-level quadratic interactions when desired).
- `fixed_acidity:7 volatile_acidity:0.27 ...`: Whitespace-delimited `name:value` pairs hashed directly into integer indices without requiring an upfront feature dictionary.

9.4.1.1 Direct Terminal Pre-Processing, Execution, and Post-Processing

Before introducing any wrapper command, we run the entire Vowpal Wabbit lifecycle directly from Nushell. First, split `wine-white.csv` into 3,500 training rows and 1,398 held-out test rows, format every record into `<quality> |f <col>:<val>` lines, and check the resulting text file:

```
let data = open wine-white.csv

let train = $data | first 3500

let test = $data | skip 3500

let fcols = $data | columns | where $it != "quality"

$train
| each { |r|
    let kv = $fcols
      | each { |c| $"($c):($r | get $c)" }
      | str join " "
    $"($r.quality) |f ($kv)"
  }
| str join "\n"
| save -f /tmp/direct_train.vw

$test
| each { |r|
    let kv = $fcols
      | each { |c| $"($c):($r | get $c)" }
      | str join " "
```

```
    $"($r.quality) |f ($kv)"
  }
| str join "\n"
| save -f /tmp/direct_test.vw

open --raw /tmp/direct_train.vw
| lines
| first 2
| each { str substring 0..64 }
```

With both `.vw` files on disk, we invoke the native C++ `vw` binary twice: once to optimize squared-error weights across 15 passes (caching parsed binary features in `/tmp/direct.cache` for fast multi-epoch sweeps), and once in frozen test mode (`-t`) to emit scalar predictions into `/tmp/direct_preds.txt`:

```
(
  vw --data /tmp/direct_train.vw
    --final_regressor /tmp/direct_model.vw
    --passes 15 -l 0.5
    --cache_file /tmp/direct.cache --quiet
)

(
  vw --data /tmp/direct_test.vw
    -i /tmp/direct_model.vw -t
    -p /tmp/direct_preds.txt --quiet
)
```

Five flags control this two-phase CLI exchange:

- `--final_regressor`: Writes learned weights to a compact binary model file.
- `--passes 15` and `--cache_file`: Streams 15 epochs over the binary cache instead of re-parsing ASCII text on every pass.
- `-l 0.5`: Sets the base SGD step size.
- `-i` and `-t`: Reloads the trained regressor and disables gradient updates during test-set evaluation.
- `-p`: Writes one newline-delimited floating-point prediction per test sample.

Because `vw` emits raw numbers without column headers or original targets, we parse `/tmp/direct_preds.txt` in Nushell, pair each prediction with `$test.quality` via `zip`, compute signed and absolute residuals, and remove our temporary files:

```
let raw_p = open --raw /tmp/direct_preds.txt
  | lines
  | where ($it | str trim | is-not-empty)
  | each { into float }
```

```
let obs_y = $test.quality | each { into float }

let direct_vw = $obs_y | zip $raw_p | each { |pair|
  let obs = $pair.0
  let pred = $pair.1 | math round -p 4
  {
    observed: $obs
    predicted: $pred
    error: (($pred - $obs) | math round -p 4)
    abs_error: (($obs - $pred) | math abs | math round -p 4)
  }
}

rm -rf /tmp/direct_train.vw /tmp/direct_test.vw

rm -rf /tmp/direct_model.vw /tmp/direct_preds.txt /tmp/direct.cache

$direct_vw | first 5
```

9.4.1.2 Encapsulating the Workflow in `to-vw` and `vw-regress`

Repeating string-interpolation loops, cache paths, and residual joins by hand across multiple experiments invites copy-paste errors. In `dsin_tools.nu`, two composable Nushell functions automate those exact stages. First, **`to-vw`** generalizes the row serializer for any target column and namespace tag. We inspect its implementation in the terminal via `view-code to-vw`:

```
view-code to-vw
```

```
# Convert a Nushell table into Vowpal Wabbit line format.
export def to-vw [
  --target(-t): string = "quality"
  --namespace(-n): string = "f"
]: table -> list<string> {
  let tbl = $in
  let cols = $tbl | columns | where $it != $target
  $tbl | each { |row|
    let y = $row | get $target
    let feats = $cols
      | each { |c| $"($c):($row | get $c)" }
      | str join " "
    $"($y) |($namespace) ($feats)"
  }
}
```

Second, **`vw-regress`** calls `to-vw` on both input tables inside an isolated `mktemp -d` workspace, runs the two `vw` passes, reconstructs the typed residual table via `zip`, and cleans up all binary artifacts before returning:

```
# Train and evaluate a Vowpal Wabbit regression model.
export def vw-regress [
  train_tbl: table
  test_tbl: table
  --target(-t): string = "quality"
  --passes(-p): int = 10
  --learning-rate(-l): float = 0.5
]: nothing -> table {
  let tmp_dir = mktemp -d
  let train_vw = $"($tmp_dir)/train.vw"
  let test_vw = $"($tmp_dir)/test.vw"
  let model_file = $"($tmp_dir)/model.vw"
  let pred_file = $"($tmp_dir)/preds.txt"
  let cache_file = $"($tmp_dir)/train.cache"

  $train_tbl
    | to-vw --target $target
    | str join "\n"
    | save -f $train_vw
  $test_tbl
    | to-vw --target $target
    | str join "\n"
    | save -f $test_vw

  let train_args = [
    "--data" $train_vw
    "--final_regressor" $model_file
    "--passes" ($passes | into string)
    "-l" ($learning_rate | into string)
    "--cache_file" $cache_file
    "--quiet"
  ]
  vw ...$train_args
  vw --data $test_vw -i $model_file -t -p $pred_file --quiet

  let preds = open --raw $pred_file
    | lines
    | where ($it | str trim | is-not-empty)
    | each { |x| $x | into float }
  let actuals = $test_tbl
    | get $target
    | each { |x| $x | into float }
```

```
  rm -rf $tmp_dir

  $actuals | zip $preds | each { |pair|
    let obs = $pair.0
    let pred = $pair.1 | math round -p 4
    let signed_err = ($pred - $obs) | math round -p 4
    let err = ($obs - $pred) | math abs | math round -p 4
    {
      actual: $obs
      observed: $obs
      predicted: $pred
      error: $signed_err
      abs_error: $err
    }
  }
}
```

9.4.1.3 Comparative Rerun with `to-vw` and `vw-regress`

Test `to-vw` on a single wine sample first, then rerun the exact 3,500/1,398 regression experiment through `vw-regress`:

```
open wine-white.csv
| first 1
| to-vw --target quality
| each { str substring 0..64 }

let data = open wine-white.csv

let train = $data | first 3500

let test = $data | skip 3500

let preds = vw-regress $train $test --target quality --passes 15

$preds | first 5

$preds | save -f wine_vw_predictions.csv
```

Notice that the `observed`, `predicted`, `error`, and `abs_error` values match our manual `vw` invocation to four decimal places. From `wine_vw_predictions.csv`, we summarize held-out Mean Absolute Error (MAE) and mean signed bias directly in Nushell:

```
let preds = open wine_vw_predictions.csv

let mae_v = $preds.abs_error | math avg | math round -p 4
```

```
let bias_v = $preds.error | math avg | math round -p 4

let metrics = {
  model: "Vowpal Wabbit (SGD)"
  n_test: ($preds | length)
  mae: $mae_v
  bias: $bias_v
}

$metrics

[$metrics] | save -f wine_vw_metrics.csv
```

9.4.2 Multiple Linear Regression with `Rscript`

Online SGD excels at throughput, yet statistical inference often demands exact standard errors, t-ratios, and p-values for individual predictors. Read `regress_wine.R` with `bat`:

```
bat regress_wine.R
```

```
#!/usr/bin/env Rscript
set.seed(42)
df <- read.csv("wine-white.csv", stringsAsFactors = FALSE)
n <- nrow(df)
idx <- sample(seq_len(n), size = floor(0.8 * n))
train <- df[idx, ]
test  <- df[-idx, ]

fit <- lm(quality ~ ., data = train)
preds <- predict(fit, newdata = test)

# Save predictions vs observed
res_df <- data.frame(
  observed = test$quality,
  predicted = round(preds, 4),
  abs_error = round(abs(test$quality - preds), 4)
)
write.csv(
  res_df, "wine_lm_predictions_r.csv",
  row.names = FALSE, quote = FALSE
)

# Output model coefficients to stdout as CSV
coefs <- summary(fit)$coefficients
```

```
out <- data.frame(
  term = rownames(coefs),
  estimate = round(coefs[, "Estimate"], 5),
  std_error = round(coefs[, "Std. Error"], 5),
  t_value = round(coefs[, "t value"], 3),
  t_stat = round(coefs[, "t value"], 3),
  p_value = signif(coefs[, "Pr(>|t|)"], 4)
)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

Three analytical stages organize `regress_wine.R`:

1. **Seeded 80/20 partition and QR-based OLS fit:** Fixing `set.seed(42)` draws an exact 80% training sample (`idx`) before `lm(quality ~ ., data = train)` solves the normal equations via QR decomposition across all eleven physicochemical predictors.
2. **Out-of-sample residual persistence:** `predict(fit, newdata = test)` evaluates held-out white wines and writes point predictions alongside absolute errors to `wine_lm_predictions_r.csv` (we will diagnose full OLS residual geometry visually in Section 9.5).
3. **Parametric inference table on `stdout()`:** Extracting `summary(fit)$coefficients` exposes $\hat{\beta}_j$, $\mathrm{SE}(\hat{\beta}_j)$, t-statistics, and two-sided p-values as clean CSV text for Nushell ingestion.

Executing `regress_wine.R` and piping its standard output into `from csv` lets us rank statistically significant predictors ($\alpha = 0.01$) by effect magnitude:

```
Rscript regress_wine.R | from csv | save -f wine_regression_r.csv

let sig = open wine_regression_r.csv
  | where p_value < 0.01
  | sort-by t_stat -r

$sig

$sig | save -f wine_significant_predictors.csv
```

9.4.3 Penalized Ridge and Ensemble Regression with Python

Because density, residual sugar, and alcohol exhibit strong collinearity in white wines, unpenalized OLS coefficients can suffer inflated variance, whereas linear models miss non-linear threshold interactions altogether. The script `regress_wine.py` benchmarks an L_2-regularized `Ridge` pipeline against a 100-tree `RandomForestRegressor` on an identical 80/20 split:

```
bat regress_wine.py
```

```
#!/usr/bin/env python3
import sys
import numpy as np
```

```
import pandas as pd
from sklearn.ensemble import RandomForestRegressor
from sklearn.linear_model import Ridge
from sklearn.metrics import (
    mean_absolute_error,
    mean_squared_error,
    r2_score,
)
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("wine-white.csv")
X = df.drop(columns=["quality"])
y = df["quality"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

models = {
    "Ridge(alpha=1.0)": make_pipeline(
        StandardScaler(), Ridge(alpha=1.0)
    ),
    "RandomForest(n=100)": RandomForestRegressor(
        n_estimators=100, random_state=42, n_jobs=4
    ),
}

rows = []
for name, model in models.items():
    model.fit(X_train, y_train)
    preds = model.predict(X_test)
    mae = mean_absolute_error(y_test, preds)
    rmse = np.sqrt(mean_squared_error(y_test, preds))
    r2 = r2_score(y_test, preds)
    rows.append({
        "model": name,
        "mae": round(float(mae), 4),
        "rmse": round(float(rmse), 4),
        "r2": round(float(r2), 4),
    })

out_df = pd.DataFrame(rows)
out_df.to_csv("wine_reg_metrics_py.csv", index=False)
out_df.to_csv(sys.stdout, index=False)
```

Three design choices make `regress_wine.py` reliable inside a shell pipeline:

1. **Leakage-free standardization (`make_pipeline`):** Wrapping `StandardScaler()` and `Ridge(alpha=1.0)` inside a single pipeline ensures predictor means and standard deviations are estimated strictly on `X_train` before shrinking collinear weights under the L_2 penalty $\alpha\|\beta\|_2^2$.
2. **Non-linear ensemble baseline (`RandomForestRegressor`):** Fitting 100 bootstrap decision trees (`n_jobs=4`) captures higher-order interactions among alcohol, volatile acidity, and free sulfur dioxide without manual feature engineering.
3. **Dual CSV emission:** Writing held-out `mae`, `rmse`, and `r2` both to `wine_reg_metrics_py.csv` and `sys.stdout` lets Nushell compare parametric and non-parametric generalization in one line:

```
python regress_wine.py | from csv

python regress_wine.py | from csv | save -f wine_regression_py.csv
```

9.5 Supervised Classification

Supervised classification learns a discrete decision rule $f : \mathcal{X} \to \{1, \ldots, C\}$ separating categorical classes such as `red` versus `white` wines in `wine-both.csv` or personal loan subscribers (`prestamo` $\in \{0, 1\}$) in the bank telemarketing dataset `banco.csv`. Across the subsections below, we progress from declarative CLI experiment runners to parametric, non-parametric, kernel, and neural classifiers in R and Python.

9.5.1 Classification Experiments with SKLL

9.5.1.1 Declarative Experimentation with SciKit-Learn Laboratory

SKLL (*SciKit-Learn Laboratory*, pronounced *skull*) is a command-line framework created by Educational Testing Service (ETS) to run repeatable, multi-learner `scikit-learn` benchmarks from plain-text `.cfg` specifications without writing glue scripts (https://skll.readthedocs.io/).

Rather than reading an arbitrary single file, SKLL expects separate `train/` and `test/` directories containing aligned feature tables with an explicit sample identifier (`id`) and target column (`type`). In `/data/ch09`, `train/features.csv` and `test/features.csv` are pre-populated, yet constructing that balanced split from scratch in Nushell takes only a single pipeline. We build an isolated replica under `/tmp/skll_demo` to see every pre-processing step explicitly, and inspect the schema of `train/features.csv`:

```
mkdir /tmp/skll_demo/train /tmp/skll_demo/test

let red = open wine-red.csv | insert type "red"
```

```
let n_red = $red | length

let white_s = open wine-white.csv
  | insert type "white"
  | first $n_red

let bal = $red
  | append $white_s
  | enumerate
  | each { |r| $r.item | insert id $r.index }

let n_test = ($bal | length) * 0.2 | into int

$bal | first $n_test | save -f /tmp/skll_demo/test/features.csv

$bal | skip $n_test | save -f /tmp/skll_demo/train/features.csv

rm -rf /tmp/skll_demo

open train/features.csv
| select id type alcohol volatile_acidity chlorides
| first 3
```

Now examine `classify.cfg` in `/data/ch09` via `bat`:

```
bat classify.cfg
```

```
[General]
experiment_name = wine
task = evaluate

[Input]
train_directory = train
test_directory = test
featuresets = [["features"]]
feature_scaling = both
label_col = type
id_col = id
shuffle = true
learners = [
    "KNeighborsClassifier",
    "LogisticRegression",
    "DecisionTreeClassifier",
    "RandomForestClassifier"
  ]
suffix = .csv
```

```
[Tuning]
grid_search = false
objectives = ["accuracy"]
param_grids = [{}, {}, {}, {}]

[Output]
logs = output
results = output
predictions = output
models = output
```

Four INI sections control the benchmark:

1. **`[General]`:** Names the experiment (`wine`) and sets `task = evaluate` so SKLL fits each learner on `train_directory` and scores it once on `test_directory`.
2. **`[Input]`:** Points to `train/features.csv` and `test/features.csv` (`featuresets` + `suffix`), applies z-score scaling to predictors (`feature_scaling = both`), identifies `id` and `type`, and lists four competing `scikit-learn` classifiers.
3. **`[Tuning]`:** Disables hyperparameter grid search (`grid_search = false`) for deterministic baseline comparison while scoring `accuracy`.
4. **`[Output]`:** Routes serialized models, per-sample predictions, JSON results, and the consolidated `output/wine_summary.tsv` report into `output/`.

9.5.1.2 Direct CLI Execution and Nushell TSV Post-Processing

Next, invoke the native `run_experiment` binary directly, inspect the files emitted into `output/`, and parse `output/wine_summary.tsv` with Nushell's native TSV reader:

```
run_experiment -l classify.cfg out+err> /dev/null

ls output/wine* | select name size | first 5

open output/wine_summary.tsv
| select learner_name feature_scaling accuracy
| sort-by accuracy -r
```

Because SKLL writes over thirty diagnostic columns into `wine_summary.tsv` (timestamps, fold identifiers, confusion matrices, objective names, and system metadata), interactive comparison requires projecting `learner_name` and `accuracy` after every run.

9.5.1.3 Encapsulating TSV Projection in `skll-metrics`

In `dsin_tools.nu`, **`skll-metrics`** wraps default path resolution and column projection into a single zero-boilerplate helper. We inspect its implementation in the terminal via `view-code skll-metrics`:

```
view-code skll-metrics
```

```
# Parse SKLL experiment summary TSV into a clean Nushell table.
export def skll-metrics [
  summary_tsv: path = "output/wine_summary.tsv"
]: nothing -> table {
  let target_path = if ($summary_tsv | path exists) {
    $summary_tsv
  } else if ("output/wine_summary.tsv" | path exists) {
    "output/wine_summary.tsv"
  } else {
    $summary_tsv
  }
  open $target_path | select learner_name accuracy
}
```

9.5.1.4 Comparative Rerun with `skll-metrics`

Calling `skll-metrics` after `run_experiment` extracts the exact four-model leaderboard directly into a clean Nushell table ready for CSV persistence:

```
run_experiment -l classify.cfg out+err> /dev/null

skll-metrics output/wine_summary.tsv

skll-metrics output/wine_summary.tsv
| save -f wine_skll_summary.csv
```

9.5.2 Binomial Logistic Regression with `Rscript`

To complement SKLL's automated suite with an explicit parametric GLM on the exact same `train/features.csv` and `test/features.csv` splits, examine `classify_wine.R`:

```
bat classify_wine.R
```

```
#!/usr/bin/env Rscript
train <- read.csv("train/features.csv", stringsAsFactors = FALSE)
test  <- read.csv("test/features.csv", stringsAsFactors = FALSE)

train$target <- ifelse(train$type == "red", 1, 0)
test$target  <- ifelse(test$type == "red", 1, 0)

predictors <- setdiff(names(train), c("id", "type", "target"))
formula_str <- paste(
  "target ~", paste(predictors, collapse = " + ")
```

```
)

fit <- glm(
  as.formula(formula_str),
  data = train,
  family = binomial(link = "logit")
)
probs <- predict(fit, newdata = test, type = "response")
preds <- ifelse(probs >= 0.5, 1, 0)

tp <- sum(preds == 1 & test$target == 1)
tn <- sum(preds == 0 & test$target == 0)
fp <- sum(preds == 1 & test$target == 0)
fn <- sum(preds == 0 & test$target == 1)

acc <- (tp + tn) / nrow(test)
prec <- tp / (tp + fp)
rec <- tp / (tp + fn)
f1 <- 2 * prec * rec / (prec + rec)

out <- data.frame(
  model = "R_glm_binomial",
  accuracy = round(acc, 4),
  precision_red = round(prec, 4),
  recall_red = round(rec, 4),
  f1_red = round(f1, 4),
  tp = tp, tn = tn, fp = fp, fn = fn
)
write.csv(out, "wine_clf_r.csv", row.names = FALSE, quote = FALSE)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

Three stages drive `classify_wine.R`:

1. **Programmatic formula construction:** `setdiff` strips non-predictive identifiers (`id`, `type`, `target`) before `paste` builds the full additive logit specification `target ~ fixed_acidity + ...`.
2. **IRLS maximum-likelihood estimation:** `glm(..., family = binomial(link = "logit"))` estimates log-odds weights via Iteratively Reweighted Least Squares, while `predict(..., type = "response")` maps linear predictors through the logistic sigmoid $\sigma(\eta) = (1 + e^{-\eta})^{-1}$ and thresholds probabilities at 0.5.
3. **Confusion-matrix decomposition:** Explicit counts of true positives (`tp`), true negatives (`tn`), false positives (`fp`), and false negatives (`fn`) accompany precision, recall, and F_1 on standard output:

```
Rscript classify_wine.R | from csv
```

```
Rscript classify_wine.R
| from csv
| save -f wine_classification_r.csv
```

9.5.3 Logistic and Random Forest Classification in Python

In Python, `classify_wine.py` evaluates both a standardized L_2-penalized `LogisticRegression` pipeline and a 100-tree `RandomForestClassifier` on the identical `train/features.csv` and `test/features.csv` partitions:

```
bat classify_wine.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
)
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler

train = pd.read_csv("train/features.csv")
test = pd.read_csv("test/features.csv")

features = [c for c in train.columns if c not in ("id", "type")]
X_train = train[features]
y_train = (train["type"] == "red").astype(int)
X_test = test[features]
y_test = (test["type"] == "red").astype(int)

models = {
    "Python_LogisticRegression": make_pipeline(
        StandardScaler(),
        LogisticRegression(max_iter=500, random_state=42),
    ),
    "Python_RandomForestClassifier": RandomForestClassifier(
        n_estimators=100, random_state=42, n_jobs=4
    ),
}
```

```
rows = []
for name, clf in models.items():
    clf.fit(X_train, y_train)
    preds = clf.predict(X_test)
    rows.append({
        "model": name,
        "accuracy": round(float(accuracy_score(y_test, preds)), 4),
        "precision_red": round(
            float(precision_score(y_test, preds)), 4
        ),
        "recall_red": round(float(recall_score(y_test, preds)), 4),
        "f1_red": round(float(f1_score(y_test, preds)), 4),
    })

out_df = pd.DataFrame(rows)
out_df.to_csv("wine_clf_py.csv", index=False)
out_df.to_csv(sys.stdout, index=False)
```

Excluding `id` and `type` mirrors the R specification, `make_pipeline(StandardScaler(), LogisticRegression(...))` prevents test-set scale leakage, and iterating over `models.items()` emits aligned `accuracy`, `precision_red`, `recall_red`, and `f1_red` rows to `sys.stdout`:

```
python classify_wine.py | from csv

python classify_wine.py
| from csv
| save -f wine_classification_py.csv
```

9.5.4 Decision Trees on Imbalanced Banking Data

Decision trees recursively partition predictor space along axis-aligned thresholds that minimize node impurity (Gini index or Shannon entropy), yielding human-readable credit rules. In `banco.csv`, where personal loan acceptance (`prestamo == 1`) is a minority class, accuracy alone can mask poor minority recall. Read the R implementation in `dt_banco.R`:

```
bat dt_banco.R
```

```
#!/usr/bin/env Rscript
suppressPackageStartupMessages(library(rpart))
set.seed(42)
df <- read.csv("banco.csv")
features <- c(
  "edad", "ingreso", "familia",
  "gasto_tarjeta", "educacion", "hipoteca", "prestamo"
)
```

```r
df_sub <- df[, features]

n <- nrow(df_sub)
train_idx <- sample(seq_len(n), size = floor(0.8 * n))
train <- df_sub[train_idx, ]
test  <- df_sub[-train_idx, ]

fit <- rpart(
  prestamo ~ ., data = train, method = "class",
  control = rpart.control(maxdepth = 4)
)
preds <- predict(fit, newdata = test, type = "class")

tbl <- table(test$prestamo, preds)
acc <- sum(diag(tbl)) / sum(tbl)
has_r1 <- "1" %in% rownames(tbl)
has_r0 <- "0" %in% rownames(tbl)
has_c1 <- "1" %in% colnames(tbl)
has_c0 <- "0" %in% colnames(tbl)
tp <- if (has_r1 && has_c1) tbl["1", "1"] else 0
fp <- if (has_r0 && has_c1) tbl["0", "1"] else 0
fn <- if (has_r1 && has_c0) tbl["1", "0"] else 0
prec <- if (tp + fp > 0) tp / (tp + fp) else 0
rec <- if (tp + fn > 0) tp / (tp + fn) else 0
f1 <- if (prec + rec > 0) 2 * prec * rec / (prec + rec) else 0

out <- data.frame(
  model = "R_rpart_decision_tree",
  max_depth = 4,
  accuracy = round(acc, 4),
  precision = round(prec, 4),
  recall = round(rec, 4),
  f1_score = round(f1, 4)
)
write.csv(out, "banco_dt_r.csv", row.names = FALSE, quote = FALSE)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

In `dt_banco.R`, `rpart(..., method = "class", control = rpart.control(maxdepth = 4))` caps tree depth at four levels to prevent leaf memorization on the six demographic and financial predictors, while explicit `has_r1`/`has_c1` guards protect confusion-matrix indexing if a shallow split predicts a single class:

```
Rscript dt_banco.R | from csv

Rscript dt_banco.R | from csv | save -f banco_dt_r.csv
```

Now examine the `scikit-learn` counterpart in `dt_banco.py`:

```
bat dt_banco.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
)
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier

df = pd.read_csv("banco.csv")
features = [
    "edad", "ingreso", "familia",
    "gasto_tarjeta", "educacion", "hipoteca",
]
X = df[features]
y = df["prestamo"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)

clf = DecisionTreeClassifier(max_depth=4, random_state=42)
clf.fit(X_train, y_train)
preds = clf.predict(X_test)

acc = accuracy_score(y_test, preds)
prec = precision_score(y_test, preds, zero_division=0)
rec = recall_score(y_test, preds, zero_division=0)
f1 = f1_score(y_test, preds, zero_division=0)

out = pd.DataFrame([{
    "model": "Python_DecisionTreeClassifier",
    "max_depth": 4,
    "accuracy": round(float(acc), 4),
    "precision": round(float(prec), 4),
    "recall": round(float(rec), 4),
    "f1_score": round(float(f1), 4),
}])
out.to_csv("banco_dt_py.csv", index=False)
```

```
out.to_csv(sys.stdout, index=False)
```

Using CART Gini impurity with `max_depth=4` and `zero_division=0` metric guards, `dt_banco.py` exposes identical evaluation columns so R and Python trees can be joined or diffed directly in Nushell:

```
python dt_banco.py | from csv

python dt_banco.py | from csv | save -f banco_dt_py.csv
```

9.5.5 k-Nearest Neighbors Classification

Unlike decision trees, which are invariant to monotonic feature rescaling, k-Nearest Neighbors (k-NN) assigns labels by majority vote among the k closest training points in Euclidean space. Without standardization, annual income (`ingreso`) and mortgage balances (`hipoteca`) completely dominate family size (`familia`) and education level (`educacion`). Open `knn_banco.R`:

```
bat knn_banco.R
```

```r
#!/usr/bin/env Rscript
suppressPackageStartupMessages(library(class))
set.seed(42)
df <- read.csv("banco.csv")
features <- c(
  "edad", "ingreso", "familia",
  "gasto_tarjeta", "educacion", "hipoteca"
)
X <- scale(df[, features])
y <- df$prestamo

n <- nrow(X)
train_idx <- sample(seq_len(n), size = floor(0.8 * n))
train_x <- X[train_idx, ]
test_x  <- X[-train_idx, ]
train_y <- y[train_idx]
test_y  <- y[-train_idx]

rows <- list()
for (k in c(3, 5, 7)) {
  preds <- knn(
    train = train_x, test = test_x, cl = train_y, k = k
  )
  tbl <- table(test_y, preds)
  acc <- sum(diag(tbl)) / sum(tbl)
  has_r1 <- "1" %in% rownames(tbl)
```

```
  has_r0 <- "0" %in% rownames(tbl)
  has_c1 <- "1" %in% colnames(tbl)
  has_c0 <- "0" %in% colnames(tbl)
  tp <- if (has_r1 && has_c1) tbl["1", "1"] else 0
  fp <- if (has_r0 && has_c1) tbl["0", "1"] else 0
  fn <- if (has_r1 && has_c0) tbl["1", "0"] else 0
  prec <- if (tp + fp > 0) tp / (tp + fp) else 0
  rec <- if (tp + fn > 0) tp / (tp + fn) else 0
  f1 <- if (prec + rec > 0) 2 * prec * rec / (prec + rec) else 0
  rows[[length(rows) + 1]] <- data.frame(
    model = "R_knn",
    k_neighbors = k,
    accuracy = round(acc, 4),
    precision = round(prec, 4),
    recall = round(rec, 4),
    f1_score = round(f1, 4)
  )
}
out <- do.call(rbind, rows)
write.csv(out, "banco_knn_r.csv", row.names = FALSE, quote = FALSE)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

After centering and scaling the six predictors (`scale(df[, features])`), `knn_banco.R` sweeps odd neighborhood sizes $k \in \{3, 5, 7\}$ (avoiding binary voting ties) via `class::knn` and stacks the confusion metrics with `do.call(rbind, rows)`:

```
Rscript knn_banco.R | from csv

Rscript knn_banco.R | from csv | save -f banco_knn_r.csv
```

Compare this with `knn_banco.py`, which refines standardization by fitting `StandardScaler()` strictly on `X_train` before transforming `X_test`:

```
bat knn_banco.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
)
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
```

```
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("banco.csv")
features = [
    "edad", "ingreso", "familia",
    "gasto_tarjeta", "educacion", "hipoteca",
]
X = df[features]
y = df["prestamo"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

rows = []
for k in [3, 5, 7]:
    clf = KNeighborsClassifier(n_neighbors=k)
    clf.fit(X_train_s, y_train)
    preds = clf.predict(X_test_s)
    acc = accuracy_score(y_test, preds)
    prec = precision_score(y_test, preds, zero_division=0)
    rec = recall_score(y_test, preds, zero_division=0)
    f1 = f1_score(y_test, preds, zero_division=0)
    rows.append({
        "model": "Python_KNeighborsClassifier",
        "k_neighbors": k,
        "accuracy": round(float(acc), 4),
        "precision": round(float(prec), 4),
        "recall": round(float(rec), 4),
        "f1_score": round(float(f1), 4),
    })

out_df = pd.DataFrame(rows)
out_df.to_csv("banco_knn_py.csv", index=False)
out_df.to_csv(sys.stdout, index=False)
```

Evaluating `knn_banco.py` from Nushell reveals how sensitive local distance voting is to neighborhood radius k:

```
python knn_banco.py | from csv

python knn_banco.py | from csv | save -f banco_knn_py.csv
```

9.5.6 Generative Naïve Bayes vs. RBF Kernel SVMs

To contrast a fast generative classifier that assumes conditional feature independence against a maximum-margin kernel boundary, examine `nb_svm_banco.py`:

```
bat nb_svm_banco.py
```

```
#!/usr/bin/env python3
import sys
import pandas as pd
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
)
from sklearn.model_selection import train_test_split
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import StandardScaler
from sklearn.svm import SVC

df = pd.read_csv("banco.csv")
features = [
    "edad", "ingreso", "familia",
    "gasto_tarjeta", "educacion", "hipoteca",
]
X = df[features]
y = df["prestamo"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

models = {
    "GaussianNB": GaussianNB(),
    "SVC(kernel=rbf)": SVC(kernel="rbf", C=1.0, random_state=42),
}

rows = []
for name, clf in models.items():
    clf.fit(X_train_s, y_train)
    preds = clf.predict(X_test_s)
    acc = accuracy_score(y_test, preds)
    prec = precision_score(y_test, preds, zero_division=0)
```

```
    rec = recall_score(y_test, preds, zero_division=0)
    f1 = f1_score(y_test, preds, zero_division=0)
    rows.append({
        "model": name,
        "accuracy": round(float(acc), 4),
        "precision": round(float(prec), 4),
        "recall": round(float(rec), 4),
        "f1_score": round(float(f1), 4),
    })

out_df = pd.DataFrame(rows)
out_df.to_csv("banco_nb_svm_py.csv", index=False)
out_df.to_csv(sys.stdout, index=False)
```

In `nb_svm_banco.py`, `GaussianNB()` models each standardized banking covariate with class-conditional univariate Gaussians $\mathcal{N}(\mu_{jc}, \sigma_{jc}^2)$, whereas `SVC(kernel="rbf", C=1.0)` lifts observations into a Gaussian radial basis function space $K(x, x') = \exp(-\gamma \|x - x'\|^2)$ to separate non-linear borrower segments:

```
python nb_svm_banco.py | from csv

python nb_svm_banco.py | from csv | save -f banco_nb_svm_py.csv

open banco_nb_svm_py.csv | sort-by f1_score -r
```

9.5.7 Feedforward Neural Networks: Multi-Layer Perceptron

Multi-Layer Perceptrons (MLPs) compose affine weight matrices with non-linear activation functions across hidden layers, approximating curved decision surfaces via backpropagation. Inspect `mlp_banco.py`, which compares a single-layer `(8,)` network against a two-layer `(16, 8)` ReLU architecture:

```
bat mlp_banco.py
```

```
#!/usr/bin/env python3
import sys
import warnings
warnings.filterwarnings("ignore")
import pandas as pd
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
)
```

```
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.preprocessing import StandardScaler

df = pd.read_csv("banco.csv")
features = [
    "edad", "ingreso", "familia",
    "gasto_tarjeta", "educacion", "hipoteca",
]
X = df[features]
y = df["prestamo"]

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42
)
scaler = StandardScaler()
X_train_s = scaler.fit_transform(X_train)
X_test_s = scaler.transform(X_test)

rows = []
for hidden in [(8,), (16, 8)]:
    clf = MLPClassifier(
        hidden_layer_sizes=hidden, max_iter=500, random_state=42
    )
    clf.fit(X_train_s, y_train)
    preds = clf.predict(X_test_s)
    acc = accuracy_score(y_test, preds)
    prec = precision_score(y_test, preds, zero_division=0)
    rec = recall_score(y_test, preds, zero_division=0)
    f1 = f1_score(y_test, preds, zero_division=0)
    rows.append({
        "model": f"Python_MLP_{hidden}",
        "accuracy": round(float(acc), 4),
        "precision": round(float(prec), 4),
        "recall": round(float(rec), 4),
        "f1_score": round(float(f1), 4),
        "iterations": int(clf.n_iter_),
    })

out = pd.DataFrame(rows)
out.to_csv("banco_mlp_py.csv", index=False)
out.to_csv(sys.stdout, index=False)
```

Three details matter in `mlp_banco.py`: input standardization keeps initial gradients out of saturation zones, `MLPClassifier(..., max_iter=500)` optimizes cross-entropy loss with Adam, and recording `int(clf.n_iter_)` alongside F_1 lets Nushell audit convergence cost directly:

```
python mlp_banco.py | from csv

python mlp_banco.py | from csv | save -f banco_mlp_py.csv
```

The two-layer `(16, 8)` network reaches an F_1-score near 0.90+ on minority loan acceptances. Finally, inspect R's single-hidden-layer quasi-Newton implementation in `mlp_banco.R`:

```
bat mlp_banco.R
```

```
#!/usr/bin/env Rscript
suppressPackageStartupMessages(library(nnet))
set.seed(42)
df <- read.csv("banco.csv")
features <- c(
  "edad", "ingreso", "familia",
  "gasto_tarjeta", "educacion", "hipoteca"
)
df[features] <- scale(df[features])

n <- nrow(df)
train_idx <- sample(seq_len(n), size = floor(0.8 * n))
train <- df[train_idx, c(features, "prestamo")]
test  <- df[-train_idx, c(features, "prestamo")]

fit <- nnet(
  prestamo ~ ., data = train,
  size = 6, decay = 0.01, maxit = 400, trace = FALSE
)
preds_prob <- predict(fit, newdata = test, type = "raw")
preds <- ifelse(preds_prob > 0.5, 1, 0)

tbl <- table(test[["prestamo"]], preds)
acc <- sum(diag(tbl)) / sum(tbl)
has_r1 <- "1" %in% rownames(tbl)
has_r0 <- "0" %in% rownames(tbl)
has_c1 <- "1" %in% colnames(tbl)
has_c0 <- "0" %in% colnames(tbl)
tp <- if (has_r1 && has_c1) tbl["1", "1"] else 0
fp <- if (has_r0 && has_c1) tbl["0", "1"] else 0
fn <- if (has_r1 && has_c0) tbl["1", "0"] else 0
prec <- if (tp + fp > 0) tp / (tp + fp) else 0
rec <- if (tp + fn > 0) tp / (tp + fn) else 0
f1 <- if (prec + rec > 0) 2 * prec * rec / (prec + rec) else 0

out <- data.frame(
  model = "R_nnet_neural_network",
```

```
  hidden_units = 6,
  accuracy = round(acc, 4),
  precision = round(prec, 4),
  recall = round(rec, 4),
  f1_score = round(f1, 4)
)
write.csv(out, "banco_mlp_r.csv", row.names = FALSE, quote = FALSE)
write.csv(out, stdout(), row.names = FALSE, quote = FALSE)
```

In `mlp_banco.R`, `nnet(..., size = 6, decay = 0.01, maxit = 400, trace = FALSE)` fits six logistic hidden units via BFGS optimization with L_2 weight decay (`0.01`) while suppressing iteration logs (`trace = FALSE`) so `stdout()` remains pure CSV:

```
Rscript mlp_banco.R | from csv

Rscript mlp_banco.R | from csv | save -f banco_mlp_r.csv
```

9.6 Modern Pipeline-Native Modeling

The classical command-line modeling utilities explored earlier in this chapter (`tapkee`, `vw`, and `run_experiment`) were designed for traditional POSIX text shells. Because standard Unix pipes transmit untyped byte streams rather than structured schemas, each legacy tool imposes an external serialization tax: `tapkee` expects headerless numeric matrices and strips class labels, `vw` requires custom row-by-row namespace formatting via `to-vw`, and `skll` demands rigid directory trees (`train/`, `test/`, `output/`) paired with `.cfg` configuration files on disk.

In a typed shell like Nushell, modeling engines can exchange structured tables (`table -> table`) directly across standard streams with minimal disk ceremony. Inside the `dsin` container, standalone compiled C++ engines (`duckdb` and `lightgbm`) sit alongside an isolated Python 3.12 environment at `/opt/modern-ml` (provisioned via the static `uv` package manager). To ensure consistent ergonomics, `/opt/modern-ml/bin` is prepended to `$env.PATH`: invoking `python` directly runs Python 3.12 with modern tools (`umap-learn`, `lightgbm`, `scikit-learn` 1.9+, `pandas`, and `matplotlib`), while legacy utilities requiring Python 3.9 (such as SKLL) invoke their dedicated interpreter via shebang without collision. Check the active versions directly from `/data/ch09`:

```
duckdb --version

uv --version

python --version

let py_chk = "import umap, lightgbm as g; print(g.__version__)"

python -c $py_chk
```

9.6.1 Topological Manifold Projection with UMAP

Principal Component Analysis restricts embeddings to orthogonal linear subspaces, while classical t-SNE optimizes local pairwise affinities at steep computational cost without preserving global inter-cluster distances. **Uniform Manifold Approximation and Projection** (**UMAP**, McInnes et al., 2018) bridges both regimes by modeling high-dimensional observations as a weighted fuzzy simplicial set over a locally connected Riemannian manifold. Minimizing fuzzy cross-entropy between the high-dimensional and low-dimensional topological graphs preserves fine-grained neighborhood structure (governed by `--neighbors`) alongside macro-scale separation across distinct subpopulations (governed by `--min-dist`).

9.6.1.1 Inspecting and Running `umap_reduce.py` Directly

Following our modular pipeline design, all manifold math lives inside the standalone Python script `umap_reduce.py` in `/data/ch09`. Display its source code with `bat`:

```
bat umap_reduce.py
```

```
#!/usr/bin/env python
"""UMAP manifold projection helper reading CSV from file or stdin."""
import argparse
import os
import sys
import warnings

os.environ.setdefault("NUMBA_CACHE_DIR", "/tmp/numba_cache")
warnings.filterwarnings("ignore")

import pandas as pd
from sklearn.preprocessing import StandardScaler
import umap


def main():
    p = argparse.ArgumentParser(
        description="UMAP reduction on CSV input."
    )
    p.add_argument("--input", "-i", type=str, default="-")
    p.add_argument("--neighbors", "-n", type=int, default=15)
    p.add_argument("--min-dist", "-m", type=float, default=0.1)
    p.add_argument("--target-dim", "-d", type=int, default=2)
    p.add_argument("--seed", "-s", type=int, default=42)
    p.add_argument("--label-col", "-l", type=str, default="type")
    p.add_argument("--cols", "-c", type=str, default="")
    args = p.parse_args()
```

```
    src = sys.stdin if args.input == "-" else args.input
    df = pd.read_csv(src)

    if args.cols:
        raw_c = [c.strip() for c in args.cols.split(",")]
        num_cols = [c for c in raw_c if c in df.columns]
    else:
        num_cols = [
            c for c in df.columns
            if c != args.label_col
            and pd.api.types.is_numeric_dtype(df[c])
        ]

    if not num_cols:
        sys.stderr.write("Error: No numeric columns found.\n")
        sys.exit(1)

    X = StandardScaler().fit_transform(df[num_cols])
    n_nb = min(args.neighbors, max(2, len(df) - 1))
    reducer = umap.UMAP(
        n_neighbors=n_nb,
        min_dist=args.min_dist,
        n_components=args.target_dim,
        random_state=args.seed,
    )
    emb = reducer.fit_transform(X)

    for idx in range(args.target_dim):
        df[f"umap_{idx + 1}"] = emb[:, idx].round(4)

    df.to_csv(sys.stdout, index=False)


if __name__ == "__main__":
    main()
```

Three architectural elements make `umap_reduce.py` stream-friendly:

1. **Dual stdin/file CLI contract (`argparse`):** Defaulting `--input` to `"-"` lets `pd.read_csv(sys.stdin)` consume piped CSV streams directly from Nushell without touching disk, while `--cols` and `--label-col` separate numeric predictors from categorical metadata.
2. **Standardized simplicial graph construction:** `StandardScaler().fit_transform(df[num_cols])` equalizes measurement units before `umap.UMAP(...)` clamps `n_neighbors` safely for small samples and optimizes the low-dimensional layout under a deterministic `--seed`.

3. **In-place column enrichment to `sys.stdout`:** Appending `umap_1, ..., umap_d` directly onto `df` preserves original labels (`type`, `quality`) in the emitted CSV stream.

Before calling any Nushell wrapper, pipe 200 wine records directly through `to csv | python umap_reduce.py ... | from csv`:

```
let fcols = open wine-both.csv
  | columns
  | where $it not-in ["type" "quality"]
  | str join ","

open wine-both.csv
| first 200
| to csv
| (
    python umap_reduce.py
      --neighbors 15 --min-dist 0.1
      --label-col type --cols $fcols
  )
| from csv
| select type quality umap_1 umap_2
| first 5
```

9.6.1.2 Encapsulating the Pipeline in `umap-reduce`

In `dsin_tools.nu`, **`umap-reduce`** wraps CSV serialization, list-to-flag formatting (`--cols`), binary/script path resolution, and `from csv` deserialization behind native Nushell type signatures (`table -> table`). We inspect its implementation in the terminal via `view-code umap-reduce`:

```
view-code umap-reduce
```

```
# Compute UMAP manifold coordinates on a numeric Nushell table.
export def umap-reduce [
  --neighbors(-n): int = 15
  --min-dist(-m): float = 0.1
  --target-dim(-d): int = 2
  --seed(-s): int = 42
  --label-col(-l): string = "type"
  --cols(-c): list<string>
]: table -> table {
  let tbl = $in
  let cols_flag = if ($cols != null) and (($cols | length) > 0) {
    ["--cols" ($cols | str join ",")]
  } else { [] }

  let tmp_in = mktemp --suffix ".csv"
```

```
  $tbl | to csv | save -f $tmp_in

  let bin_p = "/usr/local/bin/umap-reduce-py"
  let py_script = if ($bin_p | path exists) {
    $bin_p
  } else if ("/data/ch09/umap_reduce.py" | path exists) {
    "/data/ch09/umap_reduce.py"
  } else {
    "umap_reduce.py"
  }

  let umap_args = [
    "--input" $tmp_in
    "--neighbors" ($neighbors | into string)
    "--min-dist" ($min_dist | into string)
    "--target-dim" ($target_dim | into string)
    "--seed" ($seed | into string)
    "--label-col" $label_col
    ...$cols_flag
  ]
  let res = python $py_script ...$umap_args | from csv
  rm -f $tmp_in
  $res
}
```

9.6.1.3 Comparative Rerun and Visual Manifold Comparison

Rerunning the exact 200-wine projection through `umap-reduce` yields identical `umap_1` and `umap_2` coordinates in a single declarative expression:

```
let features = open wine-both.csv
  | columns
  | where $it not-in ["type" "quality"]

let umap_tbl = open wine-both.csv
  | first 200
  | umap-reduce --label-col type --cols $features
  | select type quality umap_1 umap_2

$umap_tbl | first 5

$umap_tbl | save -f wine_umap.csv
```

To see geometrically how linear variance maximization (PCA), local neighborhood attraction (t-SNE), and topological manifold projection (UMAP) arrange the same physicochemical space, in-

spect and run `plot_umap_comparison.py`:

```
bat plot_umap_comparison.py
```

```
#!/opt/modern-ml/bin/python
"""Compare PCA, t-SNE, and UMAP projections on 800 wines."""
import os
import warnings

os.environ.setdefault("NUMBA_CACHE_DIR", "/tmp/numba_cache")
os.environ.setdefault("MPLCONFIGDIR", "/tmp/matplotlib_cache")
warnings.filterwarnings("ignore")

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE
from sklearn.preprocessing import StandardScaler
import umap

df = pd.read_csv("wine-both.csv")
red = df[df["type"] == "red"].sample(n=400, random_state=42)
white = df[df["type"] == "white"].sample(n=400, random_state=42)
sub = pd.concat([red, white], ignore_index=True)

feat_cols = [c for c in sub.columns if c not in ("type", "quality")]
X = StandardScaler().fit_transform(sub[feat_cols])

c_pca = PCA(n_components=2, random_state=42).fit_transform(X)
c_tsne = TSNE(
    n_components=2, perplexity=30, random_state=42, init="pca"
).fit_transform(X)
c_umap = umap.UMAP(
    n_neighbors=15, min_dist=0.1, n_components=2, random_state=42
).fit_transform(X)

fig, axes = plt.subplots(1, 3, figsize=(13.5, 4.2), dpi=130)
panels = [
    (c_pca, "PCA (Linear Variance)", "PC 1", "PC 2"),
    (c_tsne, "t-SNE (Local Neighborhoods)", "Dim 1", "Dim 2"),
    (c_umap, "UMAP (Local + Global Topology)", "UMAP 1", "UMAP 2"),
]
palette = {"red": "#be123c", "white": "#0284c7"}
```

```
for ax, (emb, title, xlabel, ylabel) in zip(axes, panels):
    for label, color in palette.items():
        mask = (sub["type"] == label).to_numpy()
        ax.scatter(
            emb[mask, 0], emb[mask, 1],
            c=color, label=label.capitalize(),
            s=14, alpha=0.72, edgecolors="none",
        )
    ax.set_title(title, fontsize=10.5, fontweight="bold")
    ax.set_xlabel(xlabel, fontsize=9)
    ax.set_ylabel(ylabel, fontsize=9)
    ax.grid(True, linestyle="--", linewidth=0.4, alpha=0.45)

axes[2].legend(
    title="Wine Type", loc="best", frameon=True, fontsize=8.5
)
fig.suptitle(
    "Manifold Projection Comparison on Wine Attributes (n = 800)",
    fontsize=12, fontweight="bold", y=1.01,
)
fig.tight_layout()
fig.savefig("umap_comparison.png", bbox_inches="tight", dpi=130)
print("Saved umap_comparison.png")
```

Sampling 400 red and 400 white wines (`random_state=42`), standardizing the eleven continuous predictors, and rendering a three-panel headless `Agg` figure (`umap_comparison.png`) lets us compare all three embeddings directly in the terminal via `timg`:

```
python plot_umap_comparison.py

timg -g 80x25 umap_comparison.png
```

9.6.2 Gradient Boosted Decision Trees from the CLI with LightGBM

Vowpal Wabbit excels when streaming millions of sparse text or click-through indicators that exceed system RAM, yet dense tabular benchmarks with a dozen continuous predictors rarely benefit from feature hashing or purely additive linear weights. **LightGBM** (Ke et al., 2017) compiles to a standalone C++ CLI binary (`/usr/local/bin/lightgbm`) that discretizes continuous predictors into 256-bin histograms and grows regression trees leaf-wise (*best-first*) rather than level-wise. Because `lightgbm` parses headered CSV files natively (`header=true` and `label_column=name:quality`), it eliminates custom text encoders altogether.

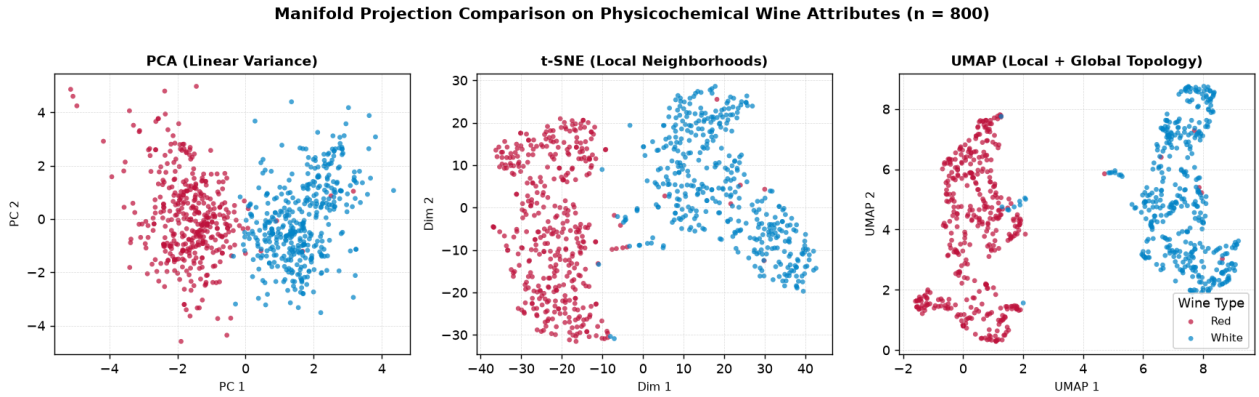


Figure 22: Comparison of bidimensional projections across PCA, t-SNE, and UMAP on wine attributes

9.6.2.1 Direct CLI Pre-Processing, Training, Prediction, and Feature Attribution

We drive `/usr/local/bin/lightgbm` directly from Nushell on the exact 3,500-row training and 1,398-row test split of `wine-white.csv` used earlier for Vowpal Wabbit. Note one essential LightGBM CLI requirement: while the training CSV contains both predictors and `quality`, the test CSV passed to `task=predict` must contain strictly the eleven predictor columns in matching order:

```
let white = open wine-white.csv

let train_w = $white | first 3500

let test_w = $white | skip 3500

let fcols = $white | columns | where $it != "quality"

$train_w
| select ...($fcols | append "quality")
| to csv
| save -f /tmp/lgb_train.csv

$test_w | select ...$fcols | to csv | save -f /tmp/lgb_test.csv

(
  lightgbm task=train objective=regression
    data=/tmp/lgb_train.csv header=true
    label_column=name:quality num_trees=100
    learning_rate=0.05 num_leaves=31 seed=42
    num_threads=4 force_col_wise=true
    output_model=/tmp/lgb_model.txt verbosity=-1
  out+err> /dev/null
)
```

```
(
  lightgbm task=predict data=/tmp/lgb_test.csv
    header=true input_model=/tmp/lgb_model.txt
    output_result=/tmp/lgb_preds.txt verbosity=-1
  out+err> /dev/null
)
```

Now we parse `/tmp/lgb_preds.txt` back into floats, pair them with `$test_w.quality` via `zip`, and verify the out-of-sample Mean Absolute Error directly in Nushell:

```
let p_raw = open --raw /tmp/lgb_preds.txt
  | lines
  | where ($it | str trim | is-not-empty)
  | each { into float }

let direct_lgb = $test_w.quality
  | each { into float }
  | zip $p_raw
  | each { |p|
      let obs = $p.0
      let pred = $p.1 | math round -p 4
      {
        observed: $obs
        predicted: $pred
        abs_error: (($obs - $pred) | math abs | math round -p 4)
      }
    }

$direct_lgb | first 4

$direct_lgb.abs_error | math avg | math round -p 4
```

To inspect which physicochemical variables drove those 100 leaf-wise trees, `/data/ch09/lgbm_importance.py` parses the saved LightGBM text booster (`/tmp/lgb_model.txt`) and emits a JSON table of split counts and squared-error gains. Read its code and invoke it directly before deleting our temporary files:

```
bat lgbm_importance.py
```

```
#!/usr/bin/env python
"""Extract split-count and gain importance from a LightGBM model."""
import json
import sys
import lightgbm as lgb
```

```
def main():
    if len(sys.argv) < 2:
        sys.stderr.write(
            "Usage: lgbm_importance.py <model_file.txt>\n"
        )
        sys.exit(1)

    model_file = sys.argv[1]
    booster = lgb.Booster(model_file=model_file)
    names = booster.feature_name()
    splits = booster.feature_importance(importance_type="split")
    gains = booster.feature_importance(importance_type="gain")
    tot = float(gains.sum())
    total_gain = tot if tot > 0 else 1.0

    rows = [
        {
            "feature": str(n),
            "splits": int(s),
            "gain": round(float(g), 4),
            "gain_pct": round(float(g) * 100.0 / total_gain, 2),
        }
        for n, s, g in zip(names, splits, gains)
    ]
    rows.sort(key=lambda r: r["gain"], reverse=True)
    print(json.dumps(rows))


if __name__ == "__main__":
    main()
```

Loading `lgb.Booster(model_file=...)` recovers exact feature names stored from the CSV header, queries both `"split"` frequencies and `"gain"` reductions, normalizes relative `gain_pct` to 100%, and writes sorted JSON records to standard output:

```
python lgbm_importance.py /tmp/lgb_model.txt | from json | first 5

rm -f /tmp/lgb_train.csv /tmp/lgb_test.csv

rm -f /tmp/lgb_model.txt /tmp/lgb_preds.txt
```

9.6.2.2 Encapsulating Training and Importance in `lgbm-regress`

In `dsin_tools.nu`, **`lgbm-regress`** bundles predictor column alignment, temporary directory lifecycle, `lightgbm` CLI invocation, and optional `--importance` extraction into one clean command. We inspect its implementation in the terminal via `view-code lgbm-regress`:

```
view-code lgbm-regress
```

```
# Train and evaluate LightGBM C++ CLI regression on tables.
export def lgbm-regress [
  train_tbl: table
  test_tbl: table
  --target(-t): string = "quality"
  --num-trees(-n): int = 100
  --learning-rate(-l): float = 0.05
  --num-leaves: int = 31
  --importance(-i)
]: nothing -> table {
  let tmp_dir = mktemp -d
  let train_csv = $"($tmp_dir)/train.csv"
  let test_csv = $"($tmp_dir)/test_features.csv"
  let model_file = $"($tmp_dir)/model.txt"
  let pred_file = $"($tmp_dir)/preds.txt"

  let feat_cols = $train_tbl | columns | where $it != $target
  let train_cols = $feat_cols | append $target
  $train_tbl | select ...$train_cols | to csv | save -f $train_csv
  $test_tbl | select ...$feat_cols | to csv | save -f $test_csv

  let lgbm_train_args = [
    "task=train"
    "objective=regression"
    $"data=($train_csv)"
    "header=true"
    $"label_column=name:($target)"
    $"num_trees=($num_trees)"
    $"learning_rate=($learning_rate)"
    $"num_leaves=($num_leaves)"
    "seed=42"
    "num_threads=4"
    "force_col_wise=true"
    $"output_model=($model_file)"
    "verbosity=-1"
  ]
  lightgbm ...$lgbm_train_args out+err> /dev/null

  if $importance {
```

```
    let bin_p = "/usr/local/bin/lgbm-importance"
    let py_imp = if ($bin_p | path exists) {
      $bin_p
    } else if ("/data/ch09/lgbm_importance.py" | path exists) {
      "/data/ch09/lgbm_importance.py"
    } else {
      "lgbm_importance.py"
    }
    let imp_tbl = python $py_imp $model_file | from json
    rm -rf $tmp_dir
    return $imp_tbl
  }

  let lgbm_pred_args = [
    "task=predict"
    $"data=($test_csv)"
    "header=true"
    $"input_model=($model_file)"
    $"output_result=($pred_file)"
    "verbosity=-1"
  ]
  lightgbm ...$lgbm_pred_args out+err> /dev/null

  let preds = open --raw $pred_file
    | lines
    | where ($it | str trim | is-not-empty)
    | each { into float }
  let actuals = $test_tbl | get $target | each { into float }
  rm -rf $tmp_dir

  $actuals | zip $preds | each { |pair|
    let obs = $pair.0
    let pred = $pair.1 | math round -p 4
    let signed_err = ($pred - $obs) | math round -p 4
    let abs_err = ($obs - $pred) | math abs | math round -p 4
    {
      actual: $obs
      observed: $obs
      predicted: $pred
      error: $signed_err
      abs_error: $abs_err
    }
  }
}
```

9.6.2.3 Comparative Rerun and Visual Feature-Gain Diagnostics

Rerunning our 3,500/1,398 benchmark through `lgbm-regress` reproduces the `0.5429` MAE (improving upon Vowpal Wabbit's linear `0.5693` MAE) and emits the full importance ranking:

```
let white = open wine-white.csv

let train_w = $white | first 3500

let test_w = $white | skip 3500

let preds_lgbm = lgbm-regress $train_w $test_w --target quality

$preds_lgbm | get abs_error | math avg | math round -p 4

let imp = lgbm-regress $train_w $test_w --target quality --importance

$imp

$imp | to json | save -f /tmp/lgbm_imp.json
```

Alcohol concentration drives `33.54%` of total squared-error reduction, followed by `volatile_acidity` (`12.17%`), `free_sulfur_dioxide` (`11.07%`), and `residual_sugar` (`7.85%`). Notice also the divergence between gain share and raw split counts: a variable such as `alcohol` achieves massive variance reduction on early root splits, whereas secondary predictors accumulate many fine leaf adjustments with smaller marginal gain. The diagnostic script `plot_lgbm_importance.py` renders both perspectives side by side:

```
bat plot_lgbm_importance.py
```

```
#!/opt/modern-ml/bin/python
"""Render LightGBM feature gain and split counts."""
import os
import sys
import warnings

os.environ.setdefault("MPLCONFIGDIR", "/tmp/matplotlib_cache")
warnings.filterwarnings("ignore")

import lightgbm as lgb
import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import pandas as pd


def load_importance(src):
```

```
    if src.endswith(".json"):
        return pd.read_json(src)
    if src.endswith(".txt"):
        booster = lgb.Booster(model_file=src)
    else:
        df = pd.read_csv(src).iloc[:3500]
        drop_c = [
            c for c in ("type", "quality") if c in df.columns
        ]
        X = df.drop(columns=drop_c)
        y = df["quality"]
        dtrain = lgb.Dataset(X, label=y)
        params = {
            "objective": "regression",
            "learning_rate": 0.05,
            "num_leaves": 31,
            "seed": 42,
            "num_threads": 4,
            "force_col_wise": True,
            "verbosity": -1,
        }
        booster = lgb.train(params, dtrain, num_boost_round=100)

    gains = booster.feature_importance(importance_type="gain")
    splits = booster.feature_importance(importance_type="split")
    tot = float(gains.sum()) if float(gains.sum()) > 0 else 1.0
    return pd.DataFrame({
        "feature": booster.feature_name(),
        "splits": splits,
        "gain_pct": gains * 100.0 / tot,
    })


def main():
    src = sys.argv[1] if len(sys.argv) > 1 else "wine-white.csv"
    png_out = (
        sys.argv[2] if len(sys.argv) > 2 else "lgbm_importance.png"
    )
    df = load_importance(src).sort_values(
        "gain_pct", ascending=True
    )

    fig, (ax1, ax2) = plt.subplots(
        1, 2, figsize=(11.0, 4.4), dpi=130
    )
```

```
    bars1 = ax1.barh(
        df["feature"], df["gain_pct"], color="#0284c7"
    )
    ax1.set_title("Relative Split Gain (%)", fontweight="bold")
    ax1.set_xlabel("Share of Squared-Error Reduction (%)")
    ax1.set_xlim(0, float(df["gain_pct"].max()) * 1.22)
    ax1.grid(True, axis="x", ls="--", lw=0.4, alpha=0.5)
    for b in bars1:
        w = b.get_width()
        ax1.text(
            w + 0.5, b.get_y() + b.get_height() / 2,
            f"{w:.1f}%", va="center", fontsize=8
        )

    ax2.barh(df["feature"], df["splits"], color="#0f766e")
    ax2.set_title("Interior Split Frequency", fontweight="bold")
    ax2.set_xlabel("Total Splits Across 100 Trees")
    ax2.set_yticks([])
    ax2.grid(True, axis="x", ls="--", lw=0.4, alpha=0.5)

    fig.suptitle(
        "LightGBM Feature Importance Diagnostics (wine-white.csv)",
        fontsize=11.5, fontweight="bold", y=1.01
    )
    fig.tight_layout()
    fig.savefig(png_out, bbox_inches="tight", dpi=130)
    top_f = df.iloc[-1]["feature"]
    top_g = df.iloc[-1]["gain_pct"]
    print(f"Saved {png_out} (top={top_f}: {top_g:.2f}%)")


if __name__ == "__main__":
    main()
```

Three components structure `plot_lgbm_importance.py`: `load_importance(src)` accepts pre-computed JSON tables, `.txt` model files, or raw CSV datasets interchangeably; `ax1` plots annotated percentage gain bars sorted from lowest to highest contributor; and `ax2` aligns raw interior split counts along the shared vertical feature axis. We render `lgbm_importance.png` and preview it in the terminal via `timg`:

```
python plot_lgbm_importance.py /tmp/lgbm_imp.json

rm -f /tmp/lgbm_imp.json

timg -g 80x25 lgbm_importance.png
```

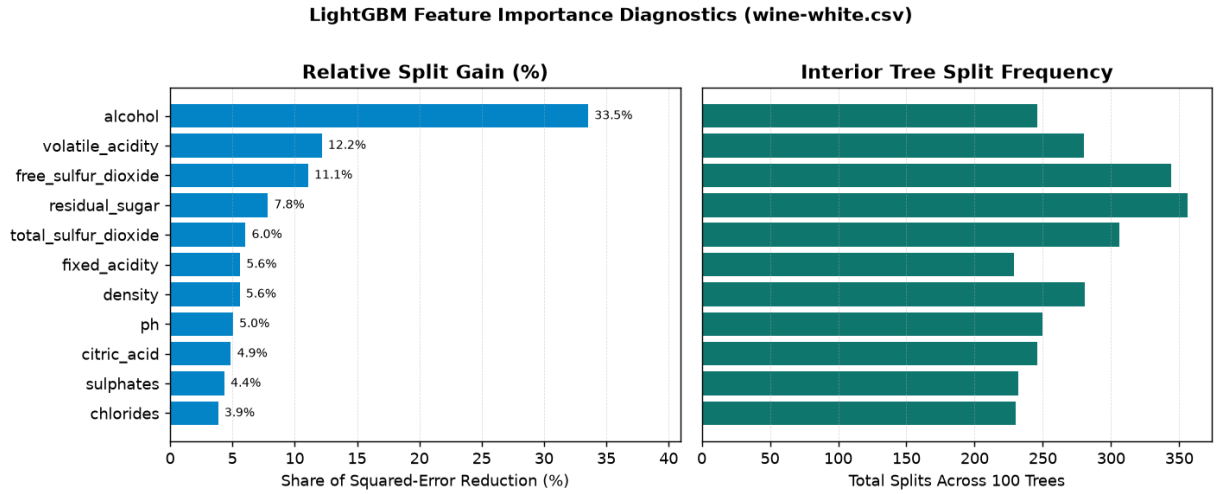


Figure 23: LightGBM feature split-gain percentage and split-count diagnostics on white wine predictors

9.6.3 Vectorized In-Process Analytical Regression with DuckDB

Before fitting multi-parameter ensembles, sound statistical modeling begins with fast marginal linear screening across candidate predictors. **DuckDB** (Raasveldt & Mühleisen, 2019) embeds a vectorized, columnar analytical SQL engine inside a single portable binary (`/usr/local/bin/duckdb`) with native JSON emission (`duckdb -json`). Its SQL:2003 statistical aggregates (`regr_slope`, `regr_intercept`, `corr`, `regr_r2`, and `regr_count`) evaluate closed-form Ordinary Least Squares moments in single-pass SIMD loops in under 200 milliseconds.

9.6.3.1 Direct CLI SQL Querying and Nushell JSON Ingestion

To see how DuckDB operates without any wrapper, construct a direct SQL query over `'wine-white.csv'` evaluating the marginal linear regression of `quality` on `alcohol`, execute `duckdb -json -c`, and pipe its JSON output straight into `from json`:

```
let q_alc = [
  "SELECT 'alcohol' AS predictor,"
  "ROUND(regr_slope(quality, alcohol), 4) AS slope,"
  "ROUND(regr_intercept(quality, alcohol), 4) AS intercept,"
  "ROUND(corr(quality, alcohol), 4) AS pearson_r,"
  "ROUND(regr_r2(quality, alcohol), 4) AS r_squared,"
  "CAST(regr_count(quality, alcohol) AS BIGINT) AS n"
  "FROM 'wine-white.csv';"
] | str join " "

duckdb -json -c $q_alc | from json
```

Comparing two or three predictors directly in SQL simply chains `UNION ALL` branches ordered by

`r_squared DESC`, yet writing eleven `UNION ALL` clauses by hand whenever a dataset schema changes quickly grows tedious.

9.6.3.2 Encapsulating Schema-Driven SQL Generation in `duckdb-ols`

In `dsin_tools.nu`, **`duckdb-ols`** inspects the incoming Nushell table schema at runtime, identifies all numeric columns (`int` or `float`) distinct from `--target`, writes a temporary CSV snapshot, synthesizes the `UNION ALL` query string automatically, and parses `duckdb -json` back into a sorted Nushell table. We inspect its implementation in the terminal via `view-code duckdb-ols`:

```
view-code duckdb-ols
```

```
# Compute vectorized OLS regression diagnostics via DuckDB CLI.
export def duckdb-ols [
  --target(-y): string = "quality"
  --predictor(-x): string = ""
]: table -> table {
  let tbl = $in
  let first_row = $tbl | first
  let candidate_cols = if ($predictor | is-not-empty) {
    [$predictor]
  } else {
    $tbl | columns | where { |c|
      let t = $first_row | get $c | describe
      ($c != $target) and (($t == "int") or ($t == "float"))
    }
  }

  let tmp_csv = mktemp --suffix ".csv"
  let keep = $candidate_cols | append $target
  $tbl | select ...$keep | to csv | save -f $tmp_csv

  let unions = $candidate_cols | each { |col|
    let q_y = $"\"($target)\""
    let q_x = $"\"($col)\""
    [
      $"SELECT '($col)' AS predictor, "
      $"ROUND\(regr_slope\(($q_y), ($q_x)\), 4\) AS slope, "
      $"ROUND\(regr_intercept\(($q_y), ($q_x)\), 4\) AS intercept, "
      $"ROUND\(corr\(($q_y), ($q_x)\), 4\) AS pearson_r, "
      $"ROUND\(regr_r2\(($q_y), ($q_x)\), 4\) AS r_squared, "
      $"CAST\(regr_count\(($q_y), ($q_x)\) AS BIGINT\) AS n "
      "FROM tbl"
    ] | str join ""
  } | str join " UNION ALL "
```

```
  let sql = [
    "CREATE TEMP TABLE tbl AS "
    $"SELECT * FROM read_csv_auto\('($tmp_csv)'\); "
    $"SELECT * FROM \(($unions)\) ORDER BY r_squared DESC;"
  ] | str join ""

  let res = duckdb -json -c $sql | from json
  rm -f $tmp_csv
  $res
}
```

9.6.3.3 Comparative Rerun and Multivariate Residual Diagnostics

Screening all eleven physicochemical predictors on `wine-white.csv` (and verifying our single-predictor `alcohol` check) now takes one pipeline:

```
open wine-white.csv | duckdb-ols --target quality

open wine-white.csv
| duckdb-ols --target quality --predictor alcohol
```

Marginally, `alcohol` exhibits the strongest linear association with perceived quality ($\hat{\beta}_1 = 0.3135$, $r = 0.4356$, $R^2 = 0.1897$), followed by `density` ($r = -0.3071$, $R^2 = 0.0943$) and `chlorides` ($r = -0.2099$, $R^2 = 0.0441$). By contrast, `free_sulfur_dioxide` and `residual_sugar` show near-zero marginal R^2 under linear OLS yet rank third and fourth in LightGBM split gain, confirming that their influence operates through non-linear threshold interactions (insufficient sulfur dioxide permits oxidation, whereas excess concentrations produce pungent sulfur aromas).

Even when all eleven predictors enter a joint multivariate OLS regression, linear specification captures only $R^2 = 0.2819$ of sensory variance. To inspect the residual geometry directly, study `plot_wine_residuals.py`:

```
bat plot_wine_residuals.py
```

```
#!/opt/modern-ml/bin/python
"""Render OLS residual diagnostics on wine-white.csv."""
import os
import sys
import warnings

os.environ.setdefault("MPLCONFIGDIR", "/tmp/matplotlib_cache")
warnings.filterwarnings("ignore")

import matplotlib
matplotlib.use("Agg")
```

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score


def main():
    csv_in = sys.argv[1] if len(sys.argv) > 1 else "wine-white.csv"
    png_out = (
        sys.argv[2] if len(sys.argv) > 2 else "wine_residuals.png"
    )

    df = pd.read_csv(csv_in)
    drop_c = [c for c in ("type", "quality") if c in df.columns]
    X = df.drop(columns=drop_c).select_dtypes(
        include=[np.number]
    )
    y = df["quality"].to_numpy(dtype=float)

    ols = LinearRegression().fit(X, y)
    y_hat = ols.predict(X)
    resid = y - y_hat
    r2 = r2_score(y, y_hat)
    sigma = float(np.std(resid, ddof=X.shape[1] + 1))

    fig, (ax1, ax2) = plt.subplots(
        1, 2, figsize=(10.5, 4.2), dpi=130
    )

    ax1.scatter(
        y_hat, resid, s=10, alpha=0.28,
        color="#0284c7", edgecolors="none"
    )
    ax1.axhline(0.0, color="#be123c", ls="--", lw=1.4)
    ax1.set_title("Residuals vs. Fitted Values", fontweight="bold")
    ax1.set_xlabel("Fitted Sensory Quality")
    ax1.set_ylabel("Residual (Observed - Fitted)")
    ax1.grid(True, ls="--", lw=0.4, alpha=0.5)

    ax2.hist(
        resid, bins=32, density=True,
        color="#0f766e", alpha=0.72, edgecolor="white"
    )
    grid_x = np.linspace(resid.min(), resid.max(), 200)
    gauss = np.exp(-0.5 * (grid_x / sigma) ** 2) / (
```

```
        sigma * np.sqrt(2.0 * np.pi)
    )
    ax2.plot(
        grid_x, gauss, color="#be123c", lw=1.8,
        label=f"Normal (sd={sigma:.2f})"
    )
    ax2.set_title("Residual Error Distribution", fontweight="bold")
    ax2.set_xlabel("Residual")
    ax2.set_ylabel("Density")
    ax2.legend(loc="best", frameon=True)
    ax2.grid(True, ls="--", lw=0.4, alpha=0.5)

    fig.suptitle(
        f"Multivariate OLS Diagnostics ({csv_in}, R2 = {r2:.3f})",
        fontsize=11.5, fontweight="bold", y=1.01
    )
    fig.tight_layout()
    fig.savefig(png_out, bbox_inches="tight", dpi=130)
    print(f"Saved {png_out} (R2={r2:.4f}, sd={sigma:.4f})")


if __name__ == "__main__":
    main()
```

Three statistical diagnostics stand out in `plot_wine_residuals.py`:

1. **Degrees-of-freedom adjusted scale (`sigma`):** Computing `np.std(resid, ddof=X.shape[1] + 1)` divides the residual sum of squares by $n - p - 1$ rather than n, yielding the unbiased OLS root-mean-square error ($\hat{\sigma} = 0.7514$).
2. **Discrete response banding (`ax1`):** Plotting $e_i = y_i - \hat{y}_i$ against $\hat{y}_i$ exposes parallel diagonal bands (`y_i = 3, 4, ..., 9`) along with mild compression at extreme quality scores where linear predictions regress toward the mean.
3. **Parametric Gaussian reference (`ax2`):** Overlaying the exact $\mathcal{N}(0, \hat{\sigma}^2)$ density curve onto the normalized residual histogram reveals a slightly heavy right tail among exceptional vintages.

Generate `wine_residuals.png` and inspect both panels directly in the terminal:

```
python plot_wine_residuals.py

timg -g 80x25 wine_residuals.png
```

9.6.4 Streaming Multi-Model Classification and Neural Networks

Earlier in this chapter, benchmarking Logistic Regression against a Random Forest via SKLL (`run_experiment`) required creating `train/` and `test/` directories, writing an INI-style `classify.cfg`

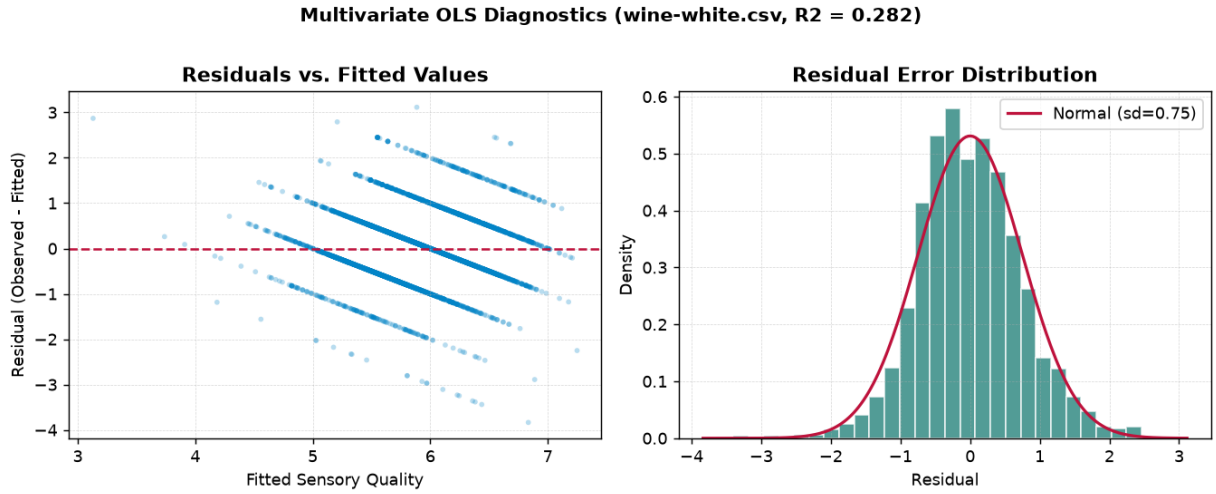


Figure 24: Multivariate OLS residual diagnostics and Gaussian density overlay on white wine sensory quality

specification, and projecting tab-separated summary files from `output/`. Modern pipeline design eliminates that file-system ceremony completely by streaming CSV in and JSON records out.

9.6.4.1 Inspecting and Invoking `ml_benchmark.py` Directly

The benchmark engine lives in the standalone script `/data/ch09/ml_benchmark.py`. It evaluates five distinct architectural families under identical stratified partitions:

1. **Logistic Regression (`logistic`):** L_2-penalized linear log-odds baseline.
2. **Linear Support Vector Classifier (`linear_svc`):** Maximum-margin hyperplane under primal squared hinge loss (`dual=False`).
3. **Multi-Layer Perceptron (`mlp`):** Feedforward neural network (`MLPClassifier`) with two ReLU hidden layers of $(64, 32)$ units, Adam optimization, and validation-based early stopping.
4. **Random Forest (`rf`):** Bagged ensemble of 100 decorrelated CART trees.
5. **LightGBM (`lgbm`):** Histogram-accelerated leaf-wise gradient boosted decision trees (`num_leaves=31`).

We inspect `ml_benchmark.py` via `bat`:

```
bat ml_benchmark.py
```

```
#!/usr/bin/env python
"""Multi-model classification benchmark for Nushell pipelines."""
import argparse
import json
import os
import sys
```

```
import time
import warnings

os.environ.setdefault("OMP_NUM_THREADS", "4")
os.environ.setdefault("PYTHONWARNINGS", "ignore")
warnings.filterwarnings("ignore")

import lightgbm as lgb
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.ensemble import RandomForestClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    f1_score,
    precision_score,
    recall_score,
)
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import (
    LabelEncoder,
    OneHotEncoder,
    StandardScaler,
)
from sklearn.svm import LinearSVC


def build_registry(prep_scaled, prep_tree, seed):
    return {
        "logistic": Pipeline([
            ("prep", prep_scaled),
            ("clf", LogisticRegression(
                max_iter=1000, random_state=seed
            )),
        ]),
        "rf": Pipeline([
            ("prep", prep_tree),
            ("clf", RandomForestClassifier(
                n_estimators=100, random_state=seed, n_jobs=4
            )),
        ]),
        "lgbm": Pipeline([
            ("prep", prep_tree),
            ("clf", lgb.LGBMClassifier(
```

```
                n_estimators=100, learning_rate=0.05,
                num_leaves=31, random_state=seed,
                verbosity=-1, n_jobs=4,
            )),
        ]),
        "linear_svc": Pipeline([
            ("prep", prep_scaled),
            ("clf", LinearSVC(
                dual=False, max_iter=2000, random_state=seed
            )),
        ]),
        "mlp": Pipeline([
            ("prep", prep_scaled),
            ("clf", MLPClassifier(
                hidden_layer_sizes=(64, 32), max_iter=300,
                early_stopping=True, n_iter_no_change=10,
                random_state=seed,
            )),
        ]),
    }


def main():
    p = argparse.ArgumentParser(
        description="Multi-model classification benchmark."
    )
    p.add_argument("--input", "-i", type=str, default="-")
    p.add_argument("--target", "-t", type=str, default="type")
    p.add_argument(
        "--models", "-m", type=str,
        default="logistic,rf,lgbm,linear_svc,mlp",
    )
    p.add_argument("--test-size", type=float, default=0.25)
    p.add_argument("--seed", "-s", type=int, default=42)
    args = p.parse_args()

    src = sys.stdin if args.input == "-" else args.input
    df = pd.read_csv(src)
    if args.target not in df.columns:
        msg = f"Error: Target '{args.target}' not found.\n"
        sys.stderr.write(msg)
        sys.exit(1)

    X = df.drop(columns=[args.target])
    y = LabelEncoder().fit_transform(df[args.target].astype(str))
    num_c = [
```

```
    c for c in X.columns if pd.api.types.is_numeric_dtype(X[c])
]
cat_c = [c for c in X.columns if c not in num_c]

ohe_s = OneHotEncoder(handle_unknown="ignore")
ohe_d = OneHotEncoder(
    handle_unknown="ignore", sparse_output=False
)
prep_scaled = ColumnTransformer(
    [("num", StandardScaler(), num_c), ("cat", ohe_s, cat_c)],
    remainder="drop",
)
prep_tree = ColumnTransformer(
    [("num", "passthrough", num_c), ("cat", ohe_d, cat_c)],
    remainder="drop",
)

X_tr, X_te, y_tr, y_te = train_test_split(
    X, y, test_size=args.test_size,
    random_state=args.seed, stratify=y,
)
parts = args.models.split(",")
req = [m.strip().lower() for m in parts if m.strip()]
reg = build_registry(prep_scaled, prep_tree, args.seed)

rows = []
for name in req:
    if name not in reg:
        continue
    pipe = reg[name]
    t0 = time.perf_counter()
    pipe.fit(X_tr, y_tr)
    fit_ms = round((time.perf_counter() - t0) * 1000.0, 2)
    preds = pipe.predict(X_te)
    kw = {"average": "weighted", "zero_division": 0}
    rows.append({
        "model": name,
        "accuracy": round(float(accuracy_score(y_te, preds)), 4),
        "precision": round(
            float(precision_score(y_te, preds, **kw)), 4
        ),
        "recall": round(
            float(recall_score(y_te, preds, **kw)), 4
        ),
        "f1_score": round(float(f1_score(y_te, preds, **kw)), 4),
        "train_time_ms": fit_ms,
```

```
        })

    rows.sort(key=lambda r: r["f1_score"], reverse=True)
    print(json.dumps(rows))


if __name__ == "__main__":
    main()
```

Four engineering choices make `ml_benchmark.py` production-grade:

1. **Architecture-specific `ColumnTransformer` pipelines:** Gradient and margin models (`logistic`, `linear_svc`, `mlp`) receive standardized numeric columns (`prep_scaled`), whereas scale-invariant tree ensembles (`rf`, `lgbm`) bypass scaling (`prep_tree`) while sharing one-hot encoding for any categorical columns.
2. **Stratified holdout evaluation (`stratify=y`):** Preserving exact class proportions across `X_tr` and `X_te` prevents minority-class variance distortion on imbalanced datasets such as `banco.csv`.
3. **Bounded thread parallelism (`OMP_NUM_THREADS=4`, `n_jobs=4`):** Capping OpenMP and worker threads prevents kernel futex contention inside containerized environments while measuring wall-clock training latency (`time.perf_counter()`).
4. **Sorted JSON array emission:** Printing `json.dumps(rows)` sorted by weighted F_1 lets Nushell ingest typed floats and millisecond timings via `from json` without losing numeric precision.

Invoke `ml_benchmark.py` directly on `wine-both.csv` over standard input before introducing its wrapper:

```
open wine-both.csv
| to csv
| python ml_benchmark.py --target type
| from json
```

9.6.4.2 Encapsulating Multi-Model Benchmarking in `ml-benchmark`

In `dsin_tools.nu`, **`ml-benchmark`** converts Nushell list flags (`--models`) into comma-separated strings, manages temporary CSV handoff, resolves `/usr/local/bin/ml-benchmark-py`, and parses the resulting JSON leaderboard directly into a Nushell table. We inspect its implementation in the terminal via `view-code ml-benchmark`:

```
view-code ml-benchmark
```

```
# Benchmark classification architectures on streaming tables.
export def ml-benchmark [
  --target(-t): string = "type"
```

```
  --models(-m): list<string> = [
    "logistic" "rf" "lgbm" "linear_svc" "mlp"
  ]
  --test-size: float = 0.25
  --seed(-s): int = 42
]: table -> table {
  let tbl = $in
  let models_csv = $models | str join ","
  let tmp_in = mktemp --suffix ".csv"
  $tbl | to csv | save -f $tmp_in

  let bin_p = "/usr/local/bin/ml-benchmark-py"
  let py_bench = if ($bin_p | path exists) {
    $bin_p
  } else if ("/data/ch09/ml_benchmark.py" | path exists) {
    "/data/ch09/ml_benchmark.py"
  } else {
    "ml_benchmark.py"
  }

  let bench_args = [
    "--input" $tmp_in
    "--target" $target
    "--models" $models_csv
    "--test-size" ($test_size | into string)
    "--seed" ($seed | into string)
  ]
  let res = python $py_bench ...$bench_args | from json
  rm -f $tmp_in
  $res
}
```

9.6.4.3 Comparative Rerun and Visual Accuracy-Latency Trade-Offs

Evaluating all five classifiers on `wine-both.csv` and on the retail banking campaign dataset `banco.csv` (after dropping non-predictive identifiers `ID` and `ZIP_cod`) now reduces to concise one-line pipelines:

```
open wine-both.csv | ml-benchmark --target type

let banco_bench = open banco.csv
  | reject ID ZIP_cod
  | ml-benchmark --target prestamo

$banco_bench
```

```
$banco_bench | to json | save -f /tmp/banco_bench.json
```

On `banco.csv`, where loan acceptance depends on sharp income, education, and family-size thresholds, linear classifiers plateau near $F_1 \approx 0.95$ (`logistic` at `0.9484` and `linear_svc` at `0.9511`) in roughly 10 milliseconds. Introducing non-linear hidden representations via the Multi-Layer Perceptron (`mlp`) lifts weighted F_1 to `0.9797` in ~250 milliseconds, while axis-aligned tree ensembles (`rf` at `0.9895` and `lgbm` at `0.9903`) achieve the highest discrimination, matching empirical findings on medium-sized tabular benchmarks (Grinsztajn et al., 2022).

To visualize both predictive discrimination and computational cost together, examine `plot_benchmark_comparison.py`:

```
bat plot_benchmark_comparison.py
```

```
#!/opt/modern-ml/bin/python
"""Render multi-model F1, accuracy, and training latency charts."""
import json
import os
import subprocess
import sys
import tempfile
import warnings

os.environ.setdefault("OMP_NUM_THREADS", "4")
os.environ.setdefault("MPLCONFIGDIR", "/tmp/matplotlib_cache")
warnings.filterwarnings("ignore")

import matplotlib
matplotlib.use("Agg")
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


def load_benchmark(src):
    if src.endswith(".json") and os.path.exists(src):
        return pd.read_json(src)

    df_raw = pd.read_csv(src)
    drop_c = [c for c in ("ID", "ZIP_cod") if c in df_raw.columns]
    df_clean = df_raw.drop(columns=drop_c)
    tgt = "prestamo" if "prestamo" in df_clean.columns else "type"

    with tempfile.NamedTemporaryFile(
        mode="w", suffix=".csv", delete=False
```

```
    ) as tmp:
        df_clean.to_csv(tmp.name, index=False)
        tmp_path = tmp.name

    script_dir = os.path.dirname(os.path.abspath(__file__))
    bench_py = os.path.join(script_dir, "ml_benchmark.py")
    cmd = [
        sys.executable, bench_py,
        "--input", tmp_path,
        "--target", tgt,
        "--models", "logistic,linear_svc,mlp,rf,lgbm",
    ]
    raw_out = subprocess.check_output(cmd, text=True)
    os.remove(tmp_path)
    return pd.DataFrame(json.loads(raw_out))


def main():
    src = sys.argv[1] if len(sys.argv) > 1 else "banco.csv"
    png_out = (
        sys.argv[2] if len(sys.argv) > 2
        else "benchmark_comparison.png"
    )
    df = load_benchmark(src).sort_values(
        "f1_score", ascending=True
    )

    fig, (ax1, ax2) = plt.subplots(
        1, 2, figsize=(11.2, 4.3), dpi=130
    )
    x = np.arange(len(df))
    w = 0.36

    b1 = ax1.bar(
        x - w / 2, df["f1_score"], w,
        label="Weighted F1", color="#0284c7"
    )
    ax1.bar(
        x + w / 2, df["accuracy"], w,
        label="Accuracy", color="#0f766e"
    )
    min_sc = float(df[["f1_score", "accuracy"]].min().min())
    ax1.set_ylim(max(0.0, min_sc - 0.04), 1.015)
    ax1.set_xticks(x)
    ax1.set_xticklabels(df["model"])
    ax1.set_title("Classification Metrics", fontweight="bold")
```

```
    ax1.set_ylabel("Score")
    ax1.legend(loc="lower right", frameon=True)
    ax1.grid(True, axis="y", ls="--", lw=0.4, alpha=0.5)
    for b in b1:
        h = b.get_height()
        ax1.text(
            b.get_x() + b.get_width() / 2, h + 0.002,
            f"{h:.3f}", ha="center", va="bottom", fontsize=7.5
        )

    bars_t = ax2.barh(
        df["model"], df["train_time_ms"], color="#be123c"
    )
    ax2.set_title("Training Latency (ms)", fontweight="bold")
    ax2.set_xlabel("Wall-Clock Time (ms)")
    max_ms = float(df["train_time_ms"].max())
    ax2.set_xlim(0, max_ms * 1.25)
    ax2.grid(True, axis="x", ls="--", lw=0.4, alpha=0.5)
    for b in bars_t:
        wt = b.get_width()
        ax2.text(
            wt + max_ms * 0.02,
            b.get_y() + b.get_height() / 2,
            f"{wt:.0f} ms", va="center", fontsize=8
        )

    fig.suptitle(
        "Multi-Model Tabular Classification Benchmark",
        fontsize=11.5, fontweight="bold", y=1.01
    )
    fig.tight_layout()
    fig.savefig(png_out, bbox_inches="tight", dpi=130)
    best = df.iloc[-1]
    print(
        f"Saved {png_out} (best={best['model']}: "
        f"F1={best['f1_score']:.4f})"
    )


if __name__ == "__main__":
    main()
```

Accepting either a pre-saved JSON leaderboard (`/tmp/banco_bench.json`) or a raw CSV dataset (`banco.csv`), `plot_benchmark_comparison.py` zooms the left vertical axis dynamically around the observed `[min_sc - 0.04, 1.015]` range so subtle percentage-point gains between linear, neural,

and boosted-tree learners remain legible alongside training latency in milliseconds:

```
python plot_benchmark_comparison.py /tmp/banco_bench.json

rm -f /tmp/banco_bench.json

timg -g 80x25 benchmark_comparison.png
```

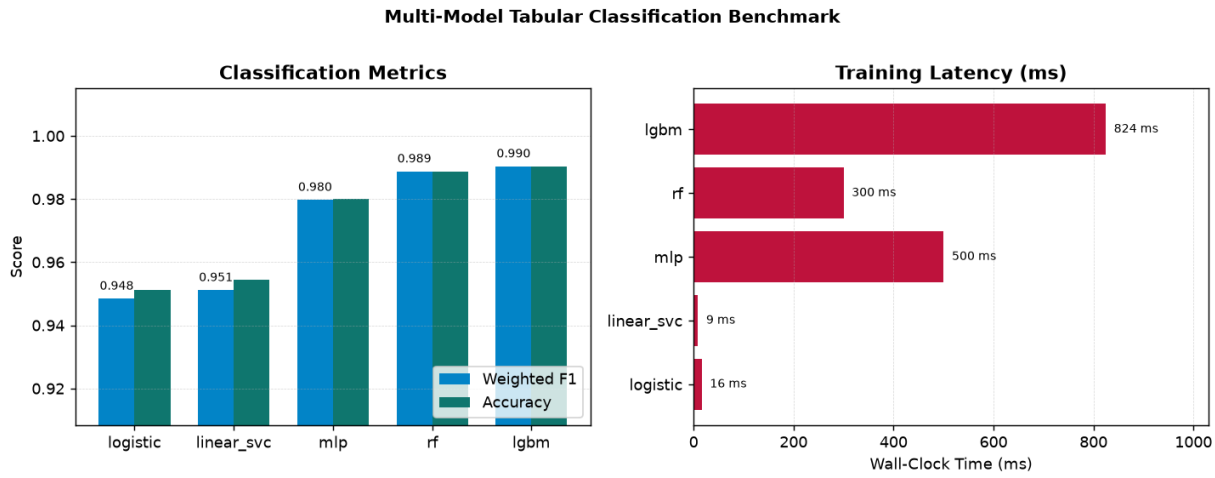


Figure 25: Comparative weighted F1, accuracy, and training latency across five classification architectures on banking data

9.6.5 The Pure-Rust Machine Learning Frontier

Because Nushell itself and `nu_plugin_polars` are written in Rust on top of Apache Arrow columnar memory, executing machine learning models directly in compiled Rust binaries offers strict compile-time type safety, effortless Rayon CPU parallelism, and memory efficiency without interpreter startup costs. Five active open-source projects represent this modern frontier and are pre-packaged as standalone command-line tools in the `dsin` container:

1. **`smartcore`** (`smartcore-cli`): Comprehensive statistical machine learning in pure Rust (SmartCore Development Team, 2024) providing random forests, regularized linear models, and SVMs with zero data races.
2. **`linfa`** (`linfa-cli`): Modular pure-Rust classical machine learning toolkit (Linfa Contributors, 2024) implementing *K-Means*, *DBSCAN*, and Principal Component Analysis (*PCA*).
3. **`candle`** (`candle-cli`): Minimalist tensor computing and neural network framework from Hugging Face (Hugging Face, 2024) designed for lightweight, high-performance model evaluation without CUDA or Python runtimes.
4. **`burn`** (`burn-cli`): Flexible deep learning framework in Rust (Burn Contributors, 2024) providing autograd tensors and multi-layer perceptron (`MLP`) training directly on CPU via the `ndarray` backend.

5. **nutorch** (nutorch): High-performance tensor utility engineered to ingest tabular Nushell streams (AstroHacker Labs, 2024), compute linear algebraic transformations, and evaluate geometric matrix norms.

Below, we examine complete, reproducible workflows for all five engines on our wine quality benchmark:

9.6.5.1 Supervised Classification with Random Forests

We fit a 100-tree random forest ensemble on physicochemical attributes to classify wine variety (`red` versus `white`), evaluating class balance and agreement directly in Nushell:

```
open wine-both.csv
| first 300
| append (open wine-both.csv | last 300)
| to csv
| smartcore-cli fit-predict --target type --trees 100
| from csv
| polars into-df
| polars group-by actual predicted
| polars agg [
    (polars col actual | polars count | polars as frequency)
  ]
| polars collect
| polars into-nu
```

The pipeline induces decision trees across CPU cores in memory, producing a confusion matrix directly in tabular format.

9.6.5.2 Unsupervised Clustering and Dimensionality Reduction

To explore geometric clusters without class labels, we execute *K-Means* clustering into three groups:

```
open wine-both.csv
| first 200
| to csv
| linfa-cli kmeans --k 3 --exclude type
| from csv
| select cluster fixed_acidity alcohol quality
| first 5
```

Similarly, we project the multivariate feature space onto two principal components via *PCA*:

```
open wine-both.csv
| first 200
| to csv
| linfa-cli pca --components 2 --exclude type
```

```
| from csv
| select pc_1 pc_2 type quality
| first 5
```

Both tools consume tabular CSV streams and emit structured tables preserving sample row identity.

9.6.5.3 Tensor Inference and Neural Networks

Hugging Face `candle` builds in-memory tensors and evaluates the forward pass of a multi-layer perceptron with hidden activations without starting a Python interpreter:

```
open wine-both.csv
| first 100
| to csv
| candle-cli forward-mlp --hidden 16 --classes 2 --exclude type
| from csv
| select logit_0 logit_1 quality type
| first 5
```

The resulting `logit_0` and `logit_1` columns represent computed class activation values for each vintage.

9.6.5.4 Deep Learning Neural Regression on CPU

Using `burn`, we iteratively train a multi-layer perceptron to predict sensory quality scores via gradient descent:

```
open wine-both.csv
| first 200
| to csv
| burn-cli train-mlp --target quality
| from csv
| select actual predicted quality alcohol
| first 5
```

The resulting output pairs the taster's ground-truth score with the neural network prediction.

9.6.5.5 In-Memory Tensor Linear Algebra

Finally, `nutorch` performs in-memory tensor transformations on tabular attributes, scaling columns and verifying tensor dimensions:

```
open wine-both.csv
| select fixed_acidity volatile_acidity citric_acid alcohol
| first 5
```

```
| to csv
| nutorch matmul --eye 4
| from csv
```

To quantify matrix magnitude and dispersion across rows, we compute tensor summary statistics:

```
open wine-both.csv
| select fixed_acidity volatile_acidity citric_acid alcohol
| first 100
| to csv
| nutorch stats
| from csv
```

While Python (`scikit-learn`, `umap-learn`) and C++ (`lightgbm`, `duckdb`) remain the primary production tools for tabular workflows today, mastering pure-Rust CLI utilities prepares data scientists for zero-copy Arrow memory workflows as the scientific Rust ecosystem continues to expand.

9.7 Chapter Summary

Throughout this chapter, we built unsupervised and supervised modeling workflows from the ground up inside Nushell, progressing deliberately from raw command-line execution and auxiliary script inspection to reusable module wrappers:

1. **Dimensionality Reduction and Collinearity Control:** Projected physicochemical wine attributes into low-dimensional subspaces by invoking **Tapkee** directly on headerless numeric matrices before encapsulating the workflow in `tapkee-reduce`, cross-checking coordinates against R (`prcomp`) and Python (`PCA`), and stabilizing collinear biological regressions through VIF diagnostics and Principal Component Regression (PCR).
2. **Unsupervised Clustering and Geometric Validation:** Partitioned unlabeled observations with standalone **K-Means** scripts (`kmeans_cluster.py` and `kmeans-cluster`), agglomerative linkages (`hclust`), density-based spatial scanning (**DBSCAN**), and outlier-resistant medoids (**PAM**), selecting partition counts via silhouette and inertia curves (`kmeans_elbow.png`).
3. **Out-of-Core and Penalized Regression:** Constructed explicit Vowpal Wabbit feature strings in Nushell, trained streaming **Vowpal Wabbit** models directly via `vw` prior to introducing `to-vw` and `vw-regress`, and compared predictive error against classical Ordinary Least Squares (`lm`), Ridge regression, and Random Forest ensembles.
4. **Declarative and Polyglot Classification:** Generated train/test splits and configuration manifests for **SKLL** (`run_experiment` and `skll-metrics`), then evaluated standalone R and Python scripts spanning binomial GLMs, recursive partitioning trees, distance-weighted k-NN, Gaussian Naïve Bayes, radial SVMs, and regularized Multi-Layer Perceptrons.
5. **Modern Pipeline-Native Modeling and Diagnostics:** Streamed tables into topological **UMAP** embeddings (`umap-reduce`), histogram-binned **LightGBM** gradient boosting (`lgbm-regress` with split-gain diagnostics in `lgbm_importance.png`), vectorized in-process **DuckDB**

SQL regressions (`duckdb-ols` paired with residual checks in `wine_residuals.png`), and five-family classification benchmarks (`ml-benchmark` and `benchmark_comparison.png`).

With every modeling stage grounded in inspectable scripts, explicit schemas, and visual diagnostics, we are ready to step back and synthesize the broader scientific and engineering lessons of the book.

10 Conclusion: The Command Line as a Data Intelligence Laboratory

Across nine chapters, we have rebuilt the classical command-line data science workflow around a simple architectural premise: data in transit should retain its mathematical and semantic structure. What began with container provisioning and basic tabular navigation grew into HTTP API ingestion, lazy columnar transformations in Polars, multithreaded pipeline orchestration, and polyglot statistical modeling across C++, Rust, R, and Python runtimes. Stepping back from individual commands and estimators, we can now examine why this architecture works, where its physical boundaries lie, and how to deploy it responsibly in scientific research and production engineering.

10.1 The OSEMN Lifecycle in Typed Shells

When Hilary Mason and Chris Wiggins introduced the OSEMN taxonomy (*Obtain*, *Scrub*, *Explore*, *Model*, and *iNterpret*) in 2010, practitioners faced a sharp divide between two incompatible environments. On one side sat the POSIX shell (`bash`, `zsh`, `awk`, `sed`), unmatched for file orchestration, process composition, and unattended scheduling, yet blind to data types. Every pipe carried an undifferentiated stream of bytes terminated by newlines. Adding a comma inside a quoted string, encountering a locale-dependent decimal separator, or passing missing values downstream routinely corrupted analytical results without raising a syntax error. On the other side sat monolithic interactive workspaces, which understood matrices and data frames internally but isolated those structures from the operating system, encouraging sprawling stateful sessions that resisted automation.

A typed shell closes that historical split. By elevating tables, records, lists, timestamps, file sizes, and durations into first-class values enforced by the shell parser itself, Nushell allows all five phases of OSEMN to share a single compositional substrate:

Analytical Dimension	Legacy POSIX Text Streams (`bash` + `awk`)	Monolithic Interactive Notebooks	Typed Shell Pipelines (`nu` + `polars` + `dsin`)
Data Representation	Untyped byte streams split on ad-hoc delimiters	In-memory objects bound to a single language kernel	Strongly typed tables, records, and Arrow arrays
Failure Mode	Silent column shifts and lexical misparsing	Out-of-order cell mutation and hidden global state	Immediate parse-time or schema-mismatch errors
Polyglot Integration	Fragile text scraping across stdout	Foreign-function wrappers or manual file exports	Explicit JSON/CSV/IPC contracts into typed tables

Reproducibility	High when scripted, brittle on edge cases	Low without strict execution-order discipline	Deterministic scripts inside versioned containers

This structural continuity alters how an analyst thinks. During **Obtain** (Chapter 5), responses from REST endpoints, SQLite queries, Excel workbooks, and Parquet files enter the workspace with explicit schemas rather than raw text buffers. During **Scrub** (Chapter 6), type conversions, null imputation, relational joins, and regular-expression extractions become verifiable schema transformations rather than fragile character slicing. **Explore** (Chapter 7) happens directly alongside the pipeline: summary statistics via `math` and `polars summary`, interactive inspection via `explore`, and high-resolution ggplot2 or matplotlib diagnostics rendered inline with `timg` keep empirical skepticism close to the raw measurements.

Most importantly, **Model** and **iNterpret** (Chapters 8 and 9) cease to be disconnected endpoints. Feature matrices flow into compiled C++ learners or statistical scripts, while predictions, coefficient tables, variance inflation factors, silhouette scores, and residual diagnostics return immediately as Nushell tables. Interpretation is no longer an afterthought pasted into a static slide deck; it is an executable query over model outputs, reproducible from raw source data in a single command.

10.2 The Intersection of Software Engineering and Statistics

Statistical methodology and software engineering matured under different epistemic pressures. Classical statistics prioritizes estimator properties: identifiability, bias-variance trade-offs, asymptotic consistency, and uncertainty quantification. Software engineering prioritizes system invariants: modularity, explicit interfaces, idempotency, version control, and automated testing. Applied data science fails most often not because an analyst chose the wrong loss function, but because the surrounding computational workflow violated basic software engineering invariants.

Consider a familiar failure pattern in academic and industrial modeling. An exploratory notebook begins with clean intentions. Over weeks of iteration, cells are executed out of sequence, an upstream filter is edited without rerunning cell 14, an unrecorded `pip install` upgrades `scikit-learn` on a laptop, and a subtle data leakage path creeps into cross-validation. Months later, when a thesis committee, journal reviewer, or production team attempts to reproduce Table 3 from raw data, the script either crashes or yields materially different numbers.

Treating the terminal as a structured laboratory enforces four software engineering disciplines directly onto statistical practice:

1. **Referential Transparency in Data Transformations:** In Nushell, values are immutable by default (`let`). Piping a table through `where`, `select`, `insert`, or `polars with-column` produces a new value without mutating upstream bindings in place. You can re-execute any pipeline expression ten times and obtain the exact same table, eliminating the temporal coupling that plagues mutable global state.

2. **Explicit Interface Boundaries:** Throughout Chapter 9, we deliberately avoided burying opaque, hundred-line Python or R strings inside shell functions. Instead, we stored statistical routines in standalone `.R` and `.py` files (`cluster_eval.py`, `regress_wine.R`, `ml_benchmark.py`), inspected them with `bat`, verified their input/output contracts directly on standard streams, and only then wrapped validated pipelines inside typed custom commands (`export def` in `dsin_tools.nu`). Separating orchestration from numerical estimation lets each language do what it does best while keeping every file independently testable and readable in `git diff`.
3. **Hermetic Environment Specification:** Statistical reproducibility requires pinning the entire execution stack, not merely setting a pseudorandom seed (`set.seed(42)` or `random_state=42`). The `dsin` container codifies OS libraries, the Nushell binary and its compiled plugins, R packages, compiled C++ engines (`vw`, `tapkee`, `lightgbm`, `duckdb`), and an isolated Python 3.12 virtual environment managed by `uv` (`/opt/modern-ml`). Even conflicting dependency trees, such as legacy `skll` on system Python alongside modern `umap-learn` and `scikit-learn`, coexist cleanly through explicit path resolution rather than ad-hoc environment activation.
4. **Executable Documentation and Continuous Verification:** Every pipeline in a scientific project should be backed by automated execution assertions. When documentation code blocks and modeling scripts run continuously inside a clean container (`test_md_execution.nu` and `test_modeling_pipelines.nu`), bit-rot becomes impossible to hide. Schema drift, deprecated flags, or broken file paths fail immediately at the verification gate rather than silently corrupting downstream inference.

When statistical rigor meets modular software design, exploratory agility and production reliability reinforce each other instead of competing.

10.3 Computational Efficiency, Out-of-Core Limits, and the Zero-Copy Arrow Frontier

A recurring lesson across Chapters 6, 8, and 9 is that syntax elegance means little if an analytical workflow collapses as sample size n or dimensionality p grows. Understanding how command-line tools interact with CPU caches, thread pools, memory allocators, and disk I/O separates a practitioner who can analyze five thousand rows from one who can engineer systems for five hundred million.

At small to moderate scales (10^3 to 10^6 rows), native Nushell tables provide rich ergonomics for row-wise inspection, nested JSON manipulation, and configuration management. Because native Nushell represents tables as collections of tagged value records, however, fine-grained row-by-row closures (`each { ... }`) incur per-element allocation and dynamic dispatch costs. Crossing into `nu_plugin_polars` shifts execution onto a fundamentally different physical layout: contiguous, cache-aligned columnar buffers governed by the Apache Arrow specification. When a pipeline invokes `polars into-lazy`, Polars constructs a logical query plan, pushes row predicates down to the CSV or Parquet reader, prunes unreferenced columns before decompression, and executes SIMD-vectorized kernels across all CPU cores prior to `polars collect`.

As datasets grow beyond available RAM or require specialized numerical solvers, the polyglot CLI tools examined in Chapter 9 exploit three distinct algorithmic strategies:

- **Streaming Out-of-Core Optimization (`vw`):** Vowpal Wabbit decouples memory consumption from dataset cardinality n. By hashing raw feature names into a fixed-width weight vector of size 2^b via the MurmurHash3 trick and updating weights incrementally with adaptive online gradient descent, `vw` processes multi-gigabyte text or tabular streams in bounded $O(2^b)$ memory during a single sequential pass over disk or network pipes.
- **Vectorized In-Process Analytical SQL (`duckdb`):** DuckDB processes columnar data in L1/L2-cache-sized vectors (typically 2,048 tuples per chunk). Aggregate operators such as `regr_slope`, `regr_intercept`, and `regr_r2` accumulate sufficient statistics ($\sum x_i$, $\sum y_i$, $\sum x_i^2$, $\sum x_i y_i$) in tight compiled C++ loops without materializing intermediate Python or R data frames.
- **Histogram-Binned Tree Induction (`lightgbm`):** Exact greedy split finding in gradient boosted trees requires sorting continuous feature values at $O(n \cdot p)$ cost per tree level. LightGBM quantizes continuous predictors into discrete integer bins (commonly 255 buckets), reducing split evaluation to $O(\text{bins} \cdot p)$ histogram subtraction with cache-friendly sequential memory access.

Even with fast engines, polyglot pipelines pay a measurable tax at process boundaries whenever structured tables are serialized to UTF-8 text (`to csv` or `to json`) and parsed back (`from csv` or `from json`). For summaries, model metrics, or medium-sized datasets (like the 6,497 wine records or 4,521 banking profiles), text serialization takes only a few milliseconds and buys universal inspectability. When tens of millions of rows cross process boundaries repeatedly, however, UTF-8 parsing and float-to-ASCII conversion dominate wall-clock runtime.

Two practical remedies address this bottleneck today and define the roadmap ahead. First, pipelines handling large intermediate tables should exchange binary columnar files (Apache Parquet or Arrow IPC Feather) on tmpfs (`/dev/shm` or `/tmp`), allowing Polars, DuckDB, Python (`pyarrow`), and R (`arrow`) to map typed arrays with minimal parsing overhead. Second, because Nushell and `nu_plugin_polars` are implemented in Rust atop Apache Arrow, native Rust plugins can exchange Arrow `RecordBatch` pointers in memory without serialization. As Rust machine learning crates (`linfa`, `smartcore`, `candle`, `burn`) mature alongside Nushell's plugin protocol, zero-copy tabular modeling directly inside typed shell pipelines is rapidly moving from architectural prototype to practical reality.

10.4 Methodological Guide for Applied Production Projects

Graduating from textbook exercises to real-world statistical consulting, academic thesis pipelines, or industrial data engineering requires disciplined operational habits. The five methodological principles below distill the practices developed throughout this book into an actionable field guide:

1. **Enforce Schema Contracts at the Perimeter:** Treat external files, scraped HTML tables, and third-party APIs as untrusted until validated. Immediately upon ingestion, inspect column types with `describe`, assert expected row counts and key uniqueness, normalize col-

umn names, and convert ambiguous strings into explicit numeric or temporal types. Catching a malformed timestamp or unexpected `null` in line three of an ingestion script saves hours of debugging singular matrices or `NaN` loss values inside downstream learners.

2. **Escalate Complexity Along a Deliberate Tool Hierarchy:** Resist reaching for heavy distributed frameworks or deep neural architectures by default. Start with native Nushell and lazy Polars pipelines for filtering, aggregation, and feature engineering. Move to vectorized SQL (`duckdb-ols`) or linear models (`lm`, Ridge, `vw`) to establish fast, interpretable statistical baselines. Introduce tree ensembles (`lightgbm`, Random Forests) or nonlinear manifold projections (`umap-reduce`, `tapkee-reduce`) only when baseline diagnostics demonstrate nonlinear structure that justifies additional model complexity, a pattern confirmed empirically on tabular benchmarks (Grinsztajn et al., 2022).
3. **Keep Orchestration Declarative and Computation Modular:** Let Nushell own directory traversal, parallel job dispatch (`par-each` with bounded `--threads`, or GNU `parallel`), schema validation, and metric aggregation. Keep nontrivial mathematical routines in standalone, concise `.R` or `.py` scripts with explicit CLI flags (`argparse` or `commandArgs`), bounded thread counts (`OMP_NUM_THREADS`) to prevent oversubscription under parallel orchestration, and clean JSON or CSV output on `stdout`. Once a workflow stabilizes across projects, promote its boilerplate into a documented custom command inside a versioned Nushell module.
4. **Pair Scalar Metrics with Visual and Structural Diagnostics:** Summary statistics alone frequently deceive. High overall accuracy on imbalanced banking classes can mask poor minority-class recall; a respectable R^2 can hide systematic heteroscedasticity or discrete target quantization (`wine_residuals.png`); an arbitrary cluster count k can slice continuous density gradients into spurious partitions (`kmeans_elbow.png`). Build lightweight diagnostic plots (`ggplot2`, `matplotlib`) directly into terminal scripts and inspect them via `timg` alongside tabular cross-tabulations before signing off on any model.
5. **Package Every Project as a Self-Contained Reproducible Artifact:** A completed analysis is not a folder of scattered scripts and manual instructions; it is an automated pipeline anchored by version control (`git`), a pinned environment definition (`Dockerfile`, `uv.lock`, or `renv.lock`), a clear `README.md`, and an automated verification script that rebuilds outputs from raw data with exit code `0`.

```
open raw/measurements.parquet
| polars into-df
| polars drop-nulls
| polars collect
| polars into-nu
| lgbm-regress --target response --trees 200
| get metrics
| where rmse < 0.65
```

Mastering the command line with Nushell does not ask you to abandon the statistical depth of R, the machine learning ecosystem of Python, or the raw speed of compiled C++ and Rust. Rather, it gives all of those ecosystems a shared, strongly typed table language where small, transparent programs compose cleanly. Equipped with structured pipelines, containerized reproducibility, and

statistical judgment, your terminal becomes far more than a utility for launching programs: it becomes a disciplined, enduring laboratory for turning raw data into reliable scientific knowledge.

11 For Further Exploration

11.1 The Command Line

- **Janssens, J. (2021).** *Data Science at the Command Line* (2nd ed.). O'Reilly Media. https://datascienceatthecommandline.com/
- **Shotts, W. (2019).** *The Linux Command Line: A Complete Introduction* (2nd ed.). No Starch Press.
- **Nushell Official Documentation:** https://www.nushell.sh/book/

11.2 Shell Programming

- **The Nushell Cookbook:** Community patterns and idiomatic scripts. https://www.nushell.sh/cookbook/
- **Robbins, A. & Beebe, N. (2005).** *Classic Shell Scripting*. O'Reilly Media.

11.3 Python, R, and SQL

- **Wickham, H. & Grolemund, G. (2017).** *R for Data Science*. O'Reilly Media. https://r4ds.had.co.nz/
- **McKinney, W. (2022).** *Python for Data Analysis* (3rd ed.). O'Reilly Media.
- **Beaulieu, A. (2020).** *Learning SQL* (3rd ed.). O'Reilly Media.

11.4 APIs

- **Richardson, L. & Ruby, S. (2007).** *RESTful Web Services*. O'Reilly Media.
- **Google Maps Platform Documentation:** https://developers.google.com/maps
- **Wikimedia EventStreams API:** https://stream.wikimedia.org/

11.5 Machine Learning

- **James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021).** *An Introduction to Statistical Learning with Applications in R* (2nd ed.). Springer.
- **Géron, A. (2022).** *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow* (3rd ed.). O'Reilly Media.
- **LightGBM Documentation:** https://lightgbm.readthedocs.io/
- **DuckDB Documentation:** https://duckdb.org/docs/
- **UMAP Documentation:** https://umap-learn.readthedocs.io/
- **SmartCore (Machine Learning in Rust):** https://github.com/smartcorelib/smartcore
- **Linfa (A Rust Machine Learning Framework):** https://github.com/rust-ml/linfa

- **Candle (Minimalist ML Framework for Rust by Hugging Face):** https://github.com/huggingface/candle
- **Burn (Flexible Deep Learning Framework in Rust):** https://github.com/tracel-ai/burn
- **NuTorch (Tensor Operations for Nushell):** https://github.com/astrohackerlabs/nutorch
- **Vowpal Wabbit Documentation:** https://vowpalwabbit.org/
- **SKLL (SciKit-Learn Laboratory):** https://skll.readthedocs.io/
- **Tapkee Dimension Reduction Library:** https://tapkee.lisitsyn.me/

12 References

12.1 Nushell Ecosystem

- Nushell Project: https://www.nushell.sh/
- Nushell GitHub Repository: https://github.com/nushell/nushell
- Nu Plugin Polars: https://github.com/nushell/nu_plugin_polars
- Nushell Command Reference: https://www.nushell.sh/commands/

12.2 Data Science at the Command Line (DSitCL)

- Janssens, J. (2021). *Data Science at the Command Line* (2nd ed.). O'Reilly Media. Online: https://datascienceatthecommandline.com/
- Companion Repository: https://github.com/jeroenjanssens/data-science-at-the-command-line

12.3 Command-Line Tools Integrated into `dsin`

- **ttt:** Terminal Text Editor. https://github.com/liamg/ttt
- **timg:** Terminal Image and Video Viewer. https://timg.sh/
- **bat:** A cat clone with syntax highlighting and Git integration. https://github.com/sharkdp/bat
- **GNU parallel:** A shell tool for executing jobs in parallel. https://www.gnu.org/software/parallel/
- **DuckDB:** Fast in-process analytical SQL database management system. https://duckdb.org/
- **LightGBM:** Fast, distributed, high-performance gradient boosting framework based on decision tree algorithms. https://github.com/microsoft/LightGBM
- **UMAP (`umap-learn`):** Uniform Manifold Approximation and Projection for dimension reduction. https://github.com/lmcinnes/umap
- **uv:** Fast Rust-based Python package and environment manager. https://github.com/astral-sh/uv
- **Vowpal Wabbit:** Fast out-of-core online learning system. https://vowpalwabbit.org/

- **Tapkee:** Fast C++ template library for dimensionality reduction. https://tapkee.lisitsyn.me/
- **SKLL:** SciKit-Learn Laboratory for command-line machine learning. https://skll.readthedocs.io/
- **ddgr:** DuckDuckGo from the terminal. https://github.com/jarun/ddgr
- **SmartCore (`smartcore-cli`):** SmartCore Development Team (2024). *SmartCore: Machine Learning in Rust*. https://github.com/smartcorelib/smartcore
- **Linfa (`linfa-cli`):** Linfa Contributors (2024). *Linfa: A Rust Machine Learning Framework*. https://github.com/rust-ml/linfa
- **Candle (`candle-cli`):** Hugging Face (2024). *Candle: Minimalist ML Framework for Rust*. https://github.com/huggingface/candle
- **Burn (`burn-cli`):** Burn Contributors (2024). *Burn: A Flexible and Comprehensive Deep Learning Framework in Rust*. https://github.com/tracel-ai/burn
- **NuTorch (`nutorch`):** AstroHacker Labs (2024). *NuTorch: Tensor Operations and Numerical Computing for Nushell*. https://github.com/astrohackerlabs/nutorch

12.4 Academic Bibliography and Literature

- **Cortez, P., Cerdeira, A., Almeida, F., Matos, T., & Reis, J. (2009)**. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4), 547-553.
- **Garnett, R., Do, C., & Lisitsyn, S. (2014)**. Tapkee: An efficient dimension reduction library. *Journal of Machine Learning Research*, 15, 399-403.
- **Grinsztajn, L., Oyallon, E., & Varoquaux, G. (2022)**. Why do tree-based models still outperform deep learning on typical tabular data? *Advances in Neural Information Processing Systems*, 35, 507-520.
- **Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., & Liu, T. Y. (2017). LightGBM: A highly efficient gradient boosting decision tree. *Advances in Neural Information Processing Systems*, 30, 3146-3154.
- **Langford, J., Li, L., & Strehl, A. (2007)**. Vowpal Wabbit open source project. *Technical Report*, Yahoo! Research.
- **McInnes, L., Healy, J., & Melville, J. (2018)**. UMAP: Uniform Manifold Approximation and Projection for dimension reduction. *arXiv preprint arXiv:1802.03426*.
- **Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12, 2825-2830.
- R Core Team. (2024). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. https://www.R-project.org/
- **Raasveldt, M., & Mühleisen, H. (2019)**. DuckDB: An embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19)* (pp. 1981-1984).
- Wickham, H. (2016). *ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New

York.

- **Mason, H. & Wiggins, C. (2010)**. A taxonomy of data science. *Dataists Blog*.
- Ester, M., Kriegel, H. P., Sander, J., & Xu, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD-96 Proceedings* (pp. 226-231).
- **Tukey, J. W. (1977)**. *Exploratory Data Analysis*. Reading, Mass.: Addison-Wesley Publishing Company.
- **Breiman, L. (2001)**. Random forests. *Machine Learning*, 45(1), 5-32.
- **Hastie, T., Tibshirani, R., & Friedman, J. (2009)**. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction* (2nd ed.). Springer New York.