



Pavol Jozef Šafárik University in Košice

PREDICTIVE MODELLING OF SOIL TYPES AND THEIR CHARACTERISTICS

Vasyl CHERLINKA
Michal GALLAY
Yuriy DMYTRUK

Košice 2025



Funded by the
European Union
NextGenerationEU

**[RECOVERY
AND RESILIENCE
PLAN]**



Funded by the
European Union
NextGenerationEU

[RECOVERY
AND RESILIENCE]
PLAN

Vasyl Cherlinka, Michal Gallay, Yuryi Dmytruk

Predictive Modeling of Soil Types and Their Characteristics

University textbook



Pavol Jozef Šafárik University in Košice

Košice 2025

This book was funded by the European Union's NextGenerationEU through the Recovery and Resilience Plan for Slovakia (project No. 09I03-03-V01-00049).

Predictive Modeling of Soil Types and Their Characteristics. University textbook.
Košice: UPJS, 2025, 204 p.

Authors:

doc. Vasyl Cherlinka, DrSc.

Institute of Geography, Faculty of Science, Pavol Jozef Šafárik University in Košice, Slovakia
EOS Data Analytics, Mountain View CA, USA / Kyiv, Ukraine
NGO SSELMB "Terra", Chernivtsi, Ukraine

Doc. Mgr. Michal Gallay, PhD.

Institute of Geography, Faculty of Science, Pavol Jozef Šafárik University in Košice, Slovakia

prof. Yuriy Dmytruk, DrSc.

NGO SSELMB "Terra", Chernivtsi, Ukraine
Podillia State University, Kamianets-Podilskyi, Ukraine

Reviewers:

prof. Serhii Chomyy, DrSc.

Doctor of Agricultural Sciences, Professor of the Department of Land Resources Management,
Petro Mohyla Black Sea National University, Ukraine

prof. Myroslav Zaiachuk, DrSc.

Doctor of Geographical Sciences, Professor of the Faculty of Geography, Yuriy
Fedkovych Chernivtsi National University, Ukraine

This publication is licensed under the Creative Commons 4.0 CC BY-NC-ND license. The authors bear sole responsibility for the scientific and linguistic content. The manuscript has not been subjected to editorial or language revision.

PREFACE

Why this book?

Soils underpin food security, climate regulation, biodiversity, and landscape resilience. Mapping soils and their properties at useful spatial scales is therefore a central task in environmental management. Over the last two decades, Digital Soil Mapping (DSM) has matured into a reproducible, data-driven practice that combines field observations, covariates from remote sensing and terrain analysis, and modern statistical learning. This textbook introduces DSM from first principles and shows how to implement the full workflow—data preparation, modelling, validation, and uncertainty communication—using open-source R.

Who this book is for?

The book is intended for graduate and postgraduate students in soil science, geography, geoinformatics, environmental science, and related fields, as well as for researchers and practitioners who need a practical, reproducible route into DSM.

What you will learn?

After completing the book, readers will be able to (i) set up a reproducible R/RStudio project for spatial analysis; (ii) assemble and clean soil point data and spatial covariates; (iii) fit, tune, and interpret local and global predictive models (e.g., regression, tree-based and ensemble methods); (iv) assess predictive performance and quantify uncertainty with appropriate diagnostics; (v) generate soil maps and related products (e.g., SOC, texture classes) and communicate their limitations.

How the book is organized?

The book consists of three main parts:

Part I. Fundamentals of Working in R and RStudio for Soil Science:

This part lays down the fundamental skills needed for further work. We will start by setting up the workspace, learn the basics of R syntax,

master the key tools for manipulating (dplyr) and visualizing (ggplot2) data, and most importantly, learn how to work with vector (sf) and raster (terra) geospatial data.

Part II. Predictive modeling of soil types: In this part, we will move on to the first practical task – creating a predictive map of soil classes (classification problem). We will consider in detail the theoretical foundations of the DSM, go all the way through data preparation using the example of Slovakia, teach, validate and interpret models based on Decision Trees and Random Forest.

Part III: Predictive Modeling of Soil Characteristics: The final part is devoted to modeling continuous soil properties (regression problem), focusing on organic carbon content. We will look at differences in modeling approaches, learn how to assess the accuracy of regression models, and, crucially, quantify the uncertainty of our predictions using Quantile Regression Forests.

Prerequisites and software

No prior experience with R is strictly required, though basic statistics and GIS concepts are helpful. Examples were tested with R (≥ 4.3) and widely used packages (e.g., sf, terra, dplyr, ggplot2, caret, ranger, randomForest, Cubist, tidymodels). Full package details and a reproducible session log are provided in the appendices. We recommend working inside an RStudio Project with relative paths to ensure portability.

Data, code, and reproducibility

All datasets and scripts used in the book are provided as companion materials so that readers can reproduce every figure and table. Where third-party figures or data are reused, the original authors and licenses are credited. Supplementary datasets used in Parts II and III are available for download; see the Appendix B.

Acknowledgements

We are grateful to our colleagues and students whose questions shaped many of the examples. We thank the manuscript reviewers for their constructive feedback. This textbook was prepared with the support of NextGenerationEU under the Recovery and Resilience Plan of the Slovak Republic, Project No. 09I03-03-V01-00049, and the grants of The Ministry of Education, Research, Development and Youth of the Slovak Republic No. VEGA 1/0168/22 and No. VEGA 1/0780/24. The authors bear sole responsibility for any remaining inaccuracies or errors in the book.

Košice, August 2025

Authors

TABLE OF CONTENTS

Introduction	1
Part I. Fundamentals of working in R and RStudio for soil science	5
Chapter 1. Installing and configuring the work environment	5
1.1. Introduction to R and RStudio	5
1.2. Installation Guide	7
1.3. Navigating the RStudio environment	12
1.4. The importance of projects for the reproducibility of research.....	14
1.5. R Package Management	16
Chapter 2. Fundamental concepts of R.....	21
2.1. R as a calculator	21
2.2. Objects and assignments.....	23
2.3. Data types and structures (vectors, factors, matrices, data tables, lists).....	25
2.4. Data Import and Export.....	29
Chapter 3. Data manipulation with dplyr	35
3.1. Introduction to Tidyverse and dplyr	35
3.2. Basic verbs dplyr (select, filter, mutate, arrange)	37
3.3. Strategy "Divide-Apply-Unite" (group_by, summarise)	43
3.4. Pipeline operator (%>%)	46
Chapter 4. Data visualization with ggplot2	49
4.1. Grammar of graphics.....	49
4.2. Creation of basic graphs for exploratory data analysis (histograms, scatter plots, box plots).....	52
4.3. Refining plots and figure design.....	56
Chapter 5. Working with spatial data in R	62
5.1. Modern spatial packages: sf and terra	62
5.2. Processing vector data from sf.....	67
5.3. Processing raster data from terra	73
5.4. Integration of spatial data	78
Part II. Predictive modeling of soil types	82
Chapter 6. Theoretical foundations of digital soil mapping	82
6.1. DSM Concept.....	82
6.2. Detailed overview of the SCORPAN model	84
6.3. DSM Workflow Overview	88
Chapter 7. Preparing data for modeling: the case of Slovakia.....	91
7.1. Determination of the study area.....	91
7.2. Sources of point data on soils.....	93
7.3. Collection and pre-treatment of raster covariates	97

7.4. Creating the final dataset for modeling	101
Chapter 8. Modeling with decision trees and random forest	106
8.1. Introduction to Machine Learning for Classification	106
8.2. Decision trees (rpart)	108
8.3. RandomForest	113
Chapter 9. Accuracy Assessment and Validation of Classification Models	120
9.1. Confusion Matrix	120
9.2. Overall Accuracy Metrics (Producer's Accuracy, User's Accuracy)	123
9.3. Kappa coefficient	126
9.4. Practical validation	129
Chapter 10. Creation and interpretation of predictive maps of soil types	134
10.1. Spatial forecasting	134
10.2. Map Interpretation	138
Part III. Predictive modeling of soil characteristics	141
Chapter 11. Continuous Variable Modeling: Organic Carbon Content	141
11.1. Differences between Modeling of Continuous and Categorical Variables	141
11.2. Focus on Soil Organic Carbon (SOC)	144
11.3. Preparing Data for SOC Modeling (Logarithmic Transformation)	147
11.4. Exploratory Data Analysis	149
Chapter 12. Regression Models: Random Forest and Cubist	154
12.1. Random forest for regression	154
12.2. The Cubist model	158
Chapter 13. Validation of regression models and uncertainty analysis	162
13.1. Key metrics for regression (R^2 , RMSE, Bias)	162
13.2. Visual diagnostics	165
13.3. Quantification of forecast uncertainty (quantile regression forests)	168
Chapter 14. Construction of final maps and their practical application	173
14.1. Creating final maps (forecast, interval boundaries, uncertainty)	173
14.2. Reverse Conversion	178
14.3. Practical application (estimation of carbon stocks, policy justification, inputs for models)	181
Conclusions	184
APPENDIX A: List of recommended R packages	187
APPENDIX B: Data sources for the example of Slovakia	189
APPENDIX C: Glossary of Terms	191
References	194

INTRODUCTION

Soil is not an inert substrate beneath our feet but a complex, dynamic resource at the intersection of the lithosphere, atmosphere, hydrosphere, and biosphere. The health and functioning of soils underpin food security, water quality, biodiversity, and climate regulation. For decades, soil scientists have produced soil maps to systematize knowledge about this invaluable resource. Traditional soil mapping—based on field surveys and expert interpolation—has contributed enormously to our understanding of soil geography. Yet in the twenty-first century, amid big data and global environmental challenges, these approaches alone are no longer sufficient.

Conventional polygon maps are static, costly to update, and often subjective because polygon boundaries reflect the experience and judgment of individual cartographers. Most importantly, they represent the soil cover as discrete, homogeneous units that do not reflect the continuous nature of soil property variability in space. Against this backdrop, **Digital Soil Mapping (DSM)** has emerged as a rapidly developing paradigm that uses numerical and statistical methods to create spatial soil information systems (McBratney et al., 2003; Hengl & MacMillan, 2019). DSM marks a transition from art to science, from qualitative description to quantitative prediction. Figure 1 contrasts a traditional polygon map with a DSM product that shows a continuous gradient of a soil property at high spatial resolution.

The relevance and feasibility of DSM flow from three interlocking revolutions:

Geospatial revolution – unprecedented availability of global covariate datasets describing environmental factors: digital elevation models, high-resolution satellite imagery (e.g., Landsat, Sentinel), global climate surfaces, geological and topographic maps.

Computing revolution – scalable computing and the rise of open-source software, especially R, which provides powerful tools for data wrangling, modelling, and visualization (R Core Team, 2023).

Statistical revolution – the development and popularization of machine learning methods capable of capturing complex, nonlinear relationships in soil-landscape systems (e.g., ensemble trees; Breiman,

2001). DSM enables the creation of information products that go well beyond static maps: (i) continuous surfaces of key soil properties (e.g., soil organic carbon (SOC), pH, bulk density); (ii) high-resolution soil type maps suitable for field- to regional-scale decision-making; (iii) uncertainty maps that quantify the confidence in predictions—essential for risk-aware applications. These products support tasks from precision agriculture and site-specific management to national inventories and global carbon cycle assessments, where reliable estimates of soil carbon stocks have planetary significance. The three revolutions demand a toolkit that is flexible, transparent, and reproducible. R has become a de facto academic standard for data science because it is open-source, extensible through thousands of community packages, and designed by and for statisticians (R Core Team, 2023; Pebesma & Bivand, 2023). For DSM workflows, packages for spatial data (*sf*, *terra*), data manipulation (*tidyverse*), machine learning (*randomForest*, *ranger*, *caret*), and visualization (*ggplot2*) provide an end-to-end environment –

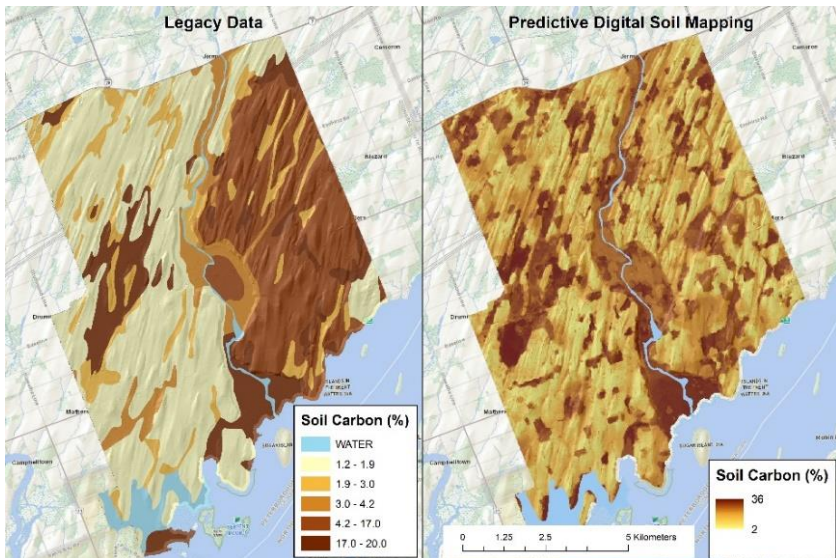


Fig. 1.1. Comparison of a soil organic carbon (%) map digitized from a conventional soil survey map (left) and a predictive digital soil map at 20 m resolution for the Keene area, Peterborough County, Ontario. © Daniel D. Saurette, Ontario Ministry of Agriculture, Food and Rural Affairs. Licensed CC BY. Adapted from Heung, Saurette, & Bulmer (2021).

from raw data to publishable figures (Breiman, 2001). A critical advantage is reproducibility: analyses are scripts, not clicks. Scripts serve as precise, transparent recipes that others can inspect and rerun, which is foundational to modern science and to the instructional goals of this book.

To orient readers for the chapters that follow, DSM generally proceeds through the following stages:

Problem framing & data assembly – define target variables, collect soil point observations, and compile environmental covariates (terrain, remote sensing, climate).

Data preparation – harmonize units, handle outliers, align coordinate reference systems, and split data for model training/validation.

Modelling – fit and tune predictive models (regression, tree-based ensembles, and other learners) appropriate to the target property and sampling design.

Spatial prediction & mapping – generate wall-to-wall predictions at the desired resolution.

Validation & uncertainty – assess accuracy with suitable diagnostics and produce uncertainty maps.

Communication & use – translate results into maps, figures, and narratives that support decisions and scientific inference.

Each chapter in this book maps to one or more of these stages and includes worked R examples, exercises, and summaries to reinforce learning. This book focuses on methods and workflows for DSM rather than exhaustive soil classification theory. While many examples are grounded in datasets from Central and Eastern Europe, the principles and code are broadly transferable. Readers should be mindful that model outputs are only as good as inputs: careful data quality control, appropriate model selection, and transparent reporting of uncertainty remain essential. This text is designed as a practical, step-by-step guide that guides the reader through the entire workflow of digital soil mapping using R. The material is structured in such a way as to provide a gradual immersion from basic concepts to complex practical tasks.

All scientific approaches, including those presented in this book, have limitations and must be understood as generalised representations of reality rather than exact reflections of the soil system. The examples in this book are drawn mainly from Central Europe, which means that model performance is tied to the local **sampling design, density, and environmental conditions** of those study areas. Predictions are most reliable within the ranges of covariates represented in the training data, while areas with sparse or unrepresentative samples usually carry higher uncertainty. Temporal mismatches between soil observations and covariates, such as when imagery and field data come from different years or seasons, can also introduce bias.

A major factor affecting the **transferability** of Digital Soil Mapping is the covariate set. Models trained with specific digital elevation models, climate grids, or satellite indices may not transfer well if these inputs differ in another region or are available at different resolutions. Resampling can alter predictor distributions and thus model behaviour. For this reason, portability improves when based on stable, widely accessible covariates, complemented by careful documentation of sources, resolutions, and preprocessing.

Model assumptions and validation methods also matter. **Non-stationarity** in soil–landscape relationships means that models fitted in one setting may not hold in another. Overfitting can arise when many correlated predictors are used with limited samples. Furthermore, ordinary random cross-validation often inflates accuracy because nearby points share information; spatial cross-validation or blocking methods give more realistic estimates.

For these reasons, predictions should always be interpreted together with **uncertainty information**. Uncertainty maps, prediction intervals, or class probabilities highlight where models are more or less trustworthy. Readers will find detailed guidance on prediction and uncertainty in Chapter 10, and on calibration and external validation in Chapter 13. Testing models against independent data from different sites or times is especially valuable for judging their true portability.

PART I. FUNDAMENTALS OF WORKING IN R AND RSTUDIO FOR SOIL SCIENCE

Chapter 1. Installing and configuring the work environment

1.1. Introduction to R and RStudio

Modern soil science, and especially its digital direction (Digital Soil Mapping, DSM), is inextricably linked with the processing of large data sets. Powerful and flexible tools are needed to effectively analyze, model, and visualize spatial information about soils. One such key tool that has become the de facto standard in the scientific world is the R programming language.

R is both a programming language and a free software environment for statistical computing and graphics. Created as a descendant of the S language developed at Bell Labs, R inherited its power but became an open-source project. This means that anyone can download, use, modify and distribute it for free. Thanks to the efforts of thousands of developers and scientists from around the world, R has grown into an extremely rich ecosystem, containing tens of thousands of extensions, or **packages** that provide functionality to solve a wide variety of tasks (Kabacoff, 2021).

For digital soil science, R is a particularly valuable platform. The success of the DSM depends largely on the ability to integrate and analyze data from a variety of sources: field survey results, Earth remote sensing data, digital terrain models, climate data, and geological maps. R offers unparalleled capabilities to perform the entire cycle of data work: from its import and cleaning to complex geostatistical modeling and the creation of high-quality cartographic materials (Malone et al., 2017). Specialized packages such as *sf* for working with vector spatial data and *terra* for working with rasters turn R into a full-fledged, code-driven geographic information system, which ensures a high level of reproducibility of research. At its core, R is an interactive environment where commands are executed through a console. You can enter commands one at a time and see the result instantly. For example, R can be used as a regular calculator or to create objects that store data.

```
# R can be used as a powerful calculator.
# The result of this operation will be printed to the
console.
(112 / 4) * 3 + 1
[1] 85

# Assigning a value to an object named 'soil_ph'.
# This object now stores the numeric value 6.5 and can be
used later.
soil_ph <- 6.5

# Print the object's value to the console.
soil_ph
[1] 6.5
```

While working directly in the R console is possible, for complex projects, which are the norm in the DSM, this approach is inefficient. It is much more convenient to use **an** Integrated Development Environment (IDE). The most popular IDE for R is **RStudio**.

RStudio is a free application that provides a user-friendly and intuitive graphical interface for working with R. It is important to understand that RStudio is not a replacement for R; It is rather a control panel for the engine. R is the engine that performs all the calculations, and RStudio is the cockpit, which makes driving this engine much more comfortable and productive (Kabacoff, 2021). RStudio organizes the workspace into four main panels, allowing you to simultaneously write code, see its results, manage objects in memory, and view graphs and reference materials.

The combination of R and RStudio creates a powerful platform that is ideal for digital soil science tasks. Not only does it simplify code writing and debugging, but it also supports key principles of modern science, such as research reproducibility, with tools for project management, integration with version control systems (such as Git), and dynamic reporting (R Markdown). It is this combination that will be our main working tool throughout the manual.

In the following sections, we will take a closer look at the process of installing R and RStudio, familiarize ourselves with the interface, and learn how to manage projects and packages. These initial steps are the foundation on which we will build our skills to further immerse ourselves in the world of predictive soil modeling.

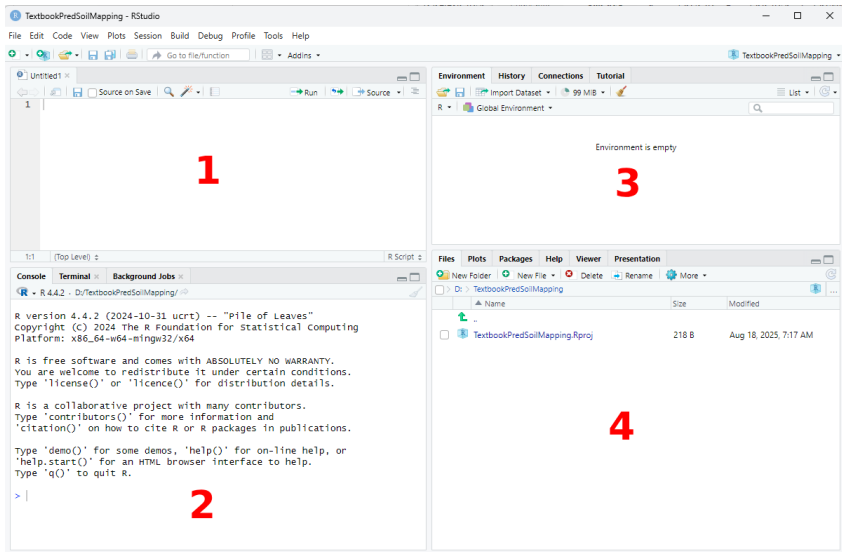


Fig. 1.2. RStudio interface. The four main panels provide an integrated environment for writing code (top left), executing commands (bottom left), viewing objects and history (top right), and accessing files, graphs, and help (bottom right)

1.2. Installation Guide

Before we can dive into the world of data analysis and digital soil science, it is necessary to set up our work environment. This process consists of two main steps: installing the R language itself, which is the computing core, and installing RStudio, an integrated development environment that will provide us with a user-friendly interface. It is important to follow the correct sequence: **R is installed first, and only then – RStudio**. This is due to the fact that RStudio is a shell and requires an already installed R "engine" for its operation.

Step 1: Installing R

The official and most reliable source for downloading R is **the Comprehensive R Archive Network (CRAN)**. It is a network of FTP and web servers around the world that store identical, up-to-date versions of R code and documentation. Open a web browser and go to the main page of Project R at: <https://www.r-project.org/>.

- On the main page, find the "download R" link in the "Getting Started" section. You will be redirected to the CRAN mirror selection page. A mirror is a server that is a copy of the main repository. You can choose any mirror geographically close to you for faster loading, or simply use the link <https://cran.r-project.org/> that will automatically direct you to the corresponding server.
- On the main page of CRAN you will see links to download R for different operating systems: "Download R for Linux", "Download R for macOS" and "Download R for Windows". Select the link that matches your system.

For Windows: Click on the "base" link. This is the basic distribution that contains everything you need to get started. On the next page, click on the large "Download R [version] for Windows" link. This will download the installation file (e.g. R-4.3.2-win.exe).

For macOS: Select the installation package (.pkg) that matches your version of macOS and processor architecture (Intel or Apple Silicon/ARM).

Once the installer is downloaded, launch it and follow the instructions. In most cases, it is safe to accept all default settings. There is no need to change the installation directory or the components to be installed.

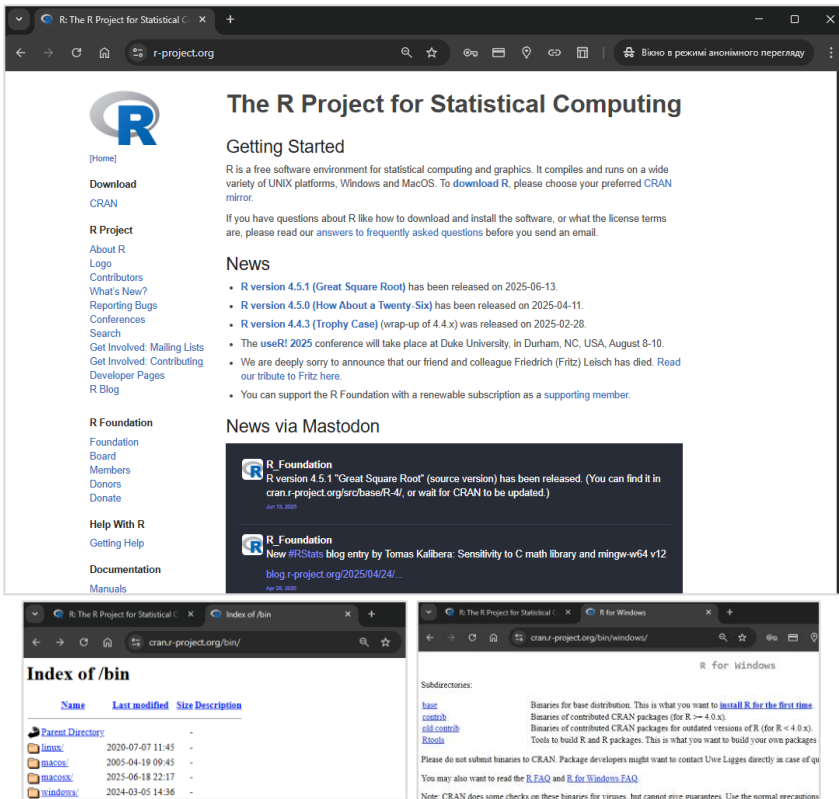


Fig. 1.3. Home page of the Comprehensive R Archive Network (CRAN). R download links shown for Linux, macOS, and Windows operating systems

Step 2: Installing RStudio

Once R has been successfully installed, you can proceed to install RStudio. RStudio is developed by Posit (formerly known as RStudio, PBC). Go to the official website of Posit to download RStudio Desktop: <https://posit.co/download/rstudio-desktop/>.

- The site will prompt you to download RStudio Desktop. There are several versions of the product, including commercial ones, but for our purposes, the free version of **RStudio Desktop**, which is free and open-source, is quite sufficient.

Click the download button. The site will automatically detect your

operating system and suggest the appropriate installation file. If it doesn't, scroll down the page to find installer lists for Windows, macOS, and various Linux distributions.

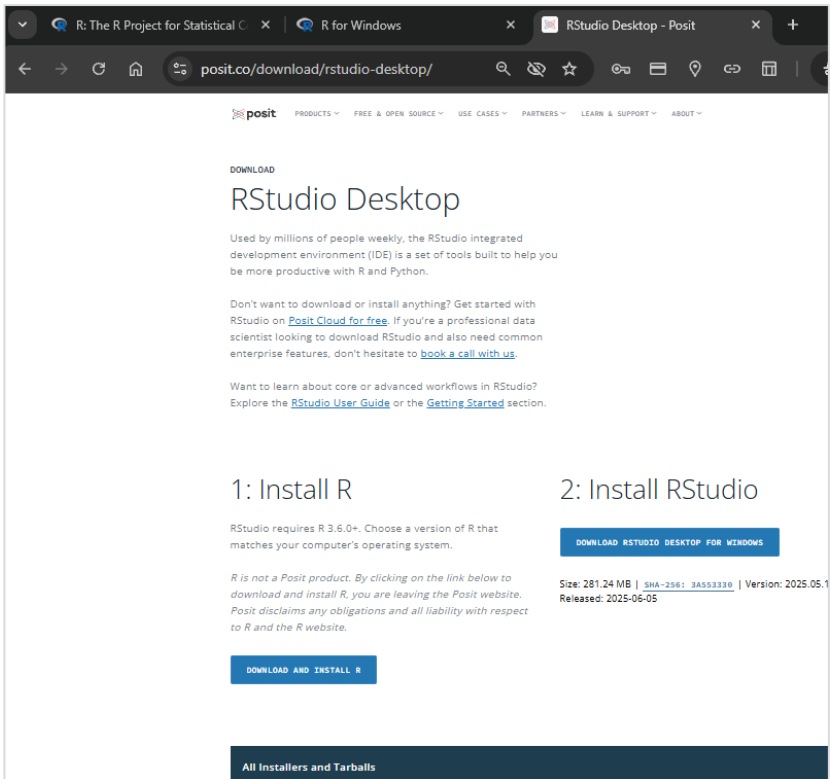


Fig. 1.4. RStudio Desktop download page. Shows the selection of the free version to download and installers for different operating systems

Download and run the installer. As with R, the installation process is standard and you can accept all settings by default.

Once the installation is complete, RStudio will automatically find the installed version of R on your computer and connect to it.

Step 3: Checking the Installation

To make sure everything is installed correctly, launch **RStudio** and

not R itself.

In the console, you will see a greeting and information about the version of R. To finally check the health of the system, type the following command in the console and press Enter. This command displays detailed information about your current session, including the version of R, operating system, and downloaded packages.

```
# This command provides details about the current R
session.
# It's a good way to check that everything is working
correctly.
sessionInfo()
```

If the installation is successful, you will see a response similar to this (versions may vary):

```
R version 4.4.2 (2024-10-31 ucrt)
Platform: x86_64-w64-mingw32/x64
Running under: Windows 11 x64 (build 22631)

Matrix products: default

locale:
[1] C

time zone: Europe/Budapest
tzcode source: internal

attached base packages:
[1] parallel stats      graphics  grDevices utils
datasets methods  base

other attached packages:
[1] doParallel_1.0.17 iterators_1.0.14 foreach_1.5.2
tictoc_1.2.1      terra_1.8-42      here_1.0.1

loaded via a namespace (and not attached):
[1] compiler_4.4.2    rprojroot_2.0.4    tools_4.4.2
rstudioapi_0.17.1 Rcpp_1.0.13        codetools_0.2-20
```

If you see a similar output without error messages, congratulations! Your work environment is ready. Now we can move on to getting to

know the RStudio interface.

1.3. Navigating the RStudio environment

After successfully installing and running RStudio for the first time, you will see an integrated environment (Fig. 1.2), which, at first glance, may seem complicated due to the large amount of information. However, its structure is logical and designed for maximum performance. By default, the RStudio workspace is divided into four main panels (or three if you haven't opened any script files yet). Understanding the purpose of each of these panels is key to working effectively.

Panel 1: Source Editor

This panel appears in the upper left corner when you open or create a new file, such as an R script (a file with the extension . R). This is your main workspace, a text editor specifically adapted for writing code in R.

- **Writing and editing code:** This is where you write sequences of commands that make up your analysis. Unlike the console, the code in the editor is not executed instantly. It allows you to prepare, edit, annotate, and structure your work before execution.
- **Syntax highlighting:** RStudio automatically colors various code elements (functions, variables, comments), which greatly improves its readability.
- **Code Autocompletion:** When typing, RStudio suggests options for function names, objects, and their arguments, which speeds up work and reduces errors.
- **Code execution:** You can execute code directly from the editor. To do this, there are special buttons or keyboard shortcuts (such as Ctrl+Enter or Cmd+Enter) that send the current line or a dedicated block of code to the console for execution.
- Working in the code editor is the basis for reproducible research. By saving your commands as scripts, you can always go back to them, modify them, or pass them on to colleagues who can fully reproduce your analysis.

Panel 2: Console

The panel in the lower left corner is a direct access to the "engine" R. Every command you execute, whether from the code editor or typed directly into the console, is handled here.

- **Interactive operation:** The console is ideal for quick calculations, testing individual commands, or checking object values.
- **Invitation Symbol (>):** This symbol indicates that R is ready to accept a new command.
- **Output results:** Command results, error messages, and warnings are displayed in the console.

While the console is a powerful tool for interactive work, for basic analysis, always prefer a code editor.

Panel 3: Environment, History, etc.

This panel, located in the upper right corner, contains several tabs providing information about the current work session.

- **Environment:** This is one of the most useful tabs. It shows a list of all the objects (variables, datasets, functions) that you have created that are currently stored in R memory.
- **History:** A chronological list of all commands that have been executed in the console is stored here. This is useful if you want to find and reuse any of the previous commands.
- **Connections:** This tab allows you to manage connections to external databases.

Panel 4: Files, Plots, Packages, Help

The lower right panel is multifunctional and also consists of several tabs.

- **Files:** Works as a simple file manager, showing the contents of your current working directory. You can navigate through folders, open files, rename them, and more.
- **Plots:** When you create a visualization with code, the result appears in this tab. RStudio makes it easy to view graphs, swipe between them, enlarge and export in various formats (PNG, PDF, JPG).
- **Packages:** Shows a list of all R packages installed on your

system.

- **Help:** Built-in help system R. Using the ? (e.g. ?mean) or by searching here, you can get detailed documentation for any feature or package.
- **Viewer:** Used to display local web content, such as interactive maps or reports created with specialized packages.

Once you've mastered navigating between these four panels, you'll be able to organize your work as efficiently as possible. The entire cycle of analysis – from writing code and executing it to viewing results, objects and graphs – takes place in a single, logically organized space.

1.4. The importance of projects for the reproducibility of research

Having mastered the basic navigation in the RStudio environment, we come to one of the fundamental concepts that underlies organized and, most importantly, reproducible scientific work – the use of **RStudio Projects**. Any data analysis, especially in such a complex field as digital soil science, quickly becomes overgrown with a large number of files: R scripts, input data sets (rasters, vector layers, tables), intermediate results, final maps and reports. Without proper organization, managing this chaos becomes almost impossible.

The RStudio project is, in fact, a way to encapsulate all the components of one analytical task into a single self-contained directory (folder). When you create a project, RStudio generates a special file with the extension . Rproj. This file does not contain your code or data, but it does "remember" settings related to this project, such as which files were last opened in the editor. Opening . Rproj file, you instantly return to the desktop environment in the exact state in which you left it.

However, the main advantage of projects is not so much convenience as solving one of the most common problems that destroys the reproducibility of analysis – **managing the working directory**. The working directory is the place on your computer from where R tries to read files by default and where it stores the results. Beginners often use the setwd() function to specify the path to this directory, which results in the following lines appearing in the code:

```
# Bad practice: Using an absolute path with setwd()
```

```
# This code will fail on any computer other than the
author's.
setwd("C:/Users/YourNicName/Documents/Soil_Analysis/Slova
kia_Project/Data")
soil_samples <- read.csv("samples.csv")
```

This approach is fragile and completely unreproducible. As soon as you try to run this script on another computer or simply move the project folder, the code will stop working because the specified absolute path no longer exists.

RStudio projects solve this problem elegantly. When you open a project, RStudio **automatically sets the working directory to the root folder of that project**. This means that you can reference any file within the project using **relative paths** that start from the root of the project. For example, if you have a data folder inside the project, the code to read the file would look like this:

```
# Good practice: Using relative paths within an RStudio
Project
# This code is portable and will work on any machine.
soil_samples <- read.csv("data/samples.csv")
```

This code will work on any computer, regardless of where the project folder is located on the disk. This makes your analysis portable and makes it much easier to collaborate with colleagues. You can simply archive the project folder, send it, and the other person can open . Rproj file and run your code without any changes.

How to create an RStudio project

Creating a project is a simple procedure that should be followed at the very beginning of work on a new task.

In RStudio, go to the File > New Project menu....

- A dialog box will open, offering three options. The first two are most often used to get started.
- **New Directory:** Create a project in a new, empty folder. You select a disk location, give a name to the project, and RStudio creates the appropriate folder and . Rproj file inside it. This is ideal for starting from scratch.

- **Existing Directory:** Create a project based on an existing folder with files. If you already have a folder with data and scripts, you can turn it into a project. RStudio will simply add .Rproj file to this directory.
- **Version Control:** Create a project by cloning a repository from a version control system, such as Git. This is a more advanced option for collaboration.

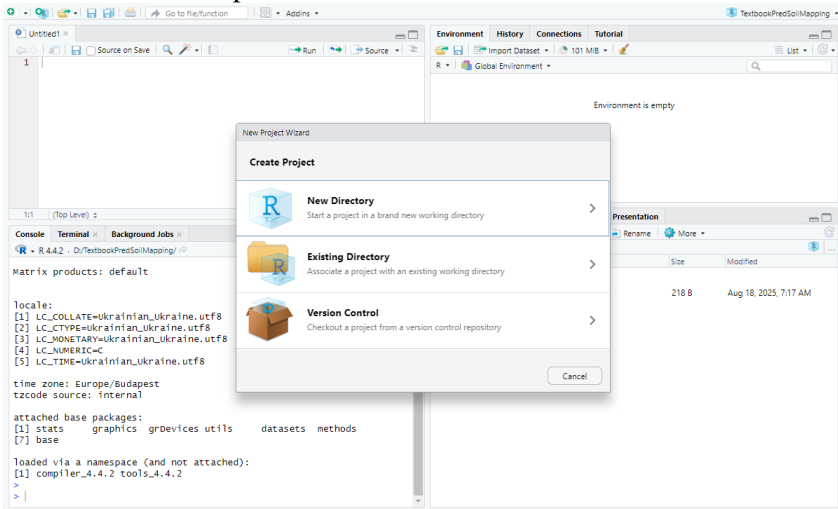


Fig. 1.5. Dialog box for creating a new project in RStudio. The options "New Directory", "Existing Directory" and "Version Control" are shown

After selecting an option and specifying a name and location, RStudio will restart and open a new project. You will notice that the project name has appeared in the upper right corner of the RStudio window, and the path in the "Files" tab now leads to the root folder of your project.

The habit of organizing each individual study or analysis into your own project is one of the most important steps towards professional and reproducible data work in R. It disciplines, simplifies file management, and ensures that your analysis can be replicated not only by you in the future, but by anyone else.

1.5. R Package Management

One of the main reasons for R's phenomenal popularity in academia is its extensibility. A basic installation of R contains a set of fundamental functions for mathematical operations, statistics, and graphing. However, the true power of R is revealed through the use of packages. A package is a standardized collection of functions, data, and compiled code created by the developer community to solve specific problems. the main package repository, **CRAN (Comprehensive R Archive Network)**, had over 22500 unique packages, making R a one-stop tool for any industry, including soil science.

For our tasks in digital soil mapping, we will actively use packages for data manipulation (dplyr), visualization (ggplot2), working with spatial data (sf, terra) and machine learning (randomForest, rpart). Being able to manage packages efficiently – installing, downloading, and updating them – is a basic skill for any R user.

Installing packages

The process of installing a package is similar to installing a new app on your smartphone: you do it once and it becomes available for use. The installation downloads the package from the CRAN repository and places it in a special folder on your computer called a library.

The easiest way to install a package is to use the `install.packages()` function. The package name must be quoted as follows. For example, we install the tidyverse package, which is a meta package containing a set of the most popular tools for working with data, including dplyr and ggplot2.

```
# Installing a package from CRAN.
# The package name must be in quotes.
# This command needs to be run only once.
install.packages("tidyverse")
Installing package into 'C:/Users/eucrariano/AppData/Local/R/win-library/4.4'
(as 'lib' is unspecified)trying URL
'https://cran.rstudio.com/bin/windows/contrib/4.4/tidyverse_2.0.0.zip'
Content type 'application/zip' length 431634 bytes (421 KB)
downloaded 421 KB

package 'tidyverse' successfully unpacked and MD5 sums checked

The downloaded binary packages are in
C:/Users/eucrariano/AppData/Local/Temp/RtmpGaTGwN/downloaded_packages
```

RStudio also provides a user-friendly graphical interface for managing packages. In the bottom right pane, go to the "Packages" tab and click the "Install" button. In the dialog box that appears, simply enter the name of the package and RStudio will execute the appropriate command for you.

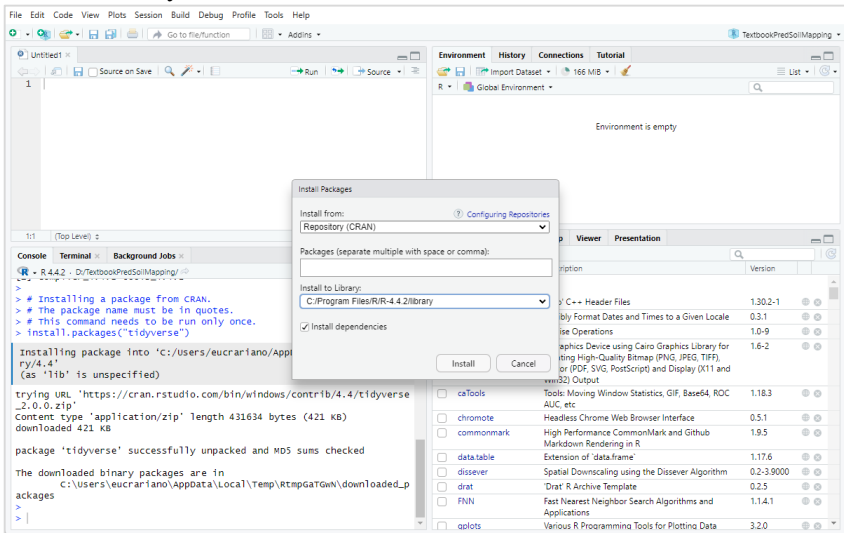


Fig. 1.6. The "Packages" tab in RStudio. The "Install" button and a list of already installed packages with a brief description are shown

Downloading packages

Installation makes the package available on your system, but in order to use its functions and data in the current R session, the package must **be downloaded**. This process can be compared to taking a book from a shelf (installation) and opening it to read (download). Every time you start a new R session (e.g. restarting RStudio), you need to re-download the packages, that you plan to use.

To load the package, use the `library()` function. This time, the package name is indicated without quotation marks.

```
# Loading a package into the current R session to make
# its functions available.
# The package name is typically not in quotes.
```

```
# This command must be run at the beginning of every new
session.
library(tidyverse)
— Attaching core tidyverse packages — tidyverse 2.0.0 —
✓ dplyr 1.1.4 ✓ readr 2.1.5
✓ forcats 1.0.0 ✓ stringr 1.5.1
✓ ggplot2 3.5.1 ✓ tibble 3.2.1
✓ lubridate 1.9.3 ✓ tidyr 1.3.1
✓ purrr 1.0.2
— Conflicts —
tidyverse_conflicts() —
✗ dplyr::filter() masks stats::filter()
✗ dplyr::lag() masks stats::lag()
i Use the conflicted package to force all conflicts to become errors
Warning messages:
1: package ‘tidyverse’ was built under R version 4.4.3
2: package ‘readr’ was built under R version 4.4.3
3: package ‘forcats’ was built under R version 4.4.3
```

After executing this command, all the functions from the tidyverse package become available to you. Usually, commands for loading all the necessary packages are placed at the very beginning of the R script. This makes the code organized and immediately makes it clear which dependencies are needed to execute it.

Package updates and deletions

The R ecosystem is constantly evolving: developers are fixing bugs and adding new functionality. Therefore, it is important to periodically update installed packages to the latest versions. This can be done using the `update.packages()` function or by clicking the "Update" button on the "Packages" tab in RStudio.

If you no longer need a package, you can remove it from the library using the `remove.packages()` function, specifying the package name in quotation marks.

```
# To remove a package from your library.
remove.packages("unneeded_package")
```

Effective package management is the key to successful work at R. It

opens up access to a huge number of tools created by the global scientific community, which allows you to solve the most complex problems of predictive modeling of soils without reinventing the wheel.

Chapter 2. Fundamental concepts of R

2.1. R as a calculator

The easiest way to get started with R is to use it as a powerful desktop calculator. This approach allows us to become familiar with basic syntax, arithmetic operators, and the order in which operations are performed without delving into more complex programming concepts. To do this, we will enter the commands directly into the console – the panel at the bottom left of RStudio, which is indicated by the invitation symbol `>`.

R supports all standard arithmetic operations. You can enter mathematical expressions, and R will calculate and output the result instantly. The result in the console is usually denoted by the prefix `[1]`, which indicates that it is the first element in the result vector (we will talk more about vectors later).

The basic arithmetic operators in R are:

<code>+</code>	Adding
<code>-</code>	Subtraction
<code>*</code>	Multiplication
<code>/</code>	Division
<code>^</code> or <code>**</code>	Exponentiation

We try to do some simple calculations. Type the following lines into the console one at a time, pressing Enter after each one.

```
# Basic addition
5 + 3
[1] 8

# Basic subtraction
10 - 4
[1] 6

# Multiplication
2.5 * 4
[1] 10

# Division
100 / 8
[1] 12.5
```

R follows the standard mathematical order of operations (sometimes remembered by the acronyms PEMDAS/BODMAS): parentheses are performed first, then exponentiation, then multiplication and division (from left to right), and finally addition and subtraction (from left to right). The use of parentheses () allows you to explicitly indicate the priority of operations and avoid ambiguity.

```
> # Without parentheses, multiplication is performed
first
> 4*5+10
[1] 30
>
> # With parentheses, the addition inside is performed
first
> 4 * (5+10)
[1] 60
```

In addition to basic operators, R has a huge library of built-in mathematical functions. A function in R is a named block of code that performs a specific action. To call a function, you need to write its name, and then pass one or more arguments (input values) to it in parentheses.

For example, the `sqrt()` function calculates the square root and `log()` calculates the natural logarithm.

```
> # Calculate the square root of 81
> sqrt(81)
[1] 9
>
> # Calculate the natural logarithm of 10
> log(10)
[1] 2.302585
>
> # Calculate the base-10 logarithm of 10
> log10(10)
[1] 1
>
> # Calculate the absolute value
> ABS(-15.5)
[1] 15.5
```

R also works seamlessly with very large or very small numbers, automatically representing them in scientific notation (e.g. 2.5×10^8 means 2.5×10^8).

Using R as a calculator is a great starting point. It demonstrates the interactive nature of the console and introduces the basic syntax that is the foundation for all the subsequent, much more complex operations that we will encounter in predictive soil modeling.

2.2. Objects and assignments

Using R as a calculator is useful for quick calculations, but its true power is revealed when we start saving the results for later use. Imagine that you are performing a complex calculation whose result is required in the next few steps of the analysis. Constantly re-entering the entire formula would be inefficient and would lead to errors. Instead, we can store the result in **an object**.

In R, almost everything is an object: a number, a set of data, the results of a statistical test, a graph. An object is essentially a named storage in the computer's memory where we can put any data. To create an object, we must give it a name and assign a specific value.

The process of assigning a value to an object is done using **the <- assignment operator**. This operator looks like an arrow pointing from left to right and can be read as "gets value".

```
# Create an object named 'soil_ph' and assign it the
value 6.8
soil_ph <- 6.8
```

After executing this command, nothing will appear in the console. This is normal behavior. R silently created an object named `soil_ph` in his memory (in the current environment) and wrote the value 6.8 into it. You can see this new object in the "Environment" panel in the upper right corner of RStudio.

To view the contents of an object, simply type its name into the console and press Enter.

```
> # Print the value of the 'soil_ph' object to the
console
```

```
> soil_ph  
[1] 6.8
```

Now that the value is stored, we can use the object name in further calculations in the same way we would use the number itself.

```
> # Use the object in a calculation  
> soil_ph + 1  
[1] 7.8  
>  
> # Assign the result of a calculation to a new object  
> adjusted_ph <- soil_ph + 0.5  
> adjusted_ph  
[1] 7.3
```

It is worth noting that in R you can also use an equal sign = for assignment, however, the operator <- is a generally accepted standard and a stylistically better choice. This is because = is also used to pass arguments to functions, while <- always means assigning a value to an object, which makes the code more unambiguous and easy to read.

Rules and tips for naming objects

Choosing names for objects is an important part of writing clean and understandable code. There are certain rules and generally accepted conventions:

- Rules (mandatory):
- Object names must begin with a letter.
- They can contain letters, numbers, a period (.) and an underscore (_).
- Names are case-sensitive: Soil_pH and soil_ph are two completely different objects.

Conventions (recommended):

- **Use meaningful names:** `x <- 10` says nothing about the purpose of the object, while `plot_width <- 10` is self-explanatory.
- **Stick to a uniform style:** The most popular are snake_case (words separated by an underscore, such as `soil_organic_carbon`) and camelCase (each new word begins with a capital letter, such as `soilOrganicCarbon`). The snake_case style is very common, especially in the tidyverse

ecosystem, and we will follow it in this guide.

- Avoid naming existing functions: You should not create an object named `c` or `mean`, as this can lead to confusion and errors.

If you want to see a list of all objects in your current environment using code, you can use the `ls()` function.

```
> # List all objects in the current environment
> ls()
[1] "adjusted_ph" "soil_ph"
```

The concept of objects and assignment is fundamental to work in R. Each data analysis that we will carry out will consist of creating, manipulating and analyzing the different objects that store our data at each stage of work.

2.3. Data types and structures (vectors, factors, matrices, data tables, lists)

Until now, we have only worked with single values, assigning them to objects. However, in real analysis, especially in soil science, we almost always deal with data sets: measurement results from dozens of soil profiles, pixel values on a satellite image, coordinates of sampling points. In order to efficiently store and process such data, R uses a variety of **data structures**.

Each object in R has a specific type that determines what kind of information it can contain (e.g., numbers, text, booleans). The structure of the data, in turn, determines how these values are organized. Understanding the basic data structures is absolutely necessary for further work. We consider the most important of them.

Vectors

A vector is the simplest and most fundamental data structure in R. It is an ordered sequence of values, with **all values in a single vector having to be of the same type**. Even the single number we created earlier is actually a vector that is one element long.

To create a vector from several elements, the function `c()` is used (from the English. *combine* or *concatenate*).

There are several main types of atomic vectors:

numeric (numeric): to store real numbers (decimals) and integers. It is the most common type for quantitative measurements.

```
# A numeric vector of soil organic carbon (SOC)
measurements in percent
soc_percent <- c(2.5, 3.1, 1.8, 2.9, 4.2)
soc_percent
[1] 2.5 3.1 1.8 2.9 4.2
```

character (character): to store text data. Text is always enclosed in double (") or single (') quotation marks.

```
# A character vector of soil horizon designations
horizons <- c("Ap", "Bt", "C", "Ap", "Bw")
horizons
[1] "Ap" "Bt" "C"  "Ap" "Bw"
```

logical : to store the values TRUE or FALSE. Often the result of logical checks.

```
# A logical vector indicating the presence of carbonates
(effervescence test)
carbonates_present <- c(FALSE, TRUE, TRUE, FALSE, FALSE)
carbonates_present
[1] FALSE TRUE TRUE FALSE FALSE
```

If you try to mix types in the same vector, R will apply a coercion rule (type casting), converting all elements to the least specific type (usually a symbolic type).

Factors

Factors are a special type of vector designed to store **categorical data**. Outwardly, they may look like symbolic vectors, but internally R stores them as integers, each of which corresponds to a certain "label" or **level**. **This** makes them very effective for statistical modeling and visualization, since R "understands" that these are not just text, but groups.

You can create a factor using the `factor()` function.

```
# A character vector of soil types
soil_types_char <- c("Chernozem", "Podzol", "Luvisol",
"Chernozem", "Chernozem")

# Convert the character vector to a factor
soil_types_factor <- factor(soil_types_char)

soil_types_factor
[1] Chernozem Podzol    Luvisol    Chernozem Chernozem
Levels: Chernozem Luvisol Podzol
```

Pay attention to the output: R not only shows the values, but also lists unique Levels: Chernozem, Luvisol, Podzol.

Matrices

A matrix is a two-dimensional data structure resembling a rectangular table. As in vectors, **all elements of a matrix must be of the same type** (usually numeric). Matrices are often used in linear algebra and geostatistics, for example, to represent covariance matrices.

You can create a matrix using the `matrix()` function.

```
# Create a matrix with 6 elements, arranged into 2 rows
and 3 columns
# representing, for example, nutrient content (N, P, K)
in 2 samples
nutrient_matrix <- matrix(c(1.2, 0.4, 0.8, 1.5, 0.3,
0.9), nrow = 2, ncol = 3)
nutrient_matrix

      [,1] [,2] [,3]
[1,]  1.2  0.8  0.3
[2,]  0.4  1.5  0.9
```

Data Frames

A data table, or **data frame**, is perhaps the most important and widespread data structure to analyze in R. It is a two-dimensional structure, similar to a table in Excel or a database, where rows correspond to observations (such as soil samples) and columns correspond to variables (properties).

The main difference and advantage of a data frame from a matrix is that **columns can have different types of data**. One column can be

numeric (pH), another can be symbolic (sample ID), and the third can be a factor (soil type).

You can create a data frame using the `data.frame()` function by combining several vectors of the same length.

```
# Combine previously created vectors into a soil data
frame
soil_data <- data.frame(
  horizon = horizons,
  soc = soc_percent,
  carbonate = carbonates_present,
  soil_type = soil_types_factor
)

soil_data
```

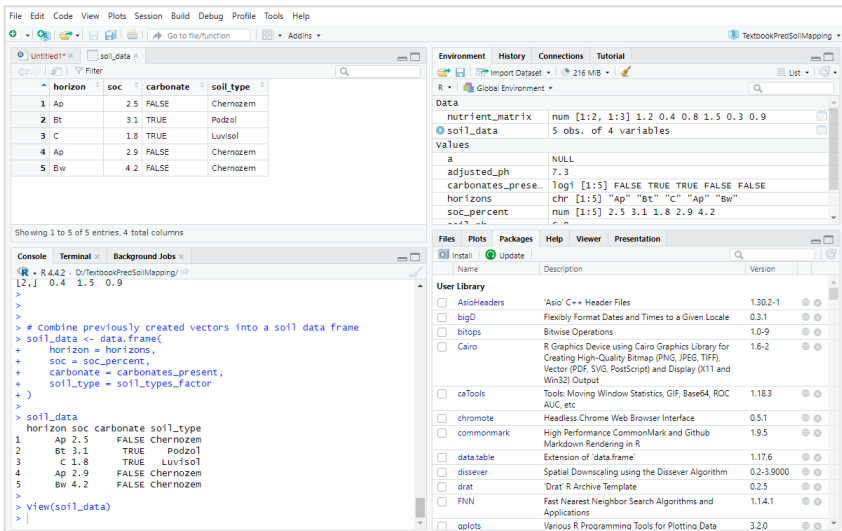


Fig. 1.7. Displaying an object `soil_data` in RStudio. A tabular structure is shown, where each column has its own name and data type, and each row represents a separate observation.

Lists

A list is the most flexible data structure in R. It is an ordered collection of elements, where **each element can be any object R**:

vector, matrix, data frame, other list, and so on. List items do not have to be the same length or type.

Lists are extremely useful for grouping heterogeneous but logically related information. For example, you can store data about a field study in one list: a table with analyzes, a vector with the names of researchers and a numerical value of the average annual precipitation on the site.

```
# Create a list containing various objects related to a
study site
study_site_info <- list(
  site_name = "Polovetsky Steppe",
  location_coords = c(49.5, 32.8),
  soil_properties = soil_data,
  average_rainfall_mm = 550
)

study_site_info
$site_name
[1] "Polovetsky Steppe"

$location_coords
[1] 49.5 32.8

$soil_properties
  horizon soc carbonate soil_type
1      Ap 2.5      FALSE Chernozem
2      Bt 3.1       TRUE  Podzol
3       C 1.8       TRUE  Luvisol
4      Ap 2.9      FALSE Chernozem
5      Bw 4.2      FALSE Chernozem

$average_rainfall_mm
[1] 550
```

Understanding these five basic data structures – vectors, factors, matrices, data tables, and lists – is the foundation on which all subsequent work with data in R is built.

2.4. Data Import and Export

Generating data directly in R, as we did in previous subsections, is useful for training and testing. However, in real research projects, data

almost always comes from external sources. These can be the results of laboratory analyzes uploaded in CSV or Excel format, data from GPS receivers, or geospatial layers prepared in GIS programs. Therefore, the ability to efficiently import data into R for analysis and export results is a fundamental skill.

Import data

The process of loading data from an external file into an R object (usually in a data frame) is called importing. R supports a huge number of data formats thanks to its basic features and specialized packages.

Text Files (CSV)

The most common format for exchanging tabular data is **CSV (Comma-Separated Values)**. It is a simple text file where columns are separated by a comma and rows are separated by a new row.

To read CSV files, the base R has a function `read.csv()`. However, we will use its modern counterpart `read_csv()` from the `readr` package (which is part of `tidyverse`) because it is much faster and smarter in determining column types.

```
# First, ensure the tidyverse package is loaded
library(tidyverse)

# Import soil profile data from a CSV file located in the
# 'data' subfolder
# The result is stored in a data frame (specifically, a
# tibble) called 'soil_profiles'
# Make sure you have a 'data' folder in your project
# directory with this file.
soil_profiles <- read_csv("data/slovakia_soil_profiles.csv")
Rows: 25 Columns: 9
— Column specification
Delimiter: ","
chr (3): SVK-01, Ap, Chernozem
dbl (6): 0, 4.2, 28, 6.8, 17.11, 48.15

i Use `spec()` to retrieve the full column specification
for this data.
i Specify the column types or set `show_col_types =
```

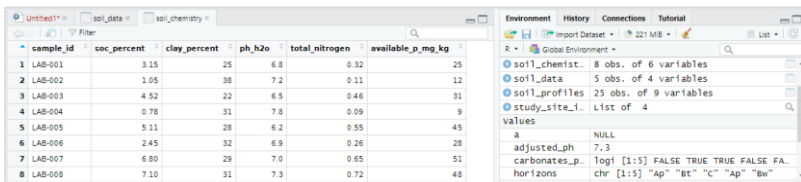
```
FALSE` to quiet this message.
```

Microsoft Excel File (.xlsx)

A lot of data, especially from labs, is stored in Excel files. The `readxl` package is great for reading them. Its `read_excel()` function allows you to easily import data by specifying the path to the file and, if necessary, the name of the sheet or its number.

```
# The readxl package is also part of the core tidyverse
library(readxl)

# Import soil chemical properties from the first sheet of
an Excel file
soil_chemistry <- read_excel("data/lab_results.xlsx",
sheet = 1)
```



The screenshot shows the RStudio interface with a data frame named `soil_chemistry` loaded. The data frame has 8 rows (LAB-001 to LAB-008) and 6 columns: `sample_id`, `soc_percent`, `clay_percent`, `ph_h2o`, `total_nitrogen`, and `available_p_mg_kg`. The right-hand pane shows the Environment tab with a list of loaded objects: `soil_chemst` (8 obs. of 6 variables), `soil_data` (5 obs. of 4 variables), `soil_profiles` (25 obs. of 9 variables), and `study_site_1` (List of 4).

sample_id	soc_percent	clay_percent	ph_h2o	total_nitrogen	available_p_mg_kg
LAB-001	3.15	25	6.8	0.32	25
LAB-002	1.05	38	7.2	0.11	12
LAB-003	4.52	22	6.5	0.46	31
LAB-004	0.78	31	7.8	0.09	9
LAB-005	5.11	28	6.2	0.55	45
LAB-006	2.45	32	6.9	0.26	28
LAB-007	6.80	29	7.0	0.65	51
LAB-008	7.10	31	7.3	0.72	48

Vector geospatial data

To work with spatial data, we will use the `sf` (Simple Features) package. Its `st_read()` function is a universal tool for reading most common vector formats. Preload data from https://geodata.ucdavis.edu/gadm/gadm4.1/shp/gadm41_SVK.shp.zip into the "gis_data" folder.

```
# Load the sf package
library(sf)

# Read a Shapefile of administrative boundaries
# The function reads the .shp file, but automatically
uses associated files (.dbf, .shx, etc.)

slovakia_boundary <- st_read("gis_data/gadm41_SVK_0.shp")
Reading layer `gadm41_SVK_0' from data source
`D:\TextbookPredSoilMapping\gis_data\gadm41_SVK_0.shp'
using driver `ESRI Shapefile'
```

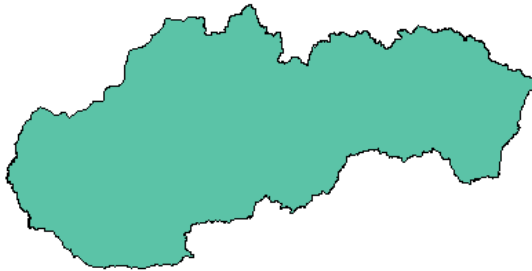
```
Simple feature collection with 1 feature and 2 fields
Geometry type: POLYGON
Dimension:      XY
Bounding box: xmin:16.83446 ymin:47.73275 xmax:22.56791
ymax:49.6138
Geodetic CRS:  WGS 84

# Load dplyr
library(dplyr) # Select COUNTRY

slovakia_boundary <- select(slovakia_boundary, COUNTRY)

plot(slovakia_boundary)
```

COUNTRY



Data export

Once the analysis, processing, or simulation is complete, the results must be saved to a file. This process is called exporting.

Text Files (CSV)

To save a data frame in CSV format, use the `write_csv()` function from the `readr` package. In the base R, there is a function `write.csv()`. If you use it, it is important to specify the argument `row.names = FALSE` to avoid writing an unnecessary column with row numbers.

```
# Create folder "results"
dir.create("results")

# Assume we have a final data frame 'final_soil_data'
```



```
final_soil_data <- soil_chemistry

# Export this data frame to a CSV file
write_csv(final_soil_data, "results/final_soil_data.csv")
```

Microsoft Excel File (.xlsx)

To export to Excel, it is convenient to use the `writexl` package and its function `write_xlsx()`. It allows you to write one or more data frames to different sheets of the same file.

```
library(writexl)

# Export a single data frame to an Excel file
write_xlsx(final_soil_data,
"results/final_soil_data.xlsx")
```

Vector geospatial data

To export spatial objects `sf`, the universal function `st_write()` is used. The choice of file format is determined by the extension you specify in the name. We consider the most popular options.

- **Shapefile (.shp)** It is historically the most common format for vector data interchange, developed by Esri. However, it has significant drawbacks. A shapefile is not a single file, but a set of several files (.shp, .shx, .dbf, .prj, etc.) that must be located in one folder. This often leads to copying errors. In addition, it has strict restrictions: column names in an attribute table cannot exceed 10 characters, which forces the abbreviation of meaningful variable names.

```
# Exporting an sf object to a Shapefile
# Note that long column names in 'predicted_soil_map'
will be truncated

# Create dummies 'predicted_soil_map' from
slovakia_boundary (for example only!!!)

predicted_soil_map <-slovakia_boundary

st_write(predicted_soil_map,
"results/predicted_soil_map.shp")
Writing layer `predicted_soil_map' to data source
`results/predicted_soil_map.shp' using driver `ESRI'
```

```
Shapefile'
Writing 1 features with 1 fields and geometry type
Polygon.
```

- **GeoPackage (.gpkg)** **GeoPackage** is a modern, open-source, standardized file format developed by the Open Geospatial Consortium (OGC). It was created as a versatile, flexible, and efficient replacement for legacy formats like Shapefile.

The advantages of GeoPackage are undeniable:

One file: All data (geometry, attributes, projection information) is stored in a single .gpkg file, making it extremely portable and easy to manage.

- **Flexibility:** There is no limit to the length of column names. A variety of data types are supported.
- **Versatility:** Multiple layers of vector data, raster data, and even tables without geometry can be stored in a single GeoPackage file.
- **Performance:** Due to its architecture, it often performs faster than Shapefile, especially with large datasets.

Because of these advantages, **GeoPackage is the recommended format** for storing and sharing geospatial data in modern projects.

```
# Exporting an sf object to a GeoPackage
# This is the recommended way to save spatial vector data
st_write(predicted_soil_map,
"results/predicted_soil_map.gpkg")
Writing layer `predicted_soil_map` to data source
`results/predicted_soil_map.gpkg` using driver `GPKG`
Writing 1 features with 1 fields and geometry type
Polygon.
```

Choosing the right format for importing and exporting data is the key to efficient and error-free operation, and the use of modern standards like GeoPackage contributes to better reproducibility and compatibility of your research.

Chapter 3. Data manipulation with dplyr

3.1. Introduction to Tidyverse and dplyr

In the previous section, we got acquainted with the fundamental data structures in R. Now we are ready to move on to one of the most important and most frequently performed tasks in data analysis – data **manipulation**. Real data that soil scientists have to work with is rarely perfect. It may contain unnecessary columns, require filtering by certain criteria, require the creation of new variables based on existing ones, or require sorting. A species suitable for analysis and modeling often takes up to 80% of the researcher's total time.

Traditionally, for these tasks, R used the so-called "basic R" – a set of functions that comes with the standard installation. While these tools are powerful, their syntax can often be cumbersome, counterintuitive, and difficult to read, especially for complex chains of operations.

Fortunately, the R ecosystem has undergone a real revolution in recent years thanks to the advent of **tidyverse**. It is not just a package, but a whole philosophy of working with data and a coherent collection of R packages designed for modern data science. All packages in tidyverse share a common philosophy of design, grammar and data structure, which makes the process of working with data extremely logical, consistent and, Most importantly, readable for a person.

The tidyverse philosophy is based on the concept of "**tidy data**". This is a standard for organizing tabular data, which has three simple rules:

- Each variable forms a column.
- Each observation forms a line.
- Each type of observed unit forms a table.

Compliance with this standard greatly simplifies further work, since tidyverse tools are designed specifically to work with such "tidy" data.

The heart of tidyverse for data manipulation is **the dplyr** package. It provides a simple and consistent set of "verbs" – functions that allow you to solve the most common data manipulation tasks. Instead of memorizing hundreds of different functions with obscure names, dplyr offers a small set of tools, each of which performs one distinct action. to dig, a dipstick to take samples, a pH meter to measure acidity. Each

tool has its own clear purpose. Likewise in dplyr, you have verbs for:

- **Select columns** (select()).
- **String filtering** (filter()).
- **Creating** new columns (mutate()).
- **Data sorting** (arrange()).
- **Data aggregation** and summation (group_by() and summarise()).

We look at a simple example. Suppose that we have data on the content of organic carbon (SOC) and clay in different genetic horizons.

```
# Load the tidyverse library
library(tidyverse)

# Create a sample soil data frame (in tidyverse, we often
use 'tibbles')
soil_samples <- tibble(
  profile_id = c("SVK-01", "SVK-01", "SVK-02", "SVK-02",
"SVK-03"),
  horizon = c("Ap", "Bt", "Ap", "BC", "A"),
  soc_percent = c(3.2, 1.1, 4.5, 0.8, 5.1),
  clay_percent = c(25, 38, 22, 31, 28))
```

Suppose we only need to take samples that were taken from the "Ap" horizon and we are only interested in the profile ID and carbon content.

Using dplyr, this query is translated into code almost verbatim:

```
# dplyr approach: filter the rows, then select the
columns
filter(soil_samples, horizon == "Ap")
select(filter(soil_samples, horizon == "Ap"), profile_id,
soc_percent)
# A tibble: 2 × 4
  profile_id horizon soc_percent clay_percent
  <chr> <chr> <dbl> <dbl>
1 SVK-01 AP 3.2 25
2 SVK-02 AP 4.5 22
> select(filter(soil_samples, horizon == "Ap"),
profile_id, soc_percent)
# A tibble: 2 × 2
  profile_id soc_percent
  <chr> <dbl>
1 SVK-01 3.2
```

```
2 SVK-02 4.5
```

The code is clear and consistent. In the following subsections, we will analyze each of these verbs in detail and learn how to combine them into powerful chains of operations using the pipeline operator (`%>%`), which will make our code even more elegant and readable. Learning `dplyr` is an investment that will drastically change your efficiency and approach to working with data in R.

3.2. Basic verbs `dplyr` (`select`, `filter`, `mutate`, `arrange`)

As we noted earlier, `dplyr` provides a small but extremely powerful set of functions, or "verbs," for manipulating data. Each verb is responsible for one specific action, which makes the code intuitive. In this subsection, we'll take a closer look at the four key verbs that form the basis of most data preparation operations: `select()`, `filter()`, `mutate()`, and `arrange()`. To demonstrate their work, we will use an extended soil sample dataset.

```
# Load the tidyverse library first
library(tidyverse)

# An expanded dataset of soil samples for our examples
soil_data <- tibble(
  profile_id = c("SVK-01", "SVK-01", "SVK-02", "SVK-02",
    "SVK-03", "SVK-03"),
  horizon = c("Ap", "Bt", "Ap", "BC", "A", "Bw"),
  depth_cm = c(0, 25, 0, 40, 0, 15),
  soc_percent = c(3.2, 1.1, 4.5, 0.8, 5.1, 2.5),
  clay_percent = c(25, 38, 22, 31, 28, 32),
  ph_h2o = c(6.8, 7.2, 6.5, 7.8, 6.2, 6.9))
```

Select columns with `select()`

Very often, our initial data contains many more variables than is necessary for a specific analysis. The verb `select()` makes it easy to **select the columns** that interest us, or, conversely, to exclude unnecessary ones.

The first argument of the function is always the data table, and the next are the names of the columns that we want to keep. Unlike the basic R, column names do not need to be enclosed in quotation marks, which

makes the code cleaner.

```
# Select three specific columns from the soil_data
select(soil_data, profile_id, horizon, soc_percent)
# A tibble: 6 × 3
  profile_id horizon soc_percent
  <chr>      <chr>      <dbl>
1 SVK-01    Ap          3.2
2 SVK-01    Bt          1.1
3 SVK-02    Ap          4.5
4 SVK-02    BC          0.8
5 SVK-03    A           5.1
6 SVK-03    Bw          2.5
```

Compare this to the equivalent in base R, which is less intuitive:
`soil_data[, c("profile_id", "horizon", "soc_percent")]`

select() also has a set of useful helper functions that allow you to select columns based on their name patterns:

- `starts_with("prefix")`: Selects columns whose names begin with a specific prefix.
- `ends_with("suffix")`: selects columns ending in a suffix.
- `contains("text")`: Selects columns whose names contain specific text.

To exclude a column, it is enough to put a minus sign (-) in front of its name.

```
# Select all columns except for the pH measurement
select(soil_data, -ph_h2o)
# Tibble: 6 × 5
  profile_id horizon depth_cm soc_percent clay_percent
  <chr> <chr> <dbl> <dbl> <dbl>
1 SVK-01 AP 0 3.2 25
2 SVK-01 Bt 25 1.1 38
3 SVK-02 AP 0 4.5 22
4 SVK-02 BC 40 0.8 31
5 SVK-03 A 0 5.1 28
6 SVK-03 Bw 15 2.5 32

# Select all columns that contain the word "percent"
select(soil_data, contains("percent"))
# A tibble: 6 × 2
  soc_percent clay_percent
```

	<dbl>	<dbl>
1	3.2	25
2	1.1	38
3	4.5	22
4	0.8	31
5	5.1	28
6	2.5	32

Filtering Strings with filter()

Perhaps the most common task is **data filtering** – selecting only those strings (observations) that meet certain conditions. The verb `filter()` is intended for this.

The first argument is the data table, and the next arguments are one or more logical conditions. Strings for which the condition is true (TRUE) remain in the result.

```
# Filter for samples from the topsoil (depth_cm == 0)
filter(soil_data, depth_cm == 0)
# A tibble: 3 × 6
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr>         <chr>      <dbl>      <dbl>      <dbl>
<dbl>
1 SVK-01      Ap          0          3.2        25
6.8
2 SVK-02      Ap          0          4.5        22
6.5
3 SVK-03      A           0          5.1        28
6.2

# Filter for samples with high organic carbon content
filter(soil_data, soc_percent > 3.0)
# A tibble: 3 × 6
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr>         <chr>      <dbl>      <dbl>      <dbl>
<dbl>
1 SVK-01      Ap          0          3.2        25
6.8
2 SVK-02      Ap          0          4.5        22
6.5
3 SVK-03      A           0          5.1        28
6.2
```

Note the use of a double equal sign `==` to check for equality.

In basic R, a similar operation looks much more cumbersome due to the constant repetition of the table name:
`soil_data[soil_data$soc_percent > 3.0,]`

`filter()` allows you to easily combine multiple conditions using boolean statements:

- ✓ `&` – logical "AND" (both conditions must be true).
- ✓ `|` – logical "OR" (at least one condition must be true).
- ✓ `!` – logical "NO" (negation).

```
# Filter for topsoil samples with high organic carbon
filter(soil_data, depth_cm == 0 & soc_percent > 4.0)
# Tibble: 2 × 6
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr> <chr> <dbl> <dbl> <dbl> <dbl>
1 SVK-02 AP 0 4.5 22 6.5
2 SVK-03 A 0 5.1 28 6.2
```

Creating new variables with `mutate()`

The verb `mutate()` allows you to **create new columns** based on existing ones or modify existing ones. This is an extremely powerful tool for feature engineering.

The syntax is simple: after the table name, you write `new_column_name = expression`.

In soil science, it is common to convert organic carbon content (SOC) to organic matter content (SOM) using the Van Bemmelen ratio (~ 1.724). We do it:

```
# Create a new column for soil organic matter (SOM)
mutate(soil_data, som_percent = soc_percent * 1.724)
# A tibble: 6 × 7
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o som_percent
  <chr> <chr> <dbl> <dbl> <dbl> <dbl> <dbl>
1 SVK-01 AP 0 3.2 25 6.8 5.52
2 SVK-01 Bt 25 1.1 38 7.2 1.90
3 SVK-02 AP 0 4.5 22 6.5 7.76
4 SVK-02 BC 40 0.8 31 7.8 1.38
```



```
5 SVK-03 A 0 5.1 28 6.2 8.79
6 SVK-03 Bw 15 2.5 32 6.9 4.31
```

The advantage of `mutate()` over the basic approach (`soil_data$som_percent <- soil_data$soc_percent * 1.724`) is that you can create multiple columns at a time, and even reference newly created columns in the same command.

```
# Create SOM column and then immediately create a C:Clay
ratio column
mutate(soil_data,
       som_percent = soc_percent * 1.724,
       c_clay_ratio = soc_percent / clay_percent
)
# A tibble: 6 × 8
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o som_percent
  <chr>      <chr>      <dbl>      <dbl>      <dbl>
<dbl>      <dbl>
1 SVK-01    Ap          0          3.2         25
6.8         5.52
2 SVK-01    Bt         25          1.1         38
7.2         1.90
3 SVK-02    Ap          0          4.5         22
6.5         7.76
4 SVK-02    BC         40          0.8         31
7.8         1.38
5 SVK-03    A          0          5.1         28
6.2         8.79
6 SVK-03    Bw         15          2.5         32
6.9         4.31
# i 1 more variable: c_clay_ratio <dbl>
```

Sorting data with `arrange()`

The last of the base verbs, `arrange()`, is responsible for **sorting the rows of** the table by the values of one or more columns.

```
# Arrange the data from lowest to highest SOC content
arrange(soil_data, soc_percent)
# A tibble: 6 × 6
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr> <chr> <dbl> <dbl> <dbl> <dbl>
1 SVK-02 BC 40 0.8 31 7.8
```

```

2 SVK-01 Bt 25 1.1 38 7.2
3 SVK-03 Bw 15 2.5 32 6.9
4 SVK-01 AP 0 3.2 25 6.8
5 SVK-02 AP 0 4.5 22 6.5
6 SVK-03 A 0 5.1 28 6.2

```

By default, `arrange()` sorts in ascending order. To change the descending order, you need to wrap the column name in the `desc()` function.

```

# Arrange the data from highest to lowest SOC content
arrange(soil_data, desc(soc_percent))
# A tibble: 6 × 6
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr> <chr> <dbl> <dbl> <dbl> <dbl>
1 SVK-03 A 0 5.1 28 6.2
2 SVK-02 AP 0 4.5 22 6.5
3 SVK-01 AP 0 3.2 25 6.8
4 SVK-03 Bw 15 2.5 32 6.9
5 SVK-01 Bt 25 1.1 38 7.2
6 SVK-02 BC 40 0.8 31 7.8

```

Again, compare this to the basic R syntax, which requires the use of the `order()` function and is less obvious:
`soil_data[order(soil_data$soc_percent, decreasing = TRUE),]`

You can sort by multiple columns. R will first sort by the first column and then, within the same values of the first column, sort by the second.

```

# Arrange by profile ID, and then by depth within each
profile
arrange(soil_data, profile_id, depth_cm)
# A tibble: 6 × 6
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr> <chr> <dbl> <dbl> <dbl> <dbl>
1 SVK-01 AP 0 3.2 25 6.8
2 SVK-01 Bt 25 1.1 38 7.2
3 SVK-02 AP 0 4.5 22 6.5
4 SVK-02 BC 40 0.8 31 7.8
5 SVK-03 A 0 5.1 28 6.2
6 SVK-03 Bw 15 2.5 32 6.9

```

These four verbs – `select`, `filter`, `mutate`, `arrange` – are dplyr's workhorses. Once you have mastered them, you will be able to complete the vast majority of data preparation and cleaning tasks, making your code not only efficient, but also extremely clear and easy to read.

3.3. Strategy "Divide-Apply-Unite" (`group_by`, `summarise`)

The previous four verbs – `select`, `filter`, `mutate`, and `arrange` – are extremely useful for working with data at the level of individual rows and columns. However, the real magic of data analysis often happens at the aggregate level, where we need to calculate the totals for different groups within our dataset. For example, in soil science, we are rarely interested in the pH of one particular sample; much more often we want to know the average pH for each genetic horizon, or the maximum organic carbon content within each soil profile.

To solve such problems, dplyr implements a powerful strategy known as "**Split-Apply-Combine**". This concept, popularized by Hadley Wickham, consists of three steps:

Split: A dataset is broken down into smaller groups based on the values of one or more categorical variables.

- **Apply:** A certain function is applied to each group independently (for example, calculating the average, sum, quantity).
- **Combine:** The results obtained from each group are collected together into a single summary table.

This strategy is implemented in dplyr using two key verbs: `group_by()` and `summarise()`. They are almost always used together and are one of the most powerful tools in your arsenal.

Grouping data using `group_by()`

The verb `group_by()` itself does not change the appearance of your data. It adds metadata to the table, telling R that all subsequent operations should not be performed for the entire table at once, but for each group separately. Suppose that we want to calculate the averages for each soil profile in our set of `soil_data`. The first step is to group the data by profile ID.

```
# Load the tidyverse library and use the same data as
before
library(tidyverse)

soil_data <- tibble(
  profile_id = c("SVK-01", "SVK-01", "SVK-02", "SVK-02",
"SVK-03", "SVK-03"),
  horizon = c("Ap", "Bt", "Ap", "BC", "A", "Bw"),
  depth_cm = c(0, 25, 0, 40, 0, 15),
  soc_percent = c(3.2, 1.1, 4.5, 0.8, 5.1, 2.5),
  clay_percent = c(25, 38, 22, 31, 28, 32),
  ph_h2o = c(6.8, 7.2, 6.5, 7.8, 6.2, 6.9)
)

# Group the data by profile_id
grouped_by_profile <- group_by(soil_data, profile_id)
grouped_by_profile
# A tibble: 6 × 6
# Groups:   profile_id [3]
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o
  <chr>         <chr>      <dbl>      <dbl>      <dbl>
<dbl>
1 SVK-01      Ap          0          3.2         25
6.8
2 SVK-01      Bt         25          1.1         38
7.2
3 SVK-02      Ap          0          4.5         22
6.5
4 SVK-02      BC         40          0.8         31
7.8
5 SVK-03      A           0          5.1         28
6.2
6 SVK-03      Bw         15          2.5         32
6.9
```

If you display `grouped_by_profile` on the screen, you will see that the data looks the same, but the inscription appears on top: A tibble: 6 × 6 [Groups: profile_id [3]]. This means that R is now "aware" of the existence of three groups (SVK-01, SVK-02, SVK-03).

Data Aggregation with summarise()

Once the data is grouped, the verb `summarise()` (or its American variant `summarize()`) allows you to "collapse" each group into one line

by calculating the final statistical indicators for it.

Inside `summarise()`, you create new columns by assigning them the results of aggregating functions such as `mean()` (mean), `sd()` (standard deviation), `min()` (minimum), `max()` (maximum), `median()` (median), and `n()` (number of observations in a group).

```
# Calculate summary statistics for each profile
summarise(grouped_by_profile,
           avg_soc = mean(soc_percent),
           max_clay = max(clay_percent),
           num_horizons = n()
)
# A tibble: 3 × 4
  profile_id avg_soc max_clay num_horizons
  <chr> <dbl> <dbl> <int>
1 SVK-01 2.15 38 2
2 SVK-02 2.65 31 2
3 SVK-03 3.8 32 2
```

The result will be a new, much smaller table where each row represents a single soil profile and its generalized characteristics. This is the end result of the "Divide-Apply-Unite" strategy.

Compare this to basic R, where to achieve the same result, you would have to use a cumbersome `aggregate()` function or a combination of `split()` and `lapply()` functions, which is significantly less readable:

```
aggregate(soil_data[, c("soc_percent", "clay_percent")],
          by = list(profile_id = soil_data$profile_id), FUN = mean)
```

You can group data by multiple variables at once. For example, if we had data from different regions, we could group them first by region and then by soil type within each region. `summarise()` will then create a summary string for each unique combination of these variables.

The combination of `group_by()` and `summarise()` is the cornerstone of exploratory data analysis. It allows for a quick transition from raw, detailed data to meaningful, aggregated insights, which is absolutely essential for understanding patterns in soil properties and preparing data for further predictive modeling.

3.4. Pipeline operator (%>%)

So far, we have used dplyr verbs one at a time, creating intermediate objects or nesting function calls inside each other. For example, if we needed to filter the data and then group it and summarize it, we could write like this:

```
# Nested approach: hard to read from inside out
summarise(group_by(filter(soil_data, depth_cm > 0),
  profile_id), avg_soc = mean(soc_percent))
# Tibble: 3 × 2
  profile_id avg_soc
  <chr>    <dbl>
1 SVK-01  1.1
2 SVK-02  0.8
3 SVK-03  2.5
```

Such code is functional, but it is very difficult to read. To understand what is happening, you need to start with the deepest nested function (filter) and move outward. This is contrary to the natural way of thinking, where we imagine a sequence of actions.

Another approach is to create intermediate variables at each step:

```
# Intermediate variables approach: verbose and clutters
the environment
filtered_data <-filter(soil_data, depth_cm > 0)
grouped_data <-group_by(filtered_data, profile_id)
summary_data <-summarise (grouped_data, avg_soc =
mean(soc_percent))

summary_data
# Tibble: 3 × 2
  profile_id avg_soc
  <chr>    <dbl>
1 SVK-01  1.1
2 SVK-02  0.8
3 SVK-03  2.5
```

This option is much more readable, but it litters your work environment with objects that you may never need again.

Fortunately, there is a much more elegant solution, which is one of

the hallmarks of tidyverse – **pipe operator %>%**. This operator, which comes from the magrittr package and is an integral part of dplyr, allows you to "pass" the result of one function to the input of the next, creating logical and readable chains of operations.

The %>% operator can be read as "and then". It takes the object to the left of it and passes it as the **first argument** to the function on the right. That is, the expression `x %>% f(y)` is equivalent to `f(x, y)`.

We rewrite our previous example using the pipeline operator:

```
# The pipe approach: intuitive, readable, and efficient
soil_data %>%
  filter(depth_cm > 0) %>%
  group_by(profile_id) %>%
  summarise(avg_soc = mean(soc_percent))
# A tibble: 3 × 2
  profile_id avg_soc
  <chr>      <dbl>
1 SVK-01      1.1
2 SVK-02      0.8
3 SVK-03      2.5
```

Now the code reads as a sentence in English: "Take `soil_data`, **and then** filter the lines where `depth_cm` greater than 0, **and then** group by `profile_id`, **and then** sum by calculating `avg_soc`." The order of the code corresponds to the order of operations, which makes the logic of the analysis crystal clear. For ease of reading, it is customary to break long chains into separate lines after each %>% operator.

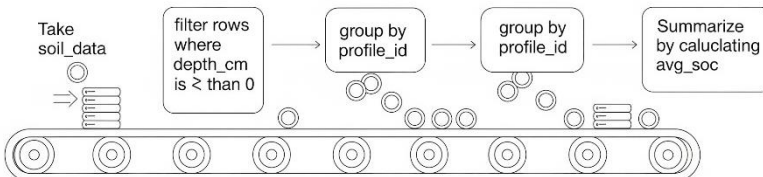


Fig. 1.8. Schematic representation of the work of the conveyor operator. Data "flows" from left to right through a sequence of functions, where the result of each step becomes the input for the next

Consider a more complex but realistic example from soil science.

Suppose we need to perform the following task: *For a dataset, soil_data calculate the average pH and the range of clay content (the difference between maximum and minimum) for each genetic horizon.*

Without a pipeline operator, this request would turn into a cumbersome and confusing design. With it, the solution looks like a clear recipe:

```
# A complex data manipulation task solved elegantly with
the pipe
soil_data %>%
  filter(soc_percent > 1.0) %>%
  group_by(horizon) %>%
  summarise(
    avg_ph = mean(ph_h2o),
    clay_range = max(clay_percent) - min(clay_percent),
    sample_count = n()
  ) %>%
  arrange(desc(avg_ph))

# A tibble: 4 × 4
  horizon avg_ph clay_range sample_count
  <chr>   <dbl>   <dbl>   <int>
1 Bt      7.2      0         1
2 Bw  6.9 0 1
3 Ap      6.65    3         2
4 A  6.2 0 1
```

The `%>%` operator is not just syntactic sugar. It is a tool that changes the way we think about data manipulation, encouraging the construction of consistent, logical and reproducible workflows. Once you get used to it, you are unlikely to want to go back to nested functions or intermediate variables.

Note: Starting with version 4.1.0, R has its own, "native" pipeline operator `|>`. It performs a similar function, but the `%>%` operator with `magrittr` remains the standard in the tidyverse ecosystem and has some additional features, so we will use it in this guide.

Chapter 4. Data visualization with ggplot2

4.1. Grammar of graphics

Once we have learned how to organize, filter, and aggregate our data, the next logical step is **to visualize** it. Graphs and charts are the most powerful tool for exploratory data analysis (EDA). The human brain is much better at perceiving visual patterns, outliers, and relationships than bare numbers in a table. Other. In the context of soil science, visualization helps to understand the distribution of soil properties, to identify relationships between, for example, organic matter content and depth, or to compare the characteristics of different types of soils.

There are several systems for creating graphs in R. The basic graphics system (`plot()`, `hist()`, `boxplot()`) functions is powerful, but it works according to the "easel and brush" model: you create a basic graph and then sequentially add elements (points, lines, legends) to it with separate commands. This approach can be flexible, but for complex graphs, the code becomes cumbersome and its logic is not obvious.

In contrast, **the ggplot2 package**, which is the cornerstone of the tidyverse ecosystem, offers a completely different, much more powerful philosophy. It is based on the concept of **"Grammar of Graphics"**, first described in the book by Leland Wilkinson. This grammar treats any graph not as a unique creation, but as a combination of independent components. Just as the grammar of a language allows us to construct an infinite number of meaningful sentences from a limited set of words and rules, the grammar of graphics allows you to create an infinite number of different visualizations from a limited set of components.

The main components of this grammar are:

Data: The dataset you want to visualize. For ggplot2, it should always be a data table (data frame or tibble) in a "tidy" format.

Aesthetics: This is how variables from your data **are mapped** to the visual properties of the graph. Aesthetics are specified inside the `aes()` function. The most common aesthetics are x and y (position on the axes), but there are also color, fill, shape, size, alpha, and others.

Geometries: These are geometric objects that directly represent data on a graph. They are added as layers using functions starting with `geom_`,

such as `geom_point()` to create a scatter plot, `geom_bar()` for a bar chart, `geom_boxplot()` for a "whisker box". This is **how you see your data**.

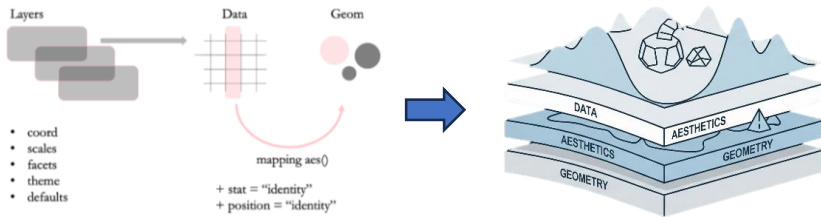


Fig. 1.9. Conceptual scheme of the Grammar of graphics. Shows how data, aesthetic mappings, and geometric objects are combined into layers to create the final graph

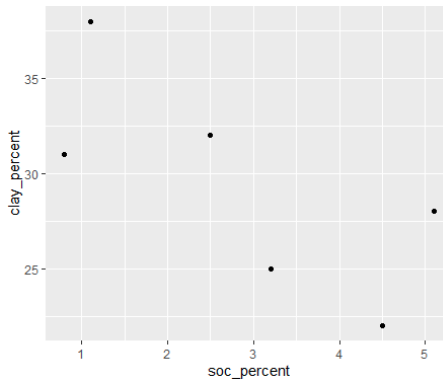
Plotting in `ggplot2` is the process of adding layers. You start with the `ggplot()` function, where you specify a dataset and basic aesthetic mappings. Then, using the `+` sign, you add one or more geometric layers.

Now we look at this with our example with soil data. Let us create a scatter plot to investigate the relationship between organic carbon content (`soc_percent`) and clay content (`clay_percent`).

```
# Load the tidyverse library which includes ggplot2
library(tidyverse)

# Use the same soil_data from the previous chapter
soil_data <- tibble(
  profile_id = c("SVK-01", "SVK-01", "SVK-02", "SVK-02",
    "SVK-03", "SVK-03"),
  horizon = c("Ap", "Bt", "Ap", "BC", "A", "Bw"),
  depth_cm = c(0, 25, 0, 40, 0, 15),
  soc_percent = c(3.2, 1.1, 4.5, 0.8, 5.1, 2.5),
  clay_percent = c(25, 38, 22, 31, 28, 32),
  ph_h2o = c(6.8, 7.2, 6.5, 7.8, 6.2, 6.9)
)

# Create a scatter plot
ggplot(data = soil_data, mapping = aes(x = soc_percent, y =
  clay_percent)) +
  geom_point()
```

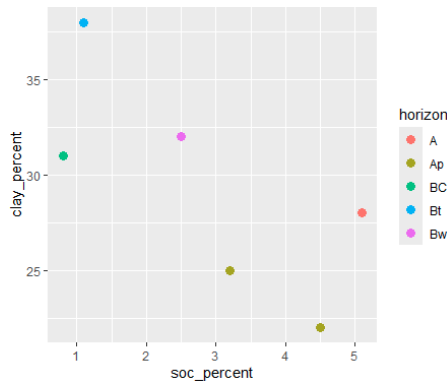


Now we analyze this code:

- `ggplot(data = soil_data, ...)`: We initialize the graph by indicating that we will use table `soil_data`.
- `mapping = aes(x = soc_percent, y = clay_percent)`: we create **an aesthetic mapping**. We say `ggplot2`: "Take column `soc_percent` and map it on the x-axis. Take column `clay_percent` and plot it on the y-axis."
- `+ geom_point()`: we add **a geometric layer**. We say: "Represent these mappings as **points**."

The power of grammar lies in how easily a graph can be modified and expanded. What if we want to see if this relationship is different for different genetic horizons? We just have to add another aesthetic reflection: `color = horizon`.

```
# Add a third variable (horizon) mapped to the color
aesthetic
ggplot(data = soil_data, mapping = aes(x = soc_percent, y
= clay_percent, color = horizon)) +
  geom_point(size = 3) # Make points a bit larger for
better visibility
```



We only added one argument `color = horizon` inside `aes()`. `ggplot2` automatically did the rest: assigned a unique color to each horizon, colored the dots accordingly, and created a legend. This is radically different from the basic R approach, where to achieve the same result, you would have to manually create a color picker, draw subsets of the data with different colors in a loop, and then manually add the legend.

This layer- and grammar-based approach makes `ggplot2` an extremely powerful and flexible tool. Having mastered its basic principles, you can create both simple exploration graphs and complex, ready-to-publish visualizations with minimal effort.

4.2. Creation of basic graphs for exploratory data analysis (histograms, scatter plots, box plots)

Having mastered the theoretical foundations of the "Grammar of Graphics", we can move on to creating the most common types of graphs, which are the workhorses of any exploratory data analysis. Each type of graph is designed to answer a specific question about your data. We will look at three main types: histograms (to study the distribution of a single continuous variable), scatter plots (to investigate the relationship between two continuous variables), and box diagrams (to compare the distribution of a continuous variable between different categories).

Histograms: the study of the distribution

Question: How are the values of a certain soil property distributed?

Is the distribution symmetrical? Are there emissions?

A histogram is the best tool for visualizing **the distribution of one continuous (quantitative) variable**. It breaks down the range of values of a variable into a series of intervals (or "bins", bins) of the same width and shows how many observations fall into each of them.

To create a histogram in ggplot2, `geom_histogram()` is used. The main aesthetics to specify is `x`, that is, the variable whose distribution we want to see.

We explore the pH distribution in our dataset.

```
# Plotting the distribution of soil pH
ggplot(data = soil_data, mapping = aes(x = ph_h2o)) +
  geom_histogram(binwidth = 0.25, color = "black", fill =
"lightblue")
```

Graph analysis: From this histogram, we can see that most of our samples have a pH in the range of 6.2 to 7.2, with a peak of about 6.8-7.0. The distribution looks a bit asymmetrical. The `binwidth` argument controls the width of the "baskets" and is very important; it is worth experimenting with it to find the optimal representation. `color` specifies the color of the lines around the columns, and `fill` specifies the color of their fill. that these arguments are specified *outside of* `aes()`, since we are specifying a fixed color rather than mapping some variable from the data to it.

Compare this to the base R: `hist(soil_data$ph_h2o)`. Although the result is easy to obtain, further customization of the appearance (colors, captions) requires much more additional arguments.

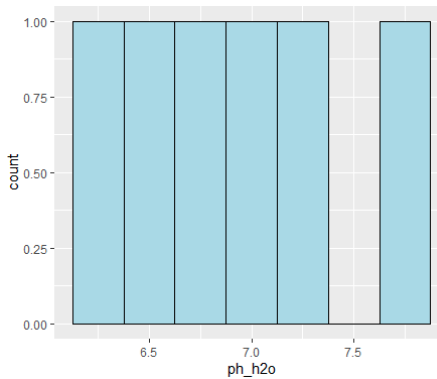


Fig. 1.10. Histogram of the distribution of pH_h2o values. Along the X axis, the pH value is deferred, along the Y axis - the number of samples that fell into the corresponding interval

Scatterplots: A Study of Relationships

Question: Is there a relationship between the two quantitative properties of soil? For example, is the organic carbon content related to the clay content?

A scatter plot is a standard for visualizing the **relationship between two continuous variables**. Each observation is represented by a point on a two-dimensional plane. `geom_point()` is used to create it, and in `aes()` it is necessary to specify variables for the x and y axes.

```
# Creating a scatter plot to investigate the relationship
# between SOC and clay
ggplot(data = soil_data, mapping = aes(x = soc_percent, y
= clay_percent)) +
  geom_point(size = 3, alpha = 0.8)
```

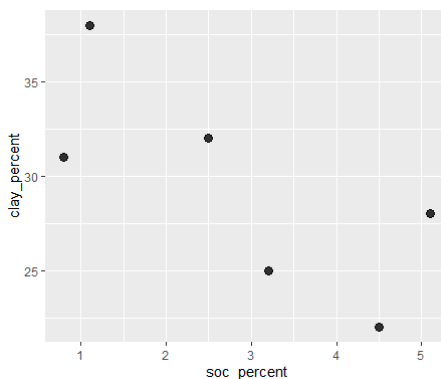


Fig. 1.11. Scatter plot showing the relationship between organic carbon content (soc_percent) and clay content (clay_percent)

Graph analysis: It is difficult to draw a definitive conclusion based on these few points, but there does not seem to be a clear linear relationship. However, we can see that the sample with the highest SOC content has a relatively low clay content. Scatterplots are indispensable for detecting linear and nonlinear trends, clusters, and outliers.

Box Charts: Distribution Comparison

Question: How does the distribution of the quantitative variable differ for different groups (categories)? For example, does the organic carbon content of different genetic horizons differ?

A box plot, or "box plot", is an ideal tool for **comparing the distribution of a continuous variable between several groups** defined by a categorical variable. It compactly displays key statistical indicators: median (center line), interquartile span (box height, IQR), as well as "whiskers" showing the range of typical values, and individual points for potential outliers.

To create it, `geom_boxplot()` is used. In `aes()`, we map a categorical variable to the x-axis, and a continuous variable to the y-axis.

```
# Comparing the distribution of SOC across different soil
horizons
ggplot(data = soil_data, mapping = aes(x = horizon, y =
soc_percent)) +
  geom_boxplot()
```

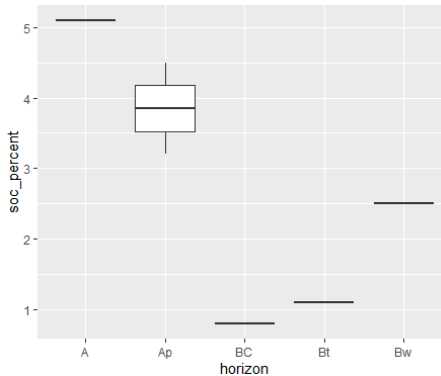


Fig. 1.12. Box diagrams comparing the distribution of organic carbon content (soc_percent) for different genetic horizons (horizon)

Graph analysis: This graph shows us a lot of information at once. For example, the horizons 'Ap' and 'A' have a significantly higher and more variable SOC content compared to the lower horizons 'Bt', 'BC' and 'Bw'. The median value for the horizon 'A' is the highest. This type of visualization is extremely effective for comparing groups, which is a constant task in soil science.

These three types of graphs – histogram, scatter plot, and box plot – form the basis of exploratory analysis. ggplot2 allows you to create them using a simple and logical syntax, allowing you to quickly test hypotheses and gain a deep understanding of the structure of your data.

4.3. Refining plots and figure design

The graphs we created in the previous unit are great for quick exploratory analysis. They allow us to see the structure of the data, but they lack context and a professional appearance to be included in a report, presentation or scientific article. Axis names generated automatically from column names may be unclear (e.g. soc_percent), graphics may lack a title, and the standard gray theme ggplot2 is not always the best choice.

The strength of "Graph Grammar" is that we can consistently add new layers to improve and customize almost every element of our graph.

These layers do not change the main view of the data (geom), but only modify its appearance and add annotations. We consider the most important tools for "polishing" our visualizations.

Adding axis headers and captions (labs())

The very first step to improving your schedule is to give it clear captions. The `labs()` function allows you to manage all text labels on a graph:

- `title`: Main title.
- `subtitle`: subtitle for more details.
- `x, y`: Captions for the X and Y axes.
- `color, fill, shape, etc.`: name for the corresponding legend.
- `caption`: the data source or notes at the bottom of the graph.

Now we take our scatter plot and make it more informative.

```
# Start with the basic scatter plot
p1 <- ggplot(data = soil_data, mapping = aes(x =
soc_percent, y = clay_percent, color = horizon)) +
geom_point(size = 3)

# Now, add informative labels
p1 + labs(
title = "Relationship between organic carbon and clay
content",
subtitle = "Data on soil profiles in Slovakia",
x = "Organic carbon content, %",
y = "Clay content, %",
color = "Genetic\nhorizon",
caption = "Source: fictitious data for example"
)
```

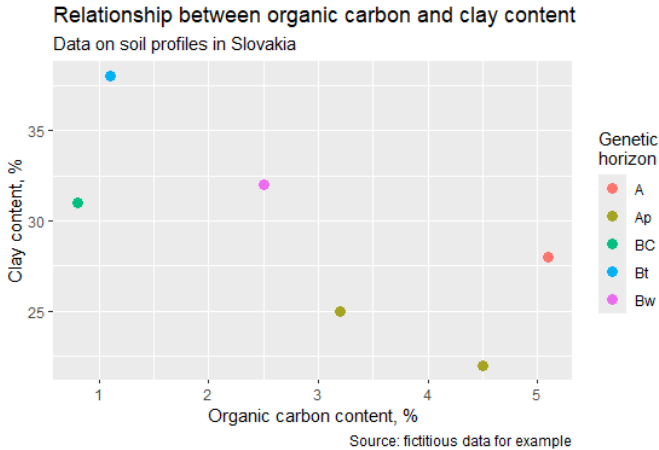


Fig. 1.13. Improved scatter plot with informative titles, axis labels, and a redesigned legend name

Changing the theme of the graph (`theme()`)

The graphic theme controls all non-data elements: background color, gridlines, font and text size, legend position, etc. `ggplot2` has several built-in "complete" themes that allow you to radically change the appearance of the graph with one command. The most popular of them are:

- `theme_bw()`: A theme with a white background and gray grid lines (black and white).
- `theme_classic()`: a minimalist theme reminiscent of graphs from scientific publications, only with X and Y axes, without background and grid.
- `theme_minimal()`: A theme without background colors.
- `theme_void()`: removes absolutely everything, leaving only the geomes themselves (useful for maps).

```
# Apply the classic theme to our plot
p1 + labs(
  title = "Relationship between organic carbon and clay
content",
  x = "Organic carbon content, %",
  y = "Clay content, %",
  color = "Horizon"
) +
```

```
theme_classic()
```

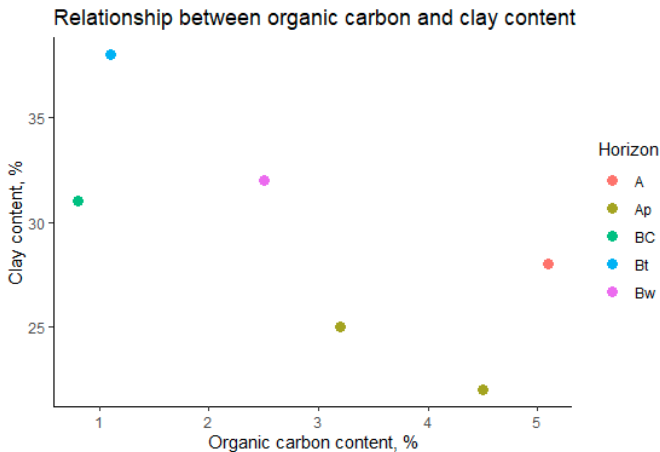


Fig. 1.14. Scatter plot with applied `theme_classic()`

In addition to pre-made themes, you can fine-tune individual elements using the `theme()` function. For example, move the legend down.

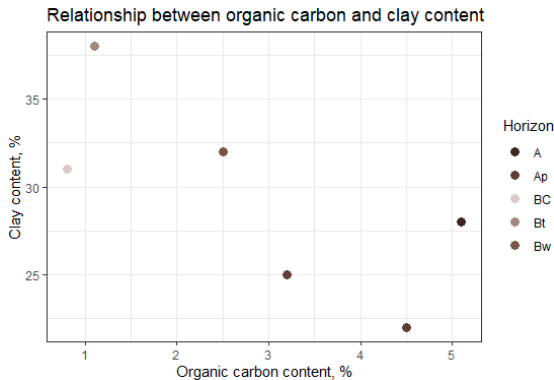
Manage scales (`scale_*()`)

The `scale_*()` functions control exactly how data is displayed in aesthetics. They allow you to customize colors, shapes, sizes, as well as graph axes. The name of the function consists of three parts: `scale_`, the name of the aesthetic (color, fill, x) and the name of the scale type (manual, gradient, continuous).

For example, ggplot2's standard color palette is well-matched, but sometimes we want to set custom colors that have a certain semantic load (e.g. darker colors for deeper horizons). For this, `scale_color_manual()` is used.

```
# Manually setting colors for each horizon
p1 + labs(
  title = "Relationship between organic carbon and clay
  content",
  x = "Organic carbon content, %",
  y = "Clay content, %",
```

```
color = "Horizon"
) +
theme_bw() +
scale_color_manual(values = c("Ap" = "#5D4037", "Bt" =
"#A1887F", "BC" = "#D7CCC8", "A" = "#3E2723", "Bw" =
"#795548"))
```



Faceting: Creating subgraphs (facet_wrap())

What if we want to compare patterns for different subsets of data, but plotting them on a single graph using a color or shape makes it overloaded? ggplot2 offers an extremely powerful tool – **faceting**. It allows you to split a single graph into a grid of several smaller subgraphs (facets), where each subgraph shows data for a specific subgroup.

To do this, the function `facet_wrap()` is used. Its main argument is a formula starting with `~`, followed by the name of the categorical variable by which the data is to be split.

Now we need to create a separate box diagram of the pH distribution for each soil profile.

```
# Create faceted boxplots
ggplot(data = soil_data, mapping = aes(x = horizon, y =
ph_h2o)) +
geom_boxplot() +
facet_wrap(~ profile_id) +
labs(
title = "Distribution of pH by horizons within each
profile",
x = "Genetic horizon",
```

```
y = "pH (H2O) "
) +
theme_bw()
```

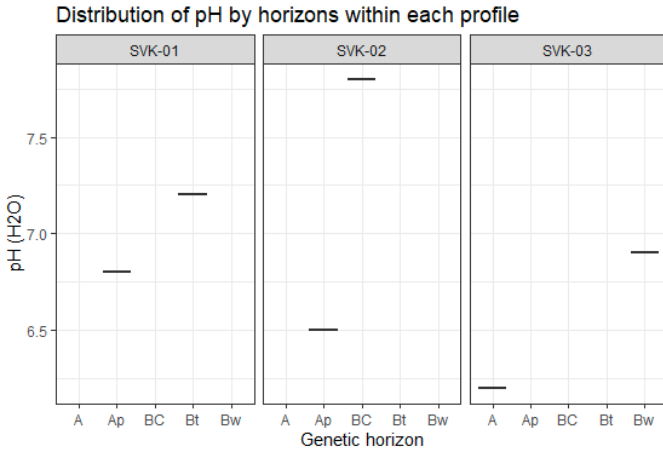


Fig. 1.15. Faceted graph showing box diagrams of the pH distribution for each profile_id separately

Faceting is one of the most effective ways to visualize complex, multidimensional data, allowing you to easily compare patterns between groups. By combining geomes, aesthetics, labels, themes, and facets, you can transform a simple exploration graph into a rich, publication-ready visualization with complete control over every aspect of its appearance. And most importantly, this whole process is fully reproducible and recorded in your R script.

Chapter 5. Working with spatial data in R

5.1. Modern spatial packages: **sf** and **terra**

So far, we have worked mainly with tabular, or "aspatial", data. However, digital soil science is inherently a geospatial discipline. Our data – whether sampling points, soil contour polygons, or raster surfaces of environmental factors – is always geographically referenced. Therefore, being able to process, analyze, and visualize spatial data efficiently is absolutely essential (Lovelace et al., 2019).

For a long time, the R ecosystem for working with spatial data has been somewhat fragmented. For vector data, the **sp** package was used, and for interaction with the GDAL, GEOS, and PROJ geospatial libraries, the **rgdal** and **rgeos** packages were used. For raster data, the standard was the **raster** package. While these tools have been and continue to be powerful, their syntax and data structures have not always been intuitive and have not integrated well with the modern tidyverse ecosystem.

Fortunately, there have been significant advances in recent years, and today we have two modern, fast and coordinated packages that have become the new standard for geospatial analysis in R:

sf (Simple Features) – for working with **vector data** (points, lines, polygons).

terra – for working with **raster data** (grids, images).

These two packages are designed to seamlessly integrate with each other and with tidyverse, creating an extremely powerful and logical environment for solving the challenges of digital soil science.

Vector data from **sf**

The **sf** package (Pebesma, 2018) implements the "Simple Features" standard from the Open Geospatial Consortium (OGC), a generally accepted way of representing vector geodata. The main innovation of **sf** is how it stores data. An **sf** object is essentially a regular **data.frame** (or **tibble**) that has one special column, usually named **geometry**. This column is a "list-column", where each element is the geometry of the corresponding row (for example, a point, line, or polygon object).



Fig. 1.16. Conceptual representation of the object `sf`. (c) 2018 by [Allison Horst](#))

This approach is revolutionary because it means that **all the dplyr verbs we have learned work "out of the box" with `sf` objects!** You can filter, select, mutate, and group spatial data as easily as you would with regular tables.

We read a vector file with soil profile points using the basic function of the package – `st_read()`. Most functions in `sf` start with the prefix `st_` (from *spatial type*).

```
# Load the sf package
library(sf)

# Read a GeoPackage file containing soil sample locations
# This file should be in a 'gis_data' subfolder of your
# project
soil_points <-
st_read("gis_data/slovakia_soil_points.gpkg")

# Look at the structure of the sf object
print(soil_points)Simple feature collection with 26
features and 7 fields
```

```

Geometry type: POINT
Dimension:      XY
Bounding box: 17.11 Edge: 47.88 xmax: 20.25 ymax: 49.22
Geodetic CRS:  WGS 84
First 10 features:
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o    WRB          geom
1         SVK-01      Ap         0           4.2          28
6.8 Chernozem POINT (17.11 48.15)
2         SVK-01      A          25          2.1          29
7.1 Chernozem POINT (17.11 48.15)
3         SVK-01      C          60          0.5          25
8.2 Chernozem POINT (17.11 48.15)
4         SVK-02      Ap         0           3.8          32
6.5 Phaeozem POINT (17.55 48.05)
5         SVK-02      A          22          1.9          33
6.8 Phaeozem POINT (17.55 48.05)
6         SVK-02      C          70          0.4          30
7.9 Phaeozem POINT (17.55 48.05)
7         SVK-03      Ap         0           2.5          22
6.2 Luvisol POINT (18.23 48.34)
8 SVK-03 and 20 0.8 15 5.8 Luvisol POINT (18.23 48.34)
9 SVK-03 Bt 50 0.6 35 6.1 Luvisol POINT (18.23 48.34)
10 SVK-04 A 0 3.1 18 5.9 Cambisol POINT (19.04 48.73)

```

The output shows us that `soil_points` is a Simple feature collection with 26 features and 7 fields. We see information about the coordinate system (CRS) and a regular table, but with an additional geometry column of type POINT.

Raster data from terra

The `terra` package is the modern successor to the `raster` package. Written from scratch in C++ by the `raster` author himself, it offers significantly higher performance, especially when working with large files, and a simpler and more consistent syntax.

Raster data in soil science are, as a rule, our covariates for modeling: digital elevation model (DEM), derivatives from it (slope, exposure), satellite indices (NDVI), etc.

The main function for reading raster data in `terra` is `rast()`. We need to download DMR3.5 (we will use DMR3.5 for the entire territory of Slovakia with a resolution of 100 m).


```
# Load the terra package
library(terra)

# Read a GeoTIFF file representing a Digital Elevation
Model (DEM)
# This file should be in a 'data' subfolder
dem <- rast("gis_data/dmr3_5_100.tif")

# Look at the structure of the SpatRaster object
print(dem)
class      : SpatRaster
dimensions : 3523, 6649, 1  (nrow, ncol, nlyr)
resolution : 100, 100  (x, y)
extent     : 1853712, 2518612, 6045427, 6397727  (xmin,
xmax, ymin, ymax)
coord. ref.: WGS 84 / Pseudo-Mercator (EPSG:3857)
source     : dmr3_5_100.tif
name       : dmr3_5_100
```

The output shows the key metadata of the raster: the number of layers, dimensions (rows, columns, cells), resolution, extent, and coordinate system.

Synergy of sf and terra

The true power of these packages is revealed when they are used together. A classic task in the DSM is to extract (extract) the values of raster covariates at the locations of point samples. Terra makes this extremely simple with the `extract()` function.

```
# Extract DEM values for each soil point location
# The function takes the raster and the sf object as
input
point_elevations <- extract(dem, soil_points)

# The result is a data frame, let's bind it to our
original sf object
soil_points_with_dem <- cbind(soil_points,
point_elevations)

print(soil_points_with_dem)
Simple feature collection with 26 features and 9 fields
Geometry type: POINT
Dimension:      XY
```

Bounding box: 17.11 Edge: 47.88 xmax: 20.25 ymax: 49.22
 Geodetic CRS: WGS 84
 First 10 features:

profile_id	horizon	depth_cm	soc_percent	clay_percent	ph_h2o	WRB ID	dmr3_5_100
1	SVK-01	Ap	0	4.2	28		
6.8	Chernozem	1	153.2029				
2	SVK-01 A	25 2.1	29 7.1	Chernozem 2	153.2029		
3	SVK-01	C	60	0.5	25		
8.2	Chernozem	3	153.2029				
4	SVK-02	Ap	0	3.8	32		
6.5	Phaeozem	4	115.9645				
5	SVK-02 A	22 1.9	33 6.8	Phaeozem 5	115.9645		
6	SVK-02	C	70	0.4	30		
7.9	Phaeozem	6	115.9645				
7	SVK-03 AP	0 2.5	22 6.2	Luvisol 7	210.0391		
8	SVK-03 and	20 0.8	15 5.8	Luvisol 8	210.0391		
9	SVK-03 Bt	50 0.6	35 6.1	Luvisol 9	210.0391		
10	SVK-04 A	0 3.1	18 5.9	Cambisol 10	723.4901		

Geom

- POINT (17.11 48.15)
- POINT (17.11 48.15)
- POINT (17.11 48.15)
- POINT (17.55 48.05)
- POINT (17.55 48.05)
- POINT (17.55 48.05)
- POINT (18.23 48.34)
- POINT (18.23 48.34)
- POINT (18.23 48.34)
- POINT (19.04 48.73)

EXTRACTION

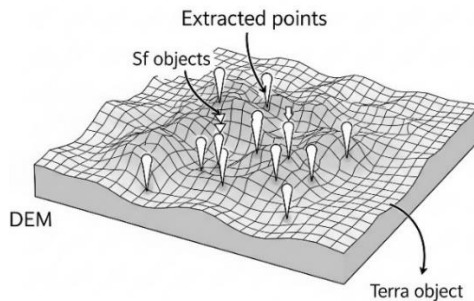


Fig. 1.17. Visualization of the extraction process. Points (sf object)

superimposed on the DEM raster surface (terra object) are shown

With one team, we solved the key task of data preparation. The combination of `sf` for vector operations, `terra` for raster operations, and `dplyr` for general manipulations creates a modern, fast, and logically consistent workflow that is ideal for all stages of predictive soil modeling.

5.2. Processing vector data from `sf`

As we have already found out, the `sf` package represents vector data in the form of regular tables with an additional geometry column. This elegant structure allows us to apply not only the standard verbs `dplyr` to manipulate attributes, but also to use a rich set of specialized functions to perform geospatial operations. These operations are the basis for data preparation in digital soil science, allowing us to solve tasks such as selecting samples within the study area, creating buffer zones or redesigning data.

Most spatial functions in `sf` are prefixed `st_` (from *spatial type*). We consider the most important of them.

Coordinate System Management (CRS)

Each geospatial dataset has a **Coordinate Reference System (CRS)** that determines how coordinates from the two-dimensional file space relate to real places on the Earth's surface. Working with data that has different or undefined CRS is a common source of errors. `sf` provides simple tools for working with CRS.

```
st_crs() : Allows you to check the CRS of an existing
object.

# Load sf and dplyr
library(sf)
library(dplyr)

# Let's assume we have our soil_points object from the
previous section
# First, check the current CRS
st_crs(soil_points)
```

```

Coordinate Reference System:
  User input: WGS 84
  wkt:
GEOGCRS["WGS 84",
  ENSEMBLE["World Geodetic System 1984 ensemble",
    MEMBER["World Geodetic System 1984 (Transit)"],
    MEMBER["World Geodetic System 1984 (G730)"],
    MEMBER["World Geodetic System 1984 (G873)"],
    MEMBER["World Geodetic System 1984 (G1150)"],
    MEMBER["World Geodetic System 1984 (G1674)"],
    MEMBER["World Geodetic System 1984 (G1762)"],
    MEMBER["World Geodetic System 1984 (G2139)"],
    MEMBER["World Geodetic System 1984 (G2296)"],
    ELLIPSOID["WGS 84",6378137,298.257223563,
      LENGTHUNIT["metre",1]],
    ENSEMBLEACCURACY[2.0]],
  PRIMEM["Greenwich",0,
    ANGLEUNIT["degree",0.0174532925199433]],
  CS[ellipsoidal,2],
    AXIS["geodetic latitude (Lat)",north,
      ORDER[1],
      ANGLEUNIT["degree",0.0174532925199433]],
    AXIS["geodetic longitude (Lon)",east,
      ORDER[2],
      ANGLEUNIT["degree",0.0174532925199433]],
  USAGE[
    SCOPE["Horizontal component of 3D system."],
    AREA["World."],
    BBOX[-90,-180,90,180]],
  ID["EPSG",4326]]

```

- **st_transform()**: Reprojects data from one CRS to another.

This is critically important, since operations such as calculating distances or areas give correct results only in projected (flat) coordinate systems (e.g. UTM/Mercator etc) and not in geographic ones (latitude/longitude).

```

# Let's say we need to transform it to a projected CRS,
# e.g., WGS 84 / Pseudo-Mercator (EPSG:3857)
soil_points_mercator <- st_transform(soil_points, crs =
3857)

# Check the new CRS
st_crs(soil_points_mercator)

```

```

Coordinate Reference System:
  User input: EPSG:3857
  wkt:
PROJCRS["WGS 84 / Pseudo-Mercator",
  BASEGEOGCRS["WGS 84",
    ENSEMBLE["World Geodetic System 1984 ensemble",
      MEMBER["World Geodetic System 1984
(Transit)"],
      MEMBER["World Geodetic System 1984 (G730)"],
      MEMBER["World Geodetic System 1984 (G873)"],
      MEMBER["World Geodetic System 1984 (G1150)"],
      MEMBER["World Geodetic System 1984 (G1674)"],
      MEMBER["World Geodetic System 1984 (G1762)"],
      MEMBER["World Geodetic System 1984 (G2139)"],
      MEMBER["World Geodetic System 1984 (G2296)"],
      ELLIPSOID["WGS 84",6378137,298.257223563,
        LENGTHUNIT["metre",1]],
      ENSEMBLEACCURACY[2.0]],
    PRIMEM["Greenwich",0,
      ANGLEUNIT["degree",0.0174532925199433]],
    ID["EPSG",4326]],
  CONVERSION["Popular Visualisation Pseudo-Mercator",
    METHOD["Popular Visualisation Pseudo Mercator",
      ID["EPSG",1024]],
    PARAMETER["Latitude of natural origin",0,
      ANGLEUNIT["degree",0.0174532925199433],
      ID["EPSG",8801]],
    PARAMETER["Longitude of natural origin",0,
      ANGLEUNIT["degree",0.0174532925199433],
      ID["EPSG",8802]],
    PARAMETER["False easting",0,
      LENGTHUNIT["metre",1],
      ID["EPSG",8806]],
    PARAMETER["False northing",0,
      LENGTHUNIT["metre",1],
      ID["EPSG",8807]]],
  CS[Cartesian,2],
    AXIS["easting (X)",east,
      ORDER[1],
      LENGTHUNIT["metre",1]],
    AXIS["northing (Y)",north,
      ORDER[2],
      LENGTHUNIT["metre",1]],
  USAGE[
    SCOPE["Web mapping and visualisation."],
    AREA["World between 85.06°S and 85.06°N."],

```

```
BBOX[-85.06,-180,85.06,180]],
ID["EPSG",3857]]
```

Spatial queries and subsets

For example, to select only those sample points that fall within the boundaries of a certain administrative unit or experimental area. For this, spatial predicate functions (for example, `st_intersects`, `st_within`) or, which is easier for beginners, the **function `st_intersection()`** are used.

```
#st_intersection() "crops" the first object according to
the geometry of the second, leaving only the part that
falls inside.
# Let's load a polygon of our study area
study_area <- st_read("gis_data/study_area_polygon.gpkg")
Reading layer `study_area_polygon' from data source

`D:\TextbookPredSoilMapping\gis_data\study_area_polygon.g
pkg' using driver `GPKG'
Simple feature collection with 1 feature and 0 fields
Geometry type: POLYGON
Dimension:      XY
Bounding box:   xmin: 1980952 ymin: 6128390 xmax: 2304214
ymax: 6305790
Projected CRS:  WGS 84 / Pseudo-Mercator

# Ensure both layers have the same CRS before
intersection
study_area <- st_transform(study_area, crs =
st_crs(soil_points))

# Select only the points that fall within the study area
points_in_area <- st_intersection(soil_points,
study_area)

# Print selectedd points in the study area
points_in_area
Simple feature collection with 12 features and 7 fields
Geometry type: POINT
Dimension:      XY
Bounding box:   xmin: 18.23 ymin: 48.34 xmax: 20.25 ymax:
49.01
Geodetic CRS:   WGS 84
First 10 features:
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o      WRB              geom
```

7	SVK-03	Ap	0	2.5	22
6.2	Luvisol POINT	(18.23 48.34)			
8	SVK-03	Et	20	0.8	15
5.8	Luvisol POINT	(18.23 48.34)			
9	SVK-03	Bt	50	0.6	35
6.1	Luvisol POINT	(18.23 48.34)			
10	SVK-04	A	0	3.1	18
5.9	Cambisol POINT	(19.04 48.73)			
11	SVK-04	Bw	25	1.2	24
6.2	Cambisol POINT	(19.04 48.73)			
12	SVK-04	C	80	0.3	22
6.5	Cambisol POINT	(19.04 48.73)			
19	SVK-07	A	0	6.5	25
5.5	Cambisol POINT	(19.45 49.01)			
20	SVK-07	Bw	30	2.8	28
5.9	Cambisol POINT	(19.45 49.01)			
21	SVK-08	A	0	0.8	5
4.8	Leptosol POINT	(20.25 48.91)			
22	SVK-08	C	15	0.2	8
5.2	Leptosol POINT	(20.25 48.91)			

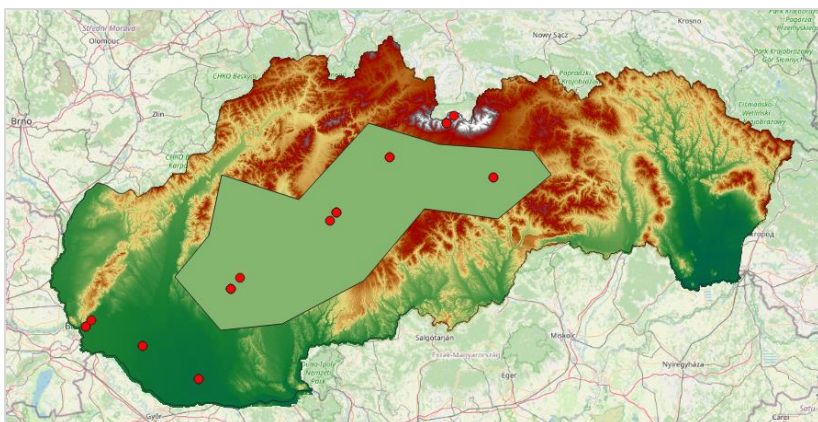


Fig. 1.18. Visualization of spatial filtering. A set of points and a test site of the experimental site are shown. As a result of the operation, only those points that are inside the landfill remain `st_intersection`

Creating buffer zones (`st_buffer()`)

Buffering is the process of creating a zone of a certain distance around a geospatial feature. This is an extremely useful operation in soil

science. For example, we can create a buffer with a radius of 500 meters around each sampling point, and then calculate the average of the slope or other terrain indicator within that zone.

Important: for the correct calculation of the buffer, the data must be in the designed coordinate system, where the units of measurement are meters, not degrees.

```
# Use our Mercator-projected points
# Create a 50-meter buffer around each point
point_buffers <- st_buffer(soil_points_mercator, dist = 500)
```

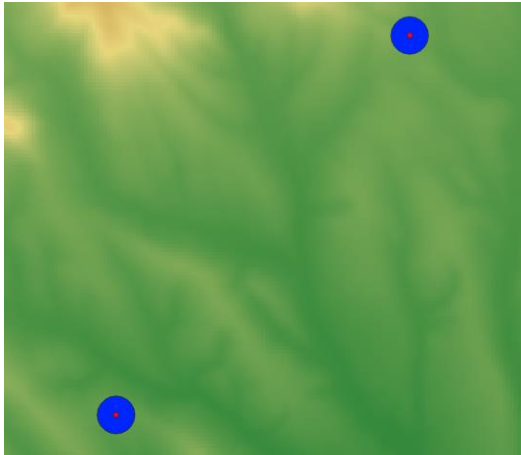


Fig. 1.19. The result of the buffering operation. A circular polygon with a radius of 500 meters has been created around each point

Combination of sf and dplyr

The true beauty of sf is revealed when we combine spatial functions with dplyr verbs in a single `%>%` chain. Since the sf object is a table, we can integrate operations seamlessly.

Suppose we need to: 1) select only points with a horizon of "Ap", 2) create a buffer zone of 500 meters for them, 3) calculate the area of each buffer zone.

```
# A powerful workflow combining dplyr verbs and sf functions
```



```

ap_horizon_buffers <- soil_points_mercator %>%
  filter(horizon == "Ap") %>%
  st_buffer(dist = 500) %>%
  mutate(buffer_area_sqm = st_area(.))

print(ap_horizon_buffers)
Simple feature collection with 5 features and 8 fields
Geometry type: POLYGON
Dimension: XY
Bounding box: xmin: 1904176 ymin: 6114677 xmax: 2037647
ymax: 6174160
Projected CRS: WGS 84 / Pseudo-Mercator
  profile_id horizon depth_cm soc_percent clay_percent
ph_h2o     WRB
1      SVK-01      Ap         0           4.2           28
6.8 Chernozem POLYGON ((1905176 6131846, ...
2      SVK-02      Ap         0           3.8           32
6.5 Phaeozem POLYGON ((1954157 6115177, ...
3      SVK-03      Ap         0           2.5           22
6.2 Luvisol POLYGON ((2029854 6163606, ...
4      SVK-09      Ap         0           4.5           30
7.2 Chernozem POLYGON ((1909629 6136853, ...
5      SVK-10      Ap         0           2.8           25
6.0 Luvisol POLYGON ((2037647 6173660, ...
  buffer_area_sqm
1  785039.3 [m^2]
2  785039.3 [m^2]
3  785039.3 [m^2]
4  785039.3 [m^2]
5  785039.3 [m^2]

```

This code is concise, readable, and fully reproducible. It demonstrates how `sf` transforms R into a full-fledged, code-driven geographic information system that perfectly meets the needs of today's digital soil science. Compared to the old `sp` package approach, where different syntaxes and access methods had to be used for each operation with attributes (`data.frame`) and geometry (`Spatial*`), the workflow with `sf` is much more intuitive and consistent.

5.3. Processing raster data from terra

If vector data represent discrete objects on the earth's surface, then **raster data** represent continuous phenomena. A raster is essentially a

grid of cells (pixels), where each cell has a specific value that characterizes the phenomenon at a given point (e.g. altitude, temperature, concentration of a chemical element). In digital soil science, rasters are our main source of predictive variables (covariate) for modeling: digital elevation model (DEM), indicators derived from it, such as slope and exposure, remote sensing data, climatic surfaces, etc.

The terra package is a modern, fast and efficient tool for working with raster data in R. It replaced its predecessor, the raster package, offering significantly higher performance and more intuitive syntax. We consider the key raster operations that are necessary to prepare data for modeling.

Basic Operations and Raster Mathematics (Map Algebra)

The basis for many operations is the concept of "Map Algebra". It allows mathematical functions and operators to be applied to raster layers as if they were ordinary variables. Terra performs these operations pixel-by-pixel.

We download our Digital Elevation Model (DEM) and perform a few basic operations.

```
# Load the terra package
library(terra)

# Load the DEM raster
time <- true ("gis_data/dem_slovakia_subset.tif")

# Basic arithmetic: convert elevation from meters to feet
dem_feet <- dem * 3.28084

# We can also apply functions, e.g., calculate the
natural logarithm of elevation
# Adding 1 to avoid log(0) if there are sea-level pixels
log_dem <- log(dem + 1)
```

'terra' automatically creates new raster objects in memory. To visualize the results, you can use the built-in 'plot()' function.

```
# Plot the results
plot(log_dem, main = "Log-transformed Elevation")
```

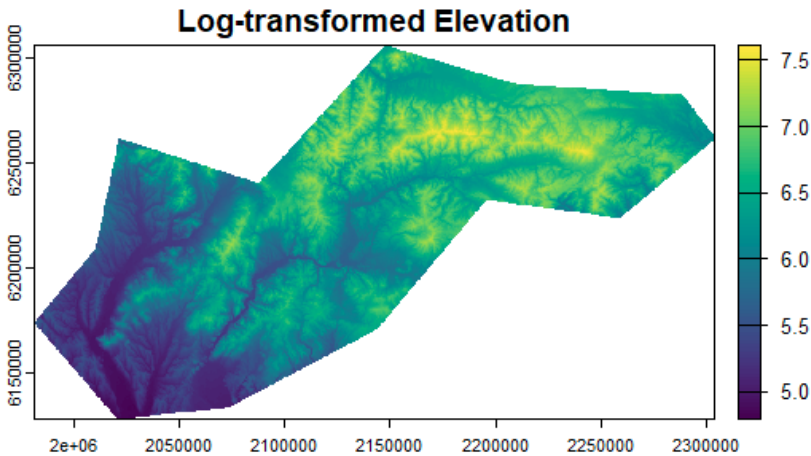


Fig. 1.20. Visualization of a raster of logarithmic height. The color scale shows the changed values, but the spatial structure of the data remains the same

Calculation of relief derivatives

One of the most important steps in the preparation of covariate is the calculation of morphometric indicators based on DEM. The terra package has a built-in `terrain()` function that makes it easy to calculate the most common ones.

```
# Calculate slope (ухил) and aspect (експозиція) from the
DEM
# The 'unit' argument specifies whether the result should
be in degrees or radians
slope <- terrain(dem, v = "slope", unit = "degrees")
aspect <- terrain(dem, v = "aspect", unit = "degrees")

# We can create a multi-layer SpatRaster object to hold
all terrain variables
terrain_derivatives <- c(dem, slope, aspect)
names(terrain_derivatives) <- c("elevation", "slope",
"aspect")

# Save the multi-layer raster to a GeoTIFF file
writeRaster(terrain_derivatives,
"gis_data/terrain_derivatives.tif", overwrite = TRUE)

# Plot all layers at once
```

```
plot(terrain_derivatives)
```

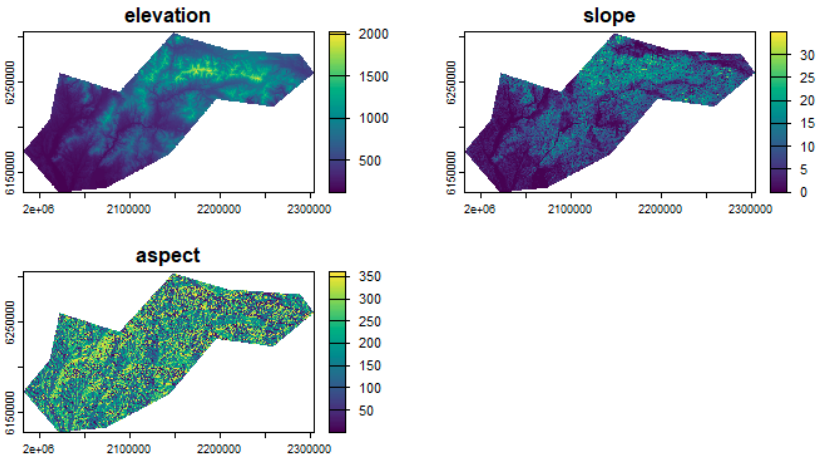


Fig. 1.21. Maps of relief derivatives. Three rasters are shown: the original DEM, the calculated slope map, and the exposure map

Area statistics

It is often necessary to generalize the raster values within certain zones defined by the vector polygonal layer. For example, calculate the average height or average slope for each administrative unit. This operation is called **zonal statistics**. In terra, the `zonal()` function is designed for this.

```
# Assume we have a vector layer of study regions
('study_area' from previous section)
# Calculate the mean and standard deviation of elevation
for each region
zonal_stats <- extract(dem, study_area, fun = "mean",
na.rm = TRUE)

print(zonal_stats)
  ID dem_slovakia_subset
1  1                593.9552
```

The result will be a table where the corresponding statistical indicator will be calculated for each polygon from `study_area` (in the tutorial example, there is only one polygon, so only one value is displayed). It

is an extremely powerful tool for data aggregation.

Changing the resolution and trimming the raster

For predictive modeling, all raster covariates must have **the same extent (spatial coverage) and the same resolution (pixel size)**. Terra provides simple tools for this.

- **crop()**: trims the raster to the extent of another spatial object (raster or vector).

```
# Assume we have another raster 'land_cover' with a
different extent and resolution
land_cover <-
rast("gis_data/SVK_ESA_WorldCover_2020_25m_study_area.tif")

# Crop the DEM to match the extent of the land cover
raster
dem_cropped <- crop(dem, land_cover)
dem_cropped
class      : SpatRaster
dimensions : 1773, 3232, 1  (nrow, ncol, nlyr)
resolution : 100, 100  (x, y)
extent     : 1981012, 2304212, 6128427, 6305727  (xmin,
xmax, ymin, ymax)
coord. ref.: WGS 84 / Pseudo-Mercator (EPSG:3857)
source     : dem_slovakia_subset.tif
name       : dem_slovakia_subset
```

- **resample()**: resamples the raster to the resolution and mesh of another raster template.

```
# Resample the land cover raster to match the grid of our
cropped DEM
land_cover_resampled <- resample(land_cover, dem_cropped,
method = "near") # 'near' for categorical data

land_cover
class      : SpatRaster
dimensions : 7095, 12929, 1  (nrow, ncol, nlyr)
resolution : 25, 25  (x, y)
Magnitude  : 1980975, 2304200, 6128400, 6305775 (xmin,
xmax, ymin, ymax)
coord. ref.: WGS 84 / Pseudo-Mercator (EPSG:3857)
```

```

Source: SVK_ESA_WorldCover_2020_25m_study_area.tif
name      : SVK_ESA_WorldCover_2020_25m_study_area
min value : 0
max value : 90

land_cover_resampled
class      : SpatRaster
dimensions : 1773, 3232, 1 (nrow, ncol, nlyr)
resolution : 100, 100 (x, y)
Magnitude  : 1981012, 2304212, 6128427, 6305727 (xmin,
xmax, ymin, ymax)
coord. ref.: WGS 84 / Pseudo-Mercator (EPSG:3857)
source(s)  : memory
varname    : dem_slovakia_subset
name       : SVK_ESA_WorldCover_2020_25m_study_area
min value  : 0
max value  : 90

```

These operations are critical to creating a consistent stack of raster predictors, which are inputs to machine learning algorithms.

The terra package provides a complete set of tools for efficient and fast processing of raster data. Its close integration with sf and dplyr creates a single, logical and high-performance environment for the implementation of the entire cycle of geospatial data preparation for digital soil science.

5.4. Integration of spatial data

In the previous sections, we learned how to work with vector (sf) and raster (terra) data separately. We transformed coordinate systems, created buffers, calculated relief derivatives, and prepared rasters for joint analysis. Now it's time to merge these two worlds. **Spatial data integration** is the process of combining information from different data sources and types (vector and raster) into a single, coherent structure. For digital soil science, this is perhaps the most important step in data preparation, as this is where we link our field observations (points) to the continuous surfaces of environmental factors (raster covariates).

The main task at this stage is to create a final table for modeling, where each row corresponds to an observation point (soil profile) and the columns contain both the soil properties measured at that point (our

target variable, such as soil type or SOC content) and the values of all our raster predictors at that same point.

Extraction of raster stack values by points

The key operation for integration is **extraction** (extraction). This process consists, as described above, of "piercing" one or more raster layers at the locations defined by vector points and writing the values of the corresponding pixels to the attribute table of these points. The terra package makes this operation extremely simple and fast with the `extract()` function.

In the previous steps, we prepared a stack of raster covariates (height, slope, and exposure) and now want to get the values of these predictors for each sampling point.

```
# Load necessary libraries
library(terra)
library(sf)
library(dplyr)

# Load the multi-layer raster of terrain derivatives we
# created earlier
# It contains 'elevation', 'slope', and 'aspect'
terrain_derivatives <- True
("gis_data/terrain_derivatives.tif")

# Load the vector points of soil samples
soil_points <-
st_read("gis_data/slovakia_soil_points_3857.gpkg")

# Ensure the CRS of both datasets match
# Let's assume they are already aligned from previous
# steps
soil_points <- st_transform(soil_points, crs = crs
(terrain_derivatives))

# Perform the extraction
# The function returns a data frame with an ID and values
# for each raster layer
extracted_values <- extract(terrain_derivatives,
soil_points)

# View the rows of the result
extracted_values
```

	ID	elevation	slope	aspect
1	1	NA	NA	
2	2	THAT IS ALREADY		
3	3	THAT IS ALREADY		
4	4	NA	NA	
5	5	THAT IS ALREADY		
6	6	THAT IS ALREADY		
7	7	210.0391	0.9473696	198.664124
8	8	210.0391	0.9473696	198.664124
9	9	210.0391	0.9473696	198.664124
10	10	723.4901	8.9733219	2.968902
11	11	723.4901	8.9733219	2.968902
12	12	723.4901	8.9733219	2.968902
13	13	NA	NA	
14	14	NA		
15	15	NA		
16	16			
17	17	NA	NA	
18	18	IS ALREADY THERE		
19	19	970.3638	22.3610878	129.136536
20	20	970.3638	22.3610878	129.136536
21	21	914.6369	14.0763817	116.459236
22	22	914.6369	14.0763817	116.459236
23	23	NA	NA	
24	24	226.6492	1.0106043	68.995308
25	25	NA	NA	
26	26	1064.1965	11.9279490	273.044006

The result is a regular data table. The first column (ID) corresponds to the point number, and the next columns are the values from each layer of our raster stack at that point. Note that points that lie outside the raster layer when extracting raster stack values received a value of NA – Not Available.

Creating the final dataset for modeling

Now we only need to attach these extracted values to our original table with points. This can be done with `cbind()` or, more reliably and tidyverse-style, with `dplyr::bind_cols()`.

```
library(dplyr)
# Combine the original sf object with the extracted
covariate values
final_modeling_data <- bind_cols(soil_points,
```



```

extracted_values) %>%
  select(-ID) # We can remove the redundant ID column

# View the resulting integrated dataset
print(final_modeling_data)

```

```

> print(final_modeling_data)
Simple feature collection with 26 features and 10 fields
Geometry type: POINT
Dimension: XY
Bounding box: xmin: 1904676 ymin: 6086914 xmax: 2254220 ymax: 6312274
Projected CRS: WGS 84 / Pseudo-Mercator
First 10 features:
  profile_id horizon depth_cm soc_percent clay_percent ph_h2o wrb_msg elevation slope aspect geom
1 SVK-01 AP 0 4.2 28 6.8 Chernozem NA NA NA POINT (1904676 6131846)
2 SVK-01 A 25 2.1 29 7.1 Chernozem NA NA NA POINT (1904676 6131846)
3 SVK-01 C 60 0.5 25 8.2 Chernozem NA NA NA POINT (1904676 6131846)
4 SVK-02 AP 0 3.8 32 6.5 Phaeozem NA NA NA POINT (1953657 6115177)
5 SVK-02 A 22 1.9 33 6.8 Phaeozem NA NA NA POINT (1953657 6115177)
6 SVK-02 C 70 0.4 30 7.9 Phaeozem NA NA NA POINT (1953657 6115177)
7 SVK-03 AP 0 2.5 22 6.2 LuvISO1 210.0391 0.9473696 198.664124 POINT (2029354 6163606)
8 SVK-03 ET 20 0.8 15 5.8 LuvISO1 210.0391 0.9473696 198.664124 POINT (2029354 6163606)
9 SVK-03 BT 50 0.6 35 6.1 LuvISO1 210.0391 0.9473696 198.664124 POINT (2029354 6163606)
10 SVK-04 A 0 3.1 18 5.9 CambISO1 723.4901 8.9733219 2.968902 POINT (2119523 6229172)
> |

```

Fig. 1.22. Structure of the final dataset for modeling. A table is shown, where the first columns contain the initial attributes of the points (ID, soil type), followed by values extracted from the raster covariates (elevation, slope, aspect), and the last columns contain the geometry of the points

Now our `final_modeling_data` object is a complete dataset ready for modeling. It contains all the necessary information in a "neat" format: **Observation IDs**.

- **The target variable** (for example, `soil_type`).

Predictor variables (covariates) extracted from raster layers.

The spatial geometry of each point.

This integrated approach, combining the power of `sf`, the speed of `terra` and the readability of `dplyr`, is the foundation of the modern workflow in digital soil science. It allows you to automate and reproduce one of the most time-consuming stages of research – the preparation and harmonization of data from various sources.

Having completed this part, we have prepared a solid foundation. We have gone from installing R to creating a full-fledged, integrated geospatial dataset. Now we are ready to move on to the second part of this manual and use the knowledge of obtaining this kind of data to build predictive models of soil types and properties.

PART II. PREDICTIVE MODELING OF SOIL TYPES

Chapter 6. Theoretical foundations of digital soil mapping

6.1. DSM Concept

By completing the first part of this tutorial, we have mastered a powerful set of tools for working with data in R. We have learned how to import, manipulate, visualize, and integrate a variety of spatial and aspatial data. Now we are ready to apply these skills to solve the central task of our book – predictive soil modeling, also known as **Digital Soil Mapping (DSM)**.

What is DSM? At its core, DSM is the creation and dissemination of soil information using numerical methods based on soil observations and associated data on environmental factors (McBratney et al., 2003). This is a radical departure from traditional soil mapping, which relied heavily on manual interpolation, expert knowledge, and qualitative delineation of soil contours in the field. The DSM, on the other hand, is a quantitative, objective and reproducible approach.

The fundamental theoretical premise on which all digital soil mapping is based is the famous concept of soil formation factors, first formulated by V.V. Dokuchaev, and later formalized by Hans Jenny in the form of an equation:

$$S=f(cl,o,r,p,t,...)$$

where:

- **S** is the soil or a specific property of the soil (e.g. soil type, clay content, pH).
- **f** – denotes "function of" or "dependence on".
- **Cl** – climate.
- **o** – organisms or biota, including vegetation and human activities (organisms).
- **r** – relief or topography.
- **p** – parent material.
- **t** – time.
- **...** – three dots indicate that there may be other, locally significant factors (for example, spatial position).

Traditional soil science used this model conceptually: a soil scientist

in the field, observing changes in relief or vegetation, inferred a change in soil type. The DSM takes the next step: it tries **to quantify** this function f .

The idea behind DSM is to use available point data on soils (where we know property **S**) and large geospatial data sets that characterize soil formation factors (**cl, o, r, p, t**) to build a statistical or machine learning model. This model "learns" on point data by finding quantitative relationships between soil properties and values of environmental factors at these points.

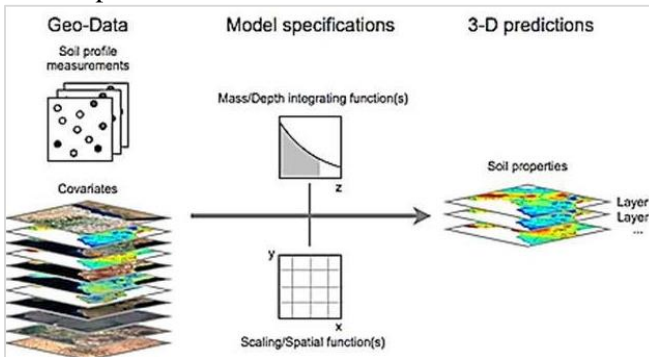


Fig. 2.1. Conceptual diagram of the DSM workflow. It is shown how point data about soils and the stack of raster covariates (relief, climate, remote sensing) are fed to the input to the machine learning model. The model examines dependencies and is then applied to the entire covariate stack to create a continuous predictive map of soil property (figure from [Divya R.K](#))

The DSM workflow can be thought of as follows:

- **Data collection:** We have a set of points (soil profiles) where we know the target property (e.g. soil type). We also have a set of raster layers (covariate) covering our entire area and representing soil formation factors (e.g. DEM, slope, satellite indices, geologic map).

Integration: Using the methods we learned in Chapter 5, we extract the values of all the raster covariates at each observation point. In this way, we create a single table to train the model.

Modeling: We use this table to "train" a model (e.g., a decision tree or a random forest) to find patterns. The model learns, for example, that

"high organic matter is typically observed on northern slopes with a low slope and under forest vegetation."

Prediction: Once the model is trained, we apply it to our entire raster covariate stack. For each pixel on the map, the model takes the values of elevation, slope, vegetation, etc., and, based on the learned patterns, predicts the most likely soil property value for that pixel.

The result is a continuous digital map that predicts soil property for the entire area, not just at observation points. This approach not only allows for more detailed and objective maps, but also provides an opportunity to assess the uncertainty of our forecasts, which is a huge advantage over traditional methods.

6.2. Detailed overview of the SCORPAN model

In the previous subsection, we found that the DSM is based on the quantification of the classical equation of soil formation factors $S=f(cl,o,r,p,t,...)$. Although this formula is conceptually powerful, it has been extended and refined for practical application in predictive modeling. The most common and functional version of this model in the modern DSM is the **SCORPAN model** (McBratney et al., 2003).

SCORPAN is a mnemonic acronym that not only incorporates the classic Yenne factors, but also adds new ones, which are critical for statistical modeling. The equation takes the form:

$$\mathbf{S}=f(\mathbf{s},\mathbf{c},\mathbf{o},\mathbf{r},\mathbf{p},\mathbf{a},\mathbf{n})$$

where **S** is now explicitly taken out as the target variable (soil property), and **a** (age) and **n** (spatial position) are added to the classical factors. This extension makes the model more pragmatic and adaptable to the real-world data we are working with. We take a closer look at each component of this model and, most importantly, how we can represent it as digital, spatial data (covariate) for our model in R.

- **S – Soil attributes:** This is our **target variable** – what we want to predict. These can be:
 - ✓ Categorical variables: for example, the type of soil according to the national or international classification (WRB, USDA Soil Taxonomy). In this case, we solve the **classification** problem.
 - ✓ **Continuous (quantitative) variables:** organic carbon

content (SOC), pH, clay content, cation exchange capacity (ECO), folding density, etc. Here we solve the **regression problem**. The source of this data is our field observations and laboratory analyses.

- **C – Climate:** Climatic factors (precipitation, temperature) are the driving forces of weathering, transport of substances and biological activity. In DSM models, climate is usually represented by raster surfaces of long-term averages, for example:
 - ✓ Average annual rainfall.
 - ✓ Average annual temperature.
 - ✓ Evapotranspiration indicators. This data can be obtained from global climate databases such as WorldClim or from regional meteorological networks.
- **O – Organisms:** This factor includes the influence of vegetation, microorganisms, animals and, more and more importantly, humans (land use). As covariates we can use:
 - ✓ **Remote sensing data:** satellite indices characterizing vegetation, such as the NDVI (Normalized Difference Vegetation Index) obtained from Landsat or Sentinel-2 images.
 - ✓ **Land Cover/Land Use Maps:** e.g. from CORINE Land Cover projects.
 - ✓ **Maps of forest or** other natural vegetation types.
- **R – Relief or Topography:** Relief is one of the most important factors at the local and regional levels, as it redistributes energy and moisture. Almost all covariates describing the relief are obtained by analyzing **the Digital Relief Model (DEM)**. They are divided into:
 - ✓ **Primary derivatives:** absolute height (DEM itself), slope, exposure (aspect).
 - ✓ **Secondary derivatives:** topographic wetness index (TWI), profile and plan curvature, stream power index. We can easily calculate these indicators using terra packages or specialized GIS programs (SAGA, GRASS).
- **P – Parent material:** It is the starting material from which the soil is formed, and it determines its mineralogical and initial chemical composition. This factor is usually represented as:

- ✓ **Digitized geological or quaternary maps** that are rasterized so that each pixel has a value corresponding to a specific type of rock.
- **A – Age:** This is the time during which soil formation factors acted on the parent rock. This is the most difficult factor to quantify in space. There are no direct age maps, so proxy variables are used:
 - **Maps of geomorphological surfaces** (e.g. river terraces of different ages).
 - **Distance to rivers or glacier boundaries**, if relevant to the territory.
- **N – Spatial position:** This factor was added to explicitly account for the spatial dependence (autocorrelation) in soil properties, which cannot be fully explained by other SCORPA factors. As covariates can be used:
 - Geographic coordinates (X and Y) directly as predictors.
 - Distance to certain objects (for example, to the coast, to a mountain range).

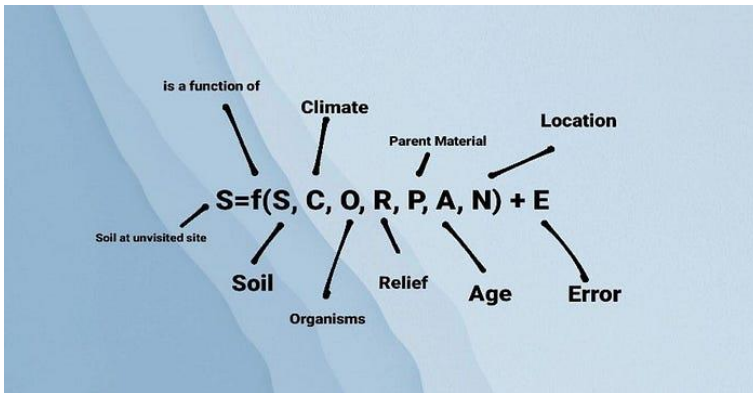


Fig. 2.2. Visualization of the components of the SCORPAN model. A diagram where each letter of the acronym is associated with an example of the corresponding raster layer-covariate (figure from [Alfiya Quraishi](#))

Thus, the SCORPAN model provides us with a clear and comprehensive conceptual framework for the selection of predictor variables. Our challenge as digital soil scientists is to find the best available spatial data that represents each of these factors for our study

area and use it to build an accurate predictive model.

6.3. DSM Workflow Overview

Armed with the SCORPAN conceptual model, we can now outline a standardized **workflow** for digital soil mapping. This process is a sequence of logical steps that leads us from raw data to the final predictive map and evaluating its reliability. since it allows you to systematize work, ensure its reproducibility and avoid common mistakes.

The entire DSM workflow can be roughly divided into three main phases: **Data Preparation, Modeling and Validation**, and **Spatial Forecasting**.

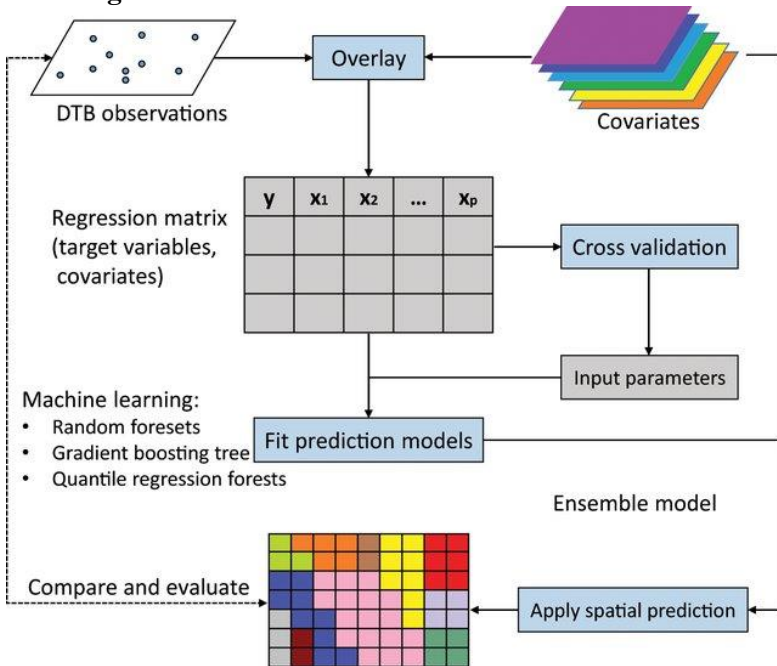


Fig. 2.3. Detailed flowchart of the Digital Soil Mapping workflow (from Yan et al., 2020)

Phase 1: Data Preparation and Integration

This is the most time-consuming, but also the most important phase, since the quality of the final map directly depends on the quality of the

input data. This phase includes the steps that we have covered in detail in Part I.

- **Collection of Point Soil Data (S):** Formation of a georeferenced dataset containing our target variable (e.g., soil type, pH, SOC).
- **Spatial Covariate Collection (CORPAN):** Search, load, and pre-treatment of raster and vector layers representing soil formation factors.
- **Covariate harmonization:** Bringing all raster layers to a single coordinate system, spatial extent, and resolution. This is a critical step to create a consistent "stack" of predictors.
- **Data Integration:** Extracting values from a harmonized covariate stack at the locations of point soil data. The result of this phase is a single, "tidy" table ready for modeling.

Phase 2: Simulation and Validation

At this point, we use the prepared table to build and evaluate our predictive model.

- **Data splitting:** The entire set of point data is usually divided into two parts: **a training set**, which is used to "train" the model (usually 70-80% of the data), and **a test (or validation) set**, which is deferred and not used for training.
- **Model Training:** A training sample is fed into the input of the selected machine learning algorithm (e.g., Random Forest, Decision Trees). The algorithm analyzes the data and builds a mathematical model describing the relationship between the predictors (CORPAN) and the target variable (S).
- **Model validation:** After training, we must objectively assess how well our model works. To do this, we use a test sample that the model "did not see" during training. We force the model to make predictions for points from the test sample and compare these predictions with real values known to us. This allows us to calculate accuracy metrics (e.g. overall accuracy, Kapp coefficient for classification; R^2 , RMSE for regression) and understand whether our model is overtrained and whether it is capable of generalizing patterns to new data.

Phase 3: Spatial Forecasting and Interpretation

Once we have made sure that our model is accurate enough, we can use it to create the final product – the map.

- **Spatial Forecasting:** The trained model is applied to the entire stack of raster covariates. For each pixel of our study area, the model takes the values of all predictors and makes a prediction of the target variable.
- **Creation of final maps:** The result of the prediction is a new raster layer (or multiple layers), which is a predictive map of the soil property. For continuous variables, this can be a map of the average expected value, as well as maps of the lower and upper limits of the confidence interval, visualizing the **uncertainty of the forecast**.
- **Post-processing and interpretation:** The final maps are visualized, drawn up and analyzed. It is important not only to obtain the map, but also to interpret it from the point of view of soil science, to check whether the spatial patterns on the map correspond to our expert ideas about the territory.

In the following sections, we will go through all these stages step by step with a practical example, using R to build predictive maps of soil types and organic carbon content for the territory of Slovakia.

Chapter 7. Preparing data for modeling: the case of Slovakia

Moving from theoretical foundations to practical application, in the following sections we will go step by step through the entire workflow of digital soil mapping with a real example. This country in Central Europe is an excellent example, as it is characterized by a significant variety of natural conditions – from lowland plains in the south to the mountain systems of the Carpathians in the north, – which leads to the formation of a wide range of soil types.

We will build two predictive models:

- **Classification model** for forecasting the main types of soils.

A regression model for predicting the content of organic carbon (SOC) – one of the key quantitative characteristics of the soil.

This section will be entirely devoted to the first and most important phase of the DSM workflow – data preparation and integration.

7.1. Determination of the study area

The first and fundamental step of any geospatial analysis is to clearly define the boundaries of the study area. This step is not a formality; The vector polygon delineating our territory will serve as a "template" for all further operations. It is outside of it that we will crop and mask all our raster covariates to ensure their complete spatial consistency.

For our example, we will use the administrative borders of Slovakia. Suppose that this data is stored in a file of the GeoPackage format, which is a modern and efficient standard for storing geospatial data.

We load this layer into R using the `sf` package and render it to make sure everything is loaded correctly.

```
# Load necessary libraries for spatial data handling and
# visualization
library(sf)
library(ggplot2)

# Define the path to our boundary data
# It's good practice to store spatial data in a dedicated
# subfolder, e.g., 'gis_data'
boundary_path <- "gis_data/slovakia_boundary.gpkg"
```

```
# Read the GeoPackage file into an sf object
slovakia_boundary <- st_read(boundary_path)

# Print the object to see its structure

print(slovakia_boundary)
Simple feature collection with 1 feature and 2 fields
Geometry type: MULTIPOLYGON
Dimension:      XYZ
Bounding box:   xmin: 1873858 ymin: 6062250 xmax: 2511997
ymax: 6379651
z_range:        zmin: 0 zmax: 0
Projected CRS:  WGS 84 / Pseudo-Mercator
OBJECTID Area_km2 geom
1          2 49026.12 MULTIPOLYGON Z (((2165404 6...
```

The output in the console will show us that `slovakia_boundary` is an `sf` object with one object (one polygon) and multiple attribute fields. It is important to pay attention to the information about the coordinate system (CRS). For further work, we will need to make sure that all our data is in a single CRS.

The best way to check if spatial data has been loaded correctly is to visualize it. `ggplot2` has a specialized geom for SF objects – `geom_sf()`, which makes creating maps extremely easy.

```
# Create a simple map of the study area
ggplot() +
  geom_sf(data = slovakia_boundary, fill = "lightgray",
  color = "black") +
  theme_bw() +
  labs(
    title = " Study area: Slovakia",
    x = "Longitude",
    y = "Latitude"
  )
```

Result Analysis: We have obtained a simple yet clear outline of our territory. `geom_sf()` automatically uses the geometry column and CRS information to display the data correctly. This object will `slovakia_boundary` become our primary tool for spatial filtering and preparation of all other data layers, ensuring that our final stack of predictors is perfectly aligned in spatial coverage.

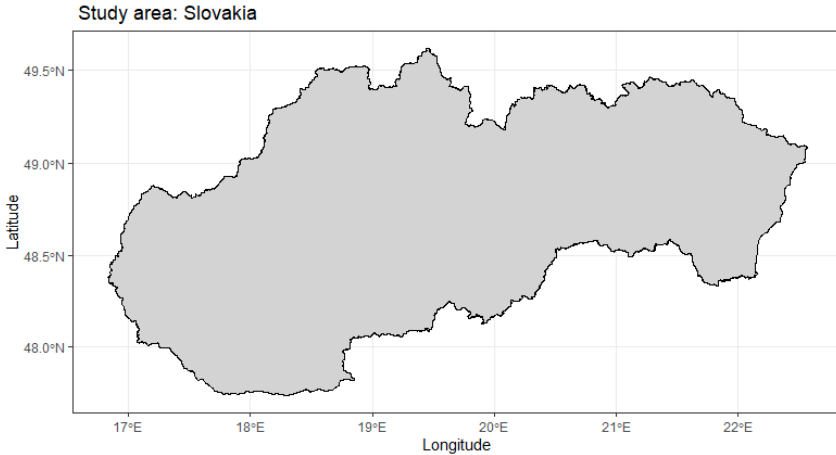


Fig. 2.4. Map of the study area. Shows the administrative borders of Slovakia downloaded from the GeoPackage file and visualized with ggplot2

7.2. Sources of point data on soils

Once the boundaries of our territory have been determined, the next step is to collect and analyze **point soil data**. This data is the empirical basis, the "ground truth" on which our model will be trained. This is the **S (Soil) component** of the SCORPAN model. The quality, quantity, and spatial distribution of this data directly determine the potential accuracy and reliability of our final predictive map.

For our example, we will use a dataset based on the data obtained as part of the creation of the Global Soil Organic Carbon Map (GSOCmap) based on the detailed soil survey program of Slovakia. We have also attached data containing information about soil types, including their geographical coordinates and classification belonging to the main abstract soil taxon (Major Soil Group) according to the WRB (World Reference Base for Soil Resources) classification.

We download this data, which, like the boundaries, is stored in GeoPackage format, and conduct an initial exploratory analysis.

```
# Load necessary libraries
library(sf)
```

```

library(dplyr)
library(ggplot2)

# Path to the soil point data
points_path <- "gis_data/slovakia_soil_points_3857.gpkg"

# Read the GeoPackage file into an sf object
soil_points <- st_read(points_path)
Reading layer `slovakia_soil_points_3857' from data
source

`D:\TextbookPredSoilMapping\gis_data\slovakia_soil_points
_3857.gpkg' using driver `GPKG'
Simple feature collection with 5478 features and 2 fields
Geometry type: POINT
Dimension:      XY
Bounding box:   xmin: -589817.7 ymin: -1334011 xmax: -
166759.8 ymax: -1132904
Projected CRS:  S-JTSK / Krovak East North

# Display the first few rows and the structure of the
data
glimpse(soil_points)
Rows: 5,478
Columns: 3
$ WRB      <chr> "Cambisols", "Cambisols", "Cambisols",
"Cambisols", "Cambisols", "Cambisols", "Fluvisols",
"Fluvisols", "Flu...
$ SOC_t_ha <dbl> 124.4495, 121.8000, 124.4000, 109.1000,
77.7000, 113.4000, 83.6000, 83.6000, 415.8000, 37.8000,
66.4000, 210...
$ geom      <POINT [m]> POINT (-387872.3 -1132904), POINT
(-387579.4 -1135850), POINT (-387596.7 -1137717), POINT
(-390370.4 -...

```

The `glimpse()` function from the `dplyr` package provides a compact overview of our table. We see the columns: `profile_id` (unique profile ID), `WRB` (our target variable – soil type), and a geometry column containing the coordinates of each point.

Class Distribution Analysis

Before moving on to modeling, it's critical to understand what kind of data we're dealing with. For the classification problem, we are primarily interested in how many observations (profiles) fall on each

class (soil type). A significant imbalance, where some classes are represented by hundreds of points and others by only a few, can negatively affect the model's ability to recognize rare classes.

We can easily calculate the number of points for each soil type using the verb `count()` with `dplyr` and visualize the result.

```
# Count the number of occurrences for each soil type
soil_type_counts <- soil_points %>%
  st_drop_geometry() %>% # Drop geometry for non-spatial
  operations
  count(WRB, sort = TRUE)

print(soil_type_counts)
```

	WRB	n
1	Cambisols	1441
2	Rendzic Leptosols	737
3	Fluvisols	534
4	Planosols and Stagnosols	519
5	Mollic Fluvisols and Mollic Gleysols	452
6	Haplic Luvisols	355
7	Chernozems	351
8	Podzols	270
9	Calcaric Cambisols	261
10	Albic Luvisols	248
11	Arenosols	181
12	Andosols	57
13	Histosols	38
14	distinctly contaminated soils	24
15	Leptosols	10

```
# Visualize the class distribution
ggplot(soil_type_counts, aes(x = n, y = reorder(WRB, n)))
+
  geom_col(fill = "steelblue") +
  theme_bw() +
  labs(
    title = "Distribution of soil profiles by soil type",
    x = "Number of profiles",
    y = "Soil type (WRB)"
  )
```

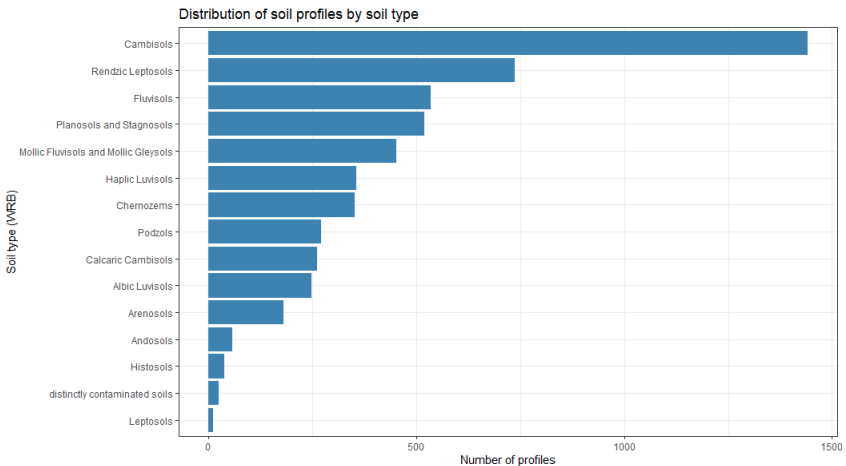


Fig. 2.5. Bar chart showing the number of observations for each soil type. Columns are sorted in ascending order for better readability

Graph analysis: We can see that the classes are unevenly distributed. The most represented are Cambisols and Rendzic Leptosols, which is typical for the mountainous and foothill areas of Slovakia. At the same time, soils such as Histosols are represented by a much smaller number of points. This **class imbalance** is an important data characteristic that will need to be taken into account during the modeling phase.

Spatial distribution analysis

Equally important is the analysis of **where** our points are located. Do they evenly cover the entire study area? Are there spatial clusters? We visualize the location of points on the map, coloring them according to the type of soil.

```
# Plot the spatial distribution of soil points
ggplot() +
  geom_sf(data = slovakia_boundary, child = "gray95") + #
  Base map
  geom_sf(data = soil_points, aes(color = WRB), size = 2,
alpha = 0.7) + # Soil points
  theme_bw() +
  labs(
```



```

title = "Spatial distribution of point data",
subtitle = "Points colored by soil type (WRB)",
color = "Soil type",
x = "Longitude", y = "Latitude" )

```

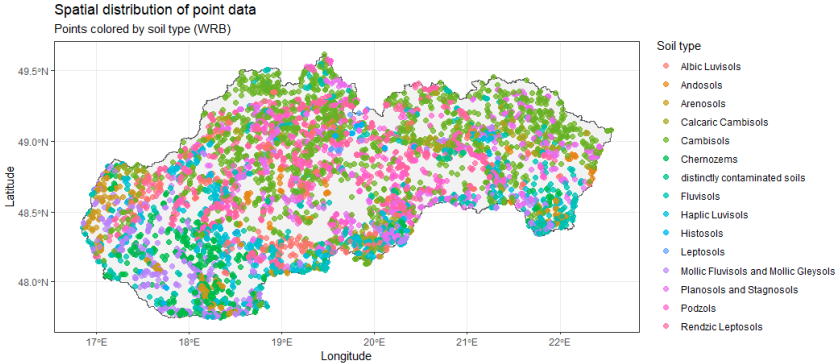


Fig. 2.6. Map of the spatial distribution of soil profiles on the territory of Slovakia. Each point corresponds to a profile and has a color indicating its type of soil

Map analysis: The map shows that the points are distributed over the area relatively unevenly, with obvious clusters in certain regions, which often happens with historical data. We can also see geographical patterns: for example, the Chernozems are predominantly concentrated in the southern lowlands, while the Cambisols dominate the mountainous areas. This visual analysis confirms that our data have a clear relationship with the landscape, and is a good sign for further predictive modeling.

7.3. Collection and pre-treatment of raster covariates

Once we have prepared the data for our target variable (**S**), it is time to collect and process the predictors – a set of raster layers that quantify the factors of soil formation according to the SCORPAN model. These raster covariates are the "eyes" of our model; it is through them that the model "sees" the landscape and learns to relate its characteristics to soil properties.

For our example with Slovakia, we will focus on obtaining a set of

covariates representing **R (Relief)** and **P (Parent Rock)**. Relief is the dominant factor at this scale, and we can obtain a large number of informative predictors from a single source – the Digital Elevation Model (DEM).

Step 1: Processing the DEM and obtaining relief derivatives

We will start with the DEM for the territory of Slovakia. Suppose we have it as a GeoTIFF file. Our first task is to download it and, using the powerful functions of the terra package (Hijmans, 2023), calculate a set of primary and secondary morphometric indicators.

```
# Load necessary libraries
library(terra)
library(sf)

# Load the base DEM for Slovakia
dem <- rast("gis_data/dmr3_5_100.tif")

# Calculate primary terrain derivatives using the
terrain() function
slope <- terrain(dem, v = "slope", unit = "degrees")
aspect <- terrain(dem, v = "aspect", unit = "degrees")

# Calculate some secondary derivatives
# TPI (Topographic Position Index) - indicates ridges vs
valleys
tpi <- terrain(dem, v = "TPI")
# TRI (Terrain Ruggedness Index) - measures local
elevation variation
tri <- terrain(dem, v = "TRI")

# It's good practice to combine all derived rasters into
a single multi-layer object
# and give them meaningful names
terrain_covariates <- c(dem, slope, aspect, tpi, tri)
names(terrain_covariates) <- c("elevation", "slope",
"aspect", "tpi", "tri")

# Plot one of the derivatives to check the result
plot(slope, main = "Surface slope, degrees")
```

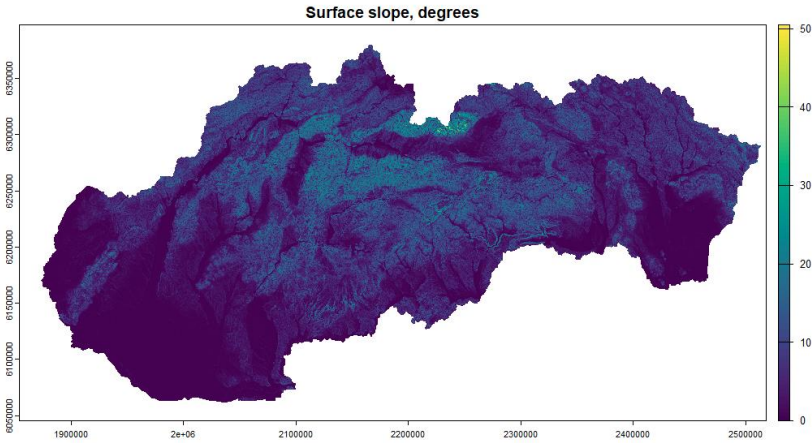


Fig. 2.7. Slope map calculated on the basis of DEM. The difference between the flat lowlands in the south and the Tatra mountain systems with steep slopes is clearly visible

Step 2: Preparation of other covariates (e.g. geology)

Suppose we have a rasterized geological map where each pixel has a numerical code corresponding to a specific type of rock. This is a **categorical raster** and its processing is slightly different, especially in the oversampling phase.

```
# Load the rasterized geological map
geology <- rast("gis_data/slovakia_geology.tif")
plot(geology)
```

Step 3: Raster Stack Harmonization

This is a **critical stage**. In order for the model to work, all raster predictors must have exactly the same spatial properties:

- **Coordinate System (CRS):** All layers must be in the same projection.

Extent: All layers must cover the same geographic area.

Resolution: All layers must have the same pixel size and be aligned to a single grid.

We use our DEM as a "template" and bring all other rasters to its

parameters. In addition, we will crop and mask the final stack along the exact contour of our research territory (`slovakia_boundary`) to avoid processing unnecessary data.

```
# Load the boundary polygon
slovakia_boundary <-
st_read("gis_data/slovakia_boundary.gpkg")

# --- Harmonization Workflow ---

# 1. Project geology raster to match the DEM's CRS
# Note: This step is only needed if CRSs are
# different. We assume they are for demonstration.
geology_proj <- project(geology, crs(dem), method =
"near") # Use 'near' for categorical data

# 2. Resample the projected geology raster to match the
# DEM's grid
geology_resampled <- resample(geology_proj, dem, method =
"near")

# 3. Combine all prepared rasters into a single stack
final_stack <- c(terrain_covariates, geology_resampled)
names(final_stack)[6] <- "geology" # Rename the last
layer

# 4. Crop and mask the final stack to the exact boundary
# of Slovakia
final_stack_masked <- crop(final_stack,
slovakia_boundary)
final_stack_masked <- mask(final_stack_masked,
slovakia_boundary)

# Check the properties of the final stack
print(final_stack_masked)
class      : SpatRaster
dimensions : 3174, 6382, 6  (nrow, ncol, nlyr)
resolution : 100, 100  (x, y)
extent     : 1873812, 2512012, 6062227, 6379627  (xmin,
xmax, ymin, ymax)
coord.ref. : WGS 84 / Pseudo-Mercator (EPSG:3857)
source(s)  : memory
varname     : dmr3_5_100
names      : elevation,      slope, aspect,      tpi,
tri, geology
```

```

min values : 94.41817, 0.00000, 0, -41.83284,
0.0000, 1
max values : 2609.17456, 53.46109, 360, 60.06293,
105.1039, 348

# Save the multi-layer raster to a GeoTIFF file
writeRaster(final_stack_masked,
"gis_data/all_slovakia_terrain_derivatives.tif",
overwrite = TRUE)

plot(final_stack_masked)

```

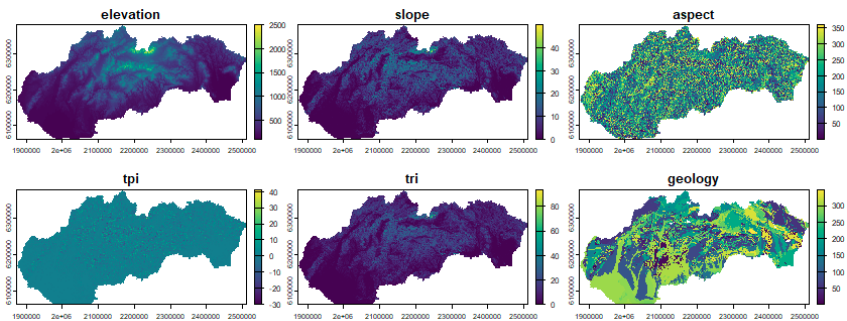


Fig. 2.9. Visualization of the final harmonized stack of raster covariates. Several layers are shown that overlap perfectly within the borders of Slovakia

Result Analysis: The output of the `print()` command will show us that `final_stack_masked` is a `SpatRaster` object with 6 layers. All of them now have identical sizes, resolution, extent, and coordinate system.

This careful preparation and harmonization process ensures that in the next step, when we extract the values of the covariate for our points, we will obtain consistent and reliable data, which is the key to building a quality predictive model.

7.4. Creating the final dataset for modeling

We have come to the culminating point in data preparation. In the previous sections, we have separately prepared two key ingredients:

- **Point Soil Data (`soil_points`):** Our `sf` (Pebesma, 2018) facility

containing the geographic location and soil type for each profile (S component).

- **Harmonized Covariate Stack (final_stack_masked):** Our multilayer SpatRaster object, where all predictors (elevation, slope, geology, etc.) are brought to a single spatial grid (**SCORPAN components**).

Now our task is to **integrate** these two data sets to create a single, "neat" table. This table, often referred to as a "modeling matrix" or "training matrix", is the final product that we will feed to machine learning algorithms. Each row in this table will represent one observation (ground profile), and the columns will represent the target variable and all the predictors for that observation.

Step 1: Extraction of covariate values

The main operation at this stage is **extraction** – the process of extracting pixel values from the raster stack at the exact locations of our soil profiles. The terra package performs this task extremely efficiently with the `extract()` function. It "pierces" the entire raster stack at each point and returns values from all layers.

```
# Load necessary libraries
library(terra)
library(sf)
library(dplyr)
library(tidyverse)

# Load the two key datasets prepared in previous sections
final_stack_masked <-
rast("gis_data/all_slovakia_terrain_derivatives.tif")

soil_points <-
st_read("gis_data/slovakia_soil_points_3857.gpkg")

Reading layer `slovakia_soil_points_3857' from data
source

`D:\TextbookPredSoilMapping\gis_data\slovakia_soil_points
_3857.gpkg' using driver `GPKG'
Simple feature collection with 5478 features and 2 fields
Geometry type: POINT
Dimension:      XY
```

```

Bounding box:  xmin: 1875992 ymin: 6063841 xmax: 2510017
ymax: 6379423
Projected CRS: WGS 84 / Pseudo-Mercator

# --- Data Integration Workflow ---

# Ensure the CRS of points matches the raster stack
before extraction
# This is a crucial sanity check
soil_points <- st_transform(soil_points, crs =
crs(final_stack_masked))

# 1. Extract covariate values for each point
# The result is a data frame
# The xy=TRUE argument tells the function to include the
coordinates in the output

extracted_covariates <- extract(final_stack_masked,
soil_points, xy = TRUE)

# Let's inspect the result
head(extracted_covariates)
> head(extracted_covariates)
  ID elevation   slope  aspect    tpi    tri geology      x      y
1  1  1163.4471  1.075246  172.6339  5.2965088  5.397705    NA 2166262 6379377
2  2  1046.7526  9.630545  249.1020  3.1379776 13.429176   177 2167062 6374877
3  3   981.1536  2.606068  139.7034  0.9097672  4.190346   177 2167262 6372077
4  4   749.1828  3.654978  306.3462 -0.1509094  4.732956   174 2163162 6369177
5  5   735.6061  5.938074  48.4185 -0.5148773  7.496658   174 2162862 6364077
6  6    800.6909  7.334056 125.8008  1.4717636  9.644859   177 2164562 6370577

```

Fig. 2.9. Results of the extraction process. Vector points "punctured" the multilayer raster stack by extracting values from each layer and writing them to a new table

As you can see, `extracted_covariates` is a table where the first column ID corresponds to the ordinal number of the point, and the rest of the columns contain the values of the predictors for each of them.

Step 2: Merge and Final Data Cleanup

Now we need to combine the output data from the points (containing, most importantly, our target WRB variable) with a table of extracted values. We'll also follow a few important cleanup steps to prepare the data directly for the simulation.

```
# 2. Combine original point data with extracted values
```

```
# We use bind_cols() as it joins dataframes side-by-side
# We also convert the sf object to a regular dataframe
for modeling
final_dataset <-bind_cols(st_drop_geometry(soil_points),
extracted_covariates)

# 3. Clean up the combined dataset
final_dataset_clean <-final_dataset%>%
  #select(-ID) %>% # Remove the redundant ID column from
extract()
  na.omit() %>% # Remove rows with NA values (e.g.,
points outside the raster mask)
  mutate(
    # Convert the numeric geology code into a factor
    # This is crucial for models to treat it as a
categorical variable
    geology = as.factor(geology)
  )

# Let's inspect the final, clean dataset
glimpse(final_dataset_clean)

# And write the final_dataset_clean dataframe to a CSV
file in the 'results' folder

write_csv(final_dataset_clean,
"results/final_modeling_dataset.csv")
```

```
> glimpse(final_dataset_clean)
Rows: 5,443
Columns: 10
$ WRB      <chr> "Cambisols", "Cambisols", "Cambisols", "Cambisols", "Cambisols", "Fluvisols", "Fluvisols", "Fluvisols...
$ soc_t_ha  <dbl> 121.8000, 124.4000, 109.1000, 77.7000, 113.4000, 83.6000, 83.6000, 37.8000, 66.4000, 210.05...
$ elevation <dbl> 1046.7526, 981.1536, 749.1828, 735.6061, 800.6909, 726.9199, 705.5182, 692.1104, 572.7368, 692.0603, ...
$ slope     <dbl> 9.6305447, 2.6060677, 3.6549778, 5.9380741, 7.3340559, 2.9390810, 0.6124931, 0.3324119, 8.9306774, 11...
$ aspect    <dbl> 249.102036, 139.703445, 306.346161, 48.418499, 125.800797, 63.980671, 97.755150, 70.102440, 301.94018...
$ tpi       <dbl> 3.13797760, 0.90976715, -0.15090942, -0.51487732, 1.47176361, 0.94549561, -0.55107117, -1.25041199, -...
$ tri       <dbl> 13.429176, 4.190346, 4.732956, 7.496658, 9.644859, 4.043839, 1.252975, 1.333267, 11.946732, 15.563187...
$ geology    <fct> 177, 177, 174, 174, 177, 174, 174, 177, 63, 63, 167, 177, 60, 60, 174, 174, 174, 174, 174, 177, 60, 6...
$ x         <dbl> 2167062, 2167262, 2163162, 2162862, 2164562, 2161662, 2162162, 2162562, 2067962, 2063262, 2066162, 20...
$ y         <dbl> 6374877, 6372077, 6369177, 6364077, 6370577, 6367777, 6367977, 6365977, 6352377, 6348677, 6358877, 63...
```

Analysis of the result: The `glimpse()` command shows us the ideal structure for modeling. We have a table where each row is a complete set of data. The first column (WRB) is our raw information, the second (SOC_t_ha) is the SOC stocks at a given point (we will model it in Section 3 as a continuum variable) and the next columns (elevation, slope, ..., geology) are predictors ready to use. that there are no missing values (NA) in the data, and the categorical predictor geology has the

correct factor type. The coordinates of the points are also saved, which is important for visualization.

This final object is `final_dataset_clean` the result of all our preparatory work and is saved to disk. We have come a long way from scattered raw data to a single, coherent, and informative dataset. We are now fully prepared to move on to the next section and use this set to train, validate, and apply machine learning models to create our first predictive soil map.

Chapter 8. Modeling with decision trees and random forest

8.1. Introduction to Machine Learning for Classification

In the previous section, we successfully completed perhaps the most difficult stage of any digital soil mapping project – data preparation and integration. The result of our work is a single, "tidy" table of `final_dataset_clean` containing both a target variable (soil type) and a set of potential predictors (covariate) for each observation point. Now we are ready to move on to the most interesting part – building a predictive model. **Machine Learning (ML).**

Machine learning is a branch of artificial intelligence that gives computers the ability to "learn" from data without being explicitly programmed for a specific task (Breiman, 2001; James et al., 2013). In the context of DSM, ML algorithms are the "engine" that analyzes our integrated data and identifies complex, often non-linear and non-human patterns linking environmental factors (CORPAN) to soil (S) properties.

The task that we will solve in this part of the book belongs to the category of **supervised learning**. It is called "controlled" because we have the "right answers" for our training data – for each soil profile, we know its true soil type. We essentially act as a "teacher" who shows the algorithm examples (point data) and correct answers (class labels), and the algorithm's task is to learn the general rules so that we can then give the correct answers for new, never seen examples (i.e., for each pixel of our map).

Within guided learning, there are two main types of tasks:

- **Regression:** Prediction of a continuous, quantitative variable (e.g., organic carbon content, pH). We will discuss this topic in detail in Part III.

Classification: Predicting a categorical variable or **class**. This is our current goal – we want to train the model to assign each pixel on the map one of the defined classes corresponding to soil types (e.g. Cambisol, Chernozem, Luvisol).

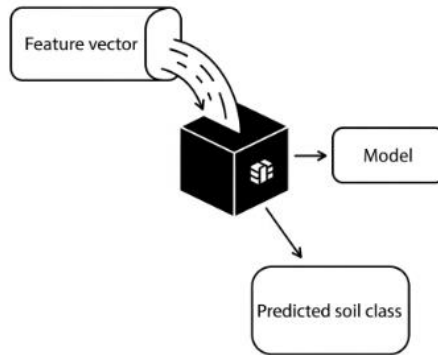


Fig. 2.10. Conceptual scheme of the classification problem. A feature vector (covariate value for one pixel) is applied to the model input. The model, which is a "black box" with learned rules, outputs a predicted class

There are many classification algorithms, from simple (logistic regression) to very complex (neural networks). In this guide, we'll focus on two extremely popular, powerful, and, importantly, intuitive methods that have proven themselves in digital soil science:

- **Decision Trees:** A simple, interpreted method that builds a set of hierarchical if-and-then rules that visually resemble a tree-like structure.
- **Random Forest:** A refinement of decision trees that uses the "wisdom of the crowd". This method builds not one, but hundreds of different decision trees on random subsamples of data, and makes the final prediction by "voting" between all the trees. This is one of the most reliable and accurate "ready-to-use" classifiers.

Before we move on to the practical implementation of these models, it is important to mention the key principle of validation outlined in subsection 6.3: for an objective assessment of accuracy, we must divide our final `final_dataset_clean` data set into **training** and **test** samples. The model will be built exclusively on the training sample, and we will test its performance on a test sample that simulates the operation of the model on completely new data.

8.2. Decision trees (rpart)

The first classification algorithm we will look at is **the Decision Tree**. Its enormous popularity, especially in applied sciences, is due not so much to its extreme accuracy as to its incredible simplicity and **interpretation**. Unlike many "black boxes" in machine learning, the decision tree produces the result in the form of a set of simple, hierarchical "if-so" rules that are easy to understand and visualize. forecast, but also to understand **why** the model made such a choice, which is extremely valuable for a soil scientist.

Imagine trying to determine the type of soil by asking successive questions about environmental factors. "Is this point at an altitude of more than 800 meters?". If yes, then it is probably "Podzol". If "no", then the next question is: "Is the mother breed a loess?". If yes, then it is probably "Chernozem". It is on this principle that the decision tree works.

The algorithm that builds such a tree (the most famous of which is CART) works on the principle **of recursive partitioning**.

- It starts with the entire set of training data (root node).

Then it goes through all the predictors (height, slope, etc.) and all possible points of their division to find **the best division** – the one that divides the data into two most "pure" groups. "Purity" means that one class of soil predominates in each of the newly formed groups.

This division process is recursively repeated for each new subgroup, creating branches and new nodes.

A tree stops growing when one of the stop criteria is met (e.g., a node becomes completely "clean", or there are too few observations left in it). Terminal nodes that are not further divided are called **leaves** and contain the final class forecast.

Practical implementation with rpart

In R, the classic package for building decision trees is rpart (Recursive Partitioning and Regression Trees, Therneau et al., 2022). We use it to build a model on our data.

Step 1: Splitting the data into training and test samples

First of all, we must divide our set of final_dataset_clean into two

parts. This is a critical step for an objective assessment of the model. We will use a modern approach from the `rsample` package (part of the `tidymodels` ecosystem). We will also remove from the dataset

```
# --- Load Necessary Libraries ---

# List of required packages
packages <- c("rpart", "rpart.plot", "rsample", "dplyr")

# Loop through the packages
for (pkg in packages) {
  # Check if the package is not already installed
  if (!require(pkg, character.only = TRUE)) {
    # If not installed, install it
    install.packages(pkg)
    # Load the package after installation
    library(pkg, character.only = TRUE)
  }
}

# Assume 'final_dataset_clean' is loaded from the
previous chapter
# Load your dataset from a CSV file
final_dataset <-
read_csv("results/final_modeling_dataset.csv")

# Remove the 'SOC_t_ha' column
final_dataset_clean <- final_dataset %>%
select(-SOC_t_ha)

# Convert the numeric WRB and geology after loading into
a factor
final_dataset_clean$WRB <-
as.factor(final_dataset_clean$WRB)
final_dataset_clean$geology <-
as.factor(final_dataset_clean$geology)

# Set a seed for reproducibility of the random split
set.seed(123)

# Create a split object that defines how to split the
data (e.g., 75% for training)
data_split <- initial_split(final_dataset_clean, prop =
0.75, strata = WRB)
```

```

# Extract the training and testing sets from the split
object
train_data <- training(data_split)
test_data  <- testing(data_split)

# --- Prepare Data ---
# It's still crucial to ensure the target variable is a
factor for classification
# and to remove any unique identifier columns like
'profile_id' before training.
train_data_clean <- train_data %>%
  select(-ID) # Remove identifier column if it exists

# We must apply the same cleaning to the test data before
prediction!
test_data_clean <- test_data %>%
  select(-ID)

train_data_clean

test_data_clean

# Check the dimensions
dim(train_data_clean)
[1] 4082    9

dim(test_data_clean)
[1] 1361    9

```

```
> train_data_clean
```

```
# A tibble: 4,082 x 9
```

	WRB	elevation	slope	aspect	tpi	tri	geology	x	y
	<fct>	<dbl>	<dbl>	<dbl>	<dbl>	<dbl>	<fct>	<dbl>	<dbl>
1	Cambisols	1047.	9.63	249.	3.14	13.4	177	2167062.	6374877.
2	Cambisols	981.	2.61	140.	0.910	4.19	177	2167262.	6372077.
3	Cambisols	749.	3.65	306.	-0.151	4.73	174	2163162.	6369177.
4	Cambisols	801.	7.33	126.	1.47	9.64	177	2164562.	6370577.
5	Fluvisols	706.	0.612	97.8	-0.551	1.25	174	2162162.	6367977.
6	Fluvisols	692.	0.332	70.1	-1.25	1.33	177	2162562.	6365977.
7	Cambisols	573.	8.93	302.	-1.33	11.9	63	2067962.	6352377.
8	Cambisols	692.	11.3	299.	1.01	15.6	63	2063262.	6348677.
9	Cambisols	783.	8.99	175.	2.33	12.2	167	2066162.	6358877.
10	Fluvisols	467.	3.14	116.	-2.70	4.06	60	2097462.	6361477.

```
# i 4,072 more rows
```

```
# i use `print(n = ...)` to see more rows
```

```
> test_data_clean
# A tibble: 1,361 × 9
  WRB      elevation slope aspect   tpi   tri geology      x      y
  <fct>      <dbl>   <dbl>   <dbl>   <dbl> <dbl> <fct>      <dbl>   <dbl>
1 Cambisols    736.    5.94    48.4  -0.515  7.50 174    2162862.  6364077.
2 Fluvisols    727.    2.94    64.0   0.945  4.04 174    2161662.  6367777.
3 Fluvisols    488.    1.85    351.  -4.16   5.10 177    2098362.  6360977.
4 Fluvisols    465.    4.25    194.   0.681  5.83  60    2095762.  6360077.
5 Cambisols    824.    4.88    150.   1.48   6.59 174    2155362.  6355477.
6 Cambisols    502.    6.30    278.   0.591  8.55 177    2091962.  6353677.
7 Cambisols    563.    4.18    307.   5.10   8.12 177    2098462.  6353977.
8 Cambisols    422.    6.24    343.  -5.25   8.52 276    2099162.  6346477.
9 Cambisols    687.    6.39    64.7  -0.895  8.62 207    2174962.  6352977.
10 Cambisols   828.    5.29    254.  -1.44   7.14 207    2164662.  6352177.
# i 1,351 more rows
# i Use `print(n = ...)` to see more rows
```

Note: The strata = WRB argument is very important. It ensures that the proportions of different soil types in the study and test samples are the same as in the original dataset, which is critical when working with unbalanced classrooms.

Step 2: Model Training

Now we can train the model to train_data. The main function `rpart()` uses formula syntax, where Target_variable ~ Predictor1 + Predictor2 + Period (.) is an abbreviation for "all other variables".

```
# Train the decision tree model
# We want to predict WRB using all other variables as
predictors
# We want to predict WRB using all other variables EXCEPT
'geology'
tree_model <- rpart(WRB ~ . - geology, data =
train_data_clean, method = "class")
```

After a rather long wait (classification is a resource-intensive process), we get the result.

Step 3: Rendering and Analyzing the Tree

The main advantage of this method is visualization. The `rpart.plot` package provides great tools for this.

```
# Plot the resulting decision tree
rpart.plot(tree_model, box.palette = "RdBu", shadow.col =
```

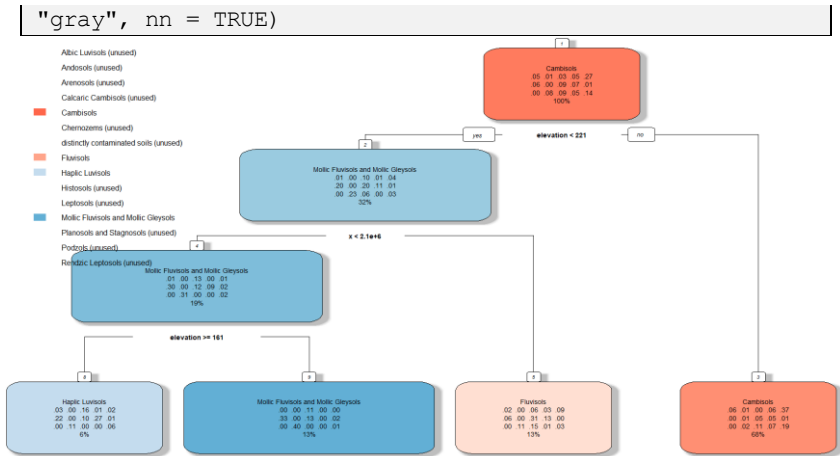


Fig. 2.11. Visualization of the decision tree for the classification of soil types. The tree shows a hierarchy of division rules, starting with the most important predictor in the root node

- **Graph analysis:** This tree is a visual instruction manual for soil classification.
- **Root node (top):** Shows the overall distribution of classes in the study sample. The first rule of separation of elevation < 221m. This means that height is the most important predictor that best divides soils into two groups.
- **Branches:** Objects that satisfy the condition go along the left branch ("yes"), the rest go along the right ("no").
- **Nodes and leaves:** Each node shows a predicted class for the objects that have fallen into it, and the distribution of classes as a percentage. For example, we can see a node where 68% of soils are "Cambisols". This is our rule: the combination of conditions leading to this node is characteristic of Cambisols.

Although decision trees are a great tool for interpreting and understanding data, they have drawbacks: they can be unstable (small changes in the data can drastically change the structure of the tree) and prone to **overlearning** (creating too complex rules that work perfectly on training data, but do not generalize well on new ones). It was to solve these problems that the Random Forest method was created, which we will consider in the next subsection.

8.3. RandomForest

In the previous subsection, we saw how intuitive and interpretable decision trees are. However, we also noted their significant drawbacks: high variance (instability) and a tendency to overlearn. This means that a single tree built on our data is only one of many possible ways to describe dependencies, and it may be too "fitted" to the specific noises and features of our particular study sample.

Random Forest is an ingenious solution to these problems. It is a **method of ensemble learning** that is based on a simple but powerful idea of "crowd wisdom": a collective solution of a large number of diverse but relatively weak models (individual trees) will be much more accurate and reliable than the solution of a single, albeit complex, model (Breiman, 2001). Instead of carefully "growing" one perfect tree, we create an entire "forest" of hundreds or thousands of different trees. And then we force them to "vote" for the final forecast.

The Random Forest algorithm achieves this "diversity" of trees using two key techniques:

- **Bagging:** Each tree in the forest is not built on the entire training dataset, but on its **random subsample with a return (bootstrap sample)**. This means that a new dataset of the same size as the original one is created for each tree, but some observations in it may be repeated several times, and some may not hit at all. That each tree sees a slightly different "picture" of data.
- **Randomness of predictors:** When plotting each node in each tree, the algorithm does not iterate through all available predictors to find the best division. Instead, it only considers a **random subset of predictors** (e.g., 3 out of 10). This forces trees to use different variables and prevents a situation where one very strong predictor (e.g., elevation) would dominate all trees, making them similar.

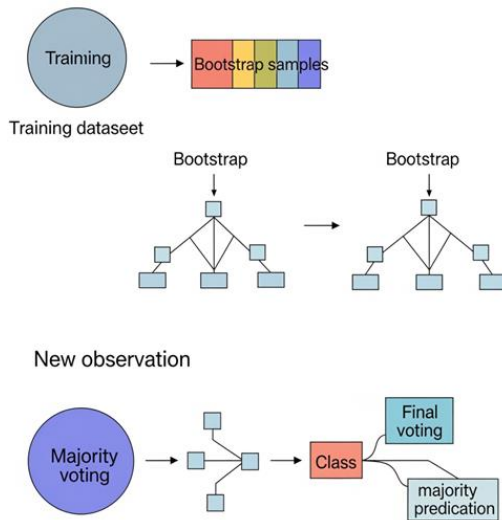


Fig. 2.12. Conceptual scheme of operation of the Random Forest for classification. It is shown how many bootstrap samples are created from the training data, each is built with its own tree, and the final forecast for the new observation is determined by voting

Practical implementation with randomForest

The classic package for working with this algorithm in R is `randomForest`.

Step 1: Model Training

We will use the same `train_data` and `test_data` as in the previous subsection. The syntax of the `randomForest()` function is very similar to that of `rpart()`.

```
# --- Load Necessary Libraries ---
# We will use 'ranger' for modeling and 'caret' for the
# confusion matrix.
# This code will also install them if they are not
# already present.
packages <- c("ranger", "dplyr", "caret", "rsample")
for (pkg in packages) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg)
    library(pkg, character.only = TRUE)
  }
}
```

```

    }
  }

  # We continue using the train_data from the previous
  split
  # Set a seed for reproducibility of the random process in
  the algorithm
  set.seed(123)

  # Train the Random Forest model
  # ntree: number of trees to grow
  # mtry: number of variables randomly sampled as
  candidates at each split
  rf_model <- ranger(
    formula = WRB ~ .,
    data = train_data_clean,
    num.trees = 500,
    importance = 'permutation' # A robust method for
    calculating variable importance
  )

  # Print the model summary
  print(rf_model)

  # --- Save the Trained Model ---

  # 1. Create the 'models' directory if it doesn't already
  exist.
  if (!dir.exists("models")) {
    dir.create("models")
  }

  # 2. Save the ranger_model object to the specified file.
  # The .rda format is a standard R data file format.
  save(rf_model, file = "models/rf_model.rda")

  print("Model has been successfully saved to
  models/rf_model.rda")

  # Create and view the Confusion Matrix ---

  # 1. Make predictions on the unseen test data.
  predictions <- predict(rf_model, data = test_data)

  # 2. Use the confusionMatrix() function from the 'caret'
  package.

```

```
# We compare the model's predictions with the actual
true values.
conf_matrix <- confusionMatrix(
  data = predictions$predictions,      # The predicted
  classes
  reference = test_data$WRB          # The true classes
)

# 3. Print the detailed confusion matrix and all
associated statistics.
print(conf_matrix)
```

Output analysis: Unlike `rpart`, `print(rf_model)` output does not show rules. Instead, it provides extremely useful information:

- **Forest type:** Classification.
- **Number of trees:** 500.
- **OOB estimate of error rate:** 47,99%. **OOB (Out-of-Bag) error** is a built-in validation mechanism. For each tree, about a third of the output data does not make it into the bootstrap sample. An OOB error is an average error across all trees, calculated on data that was not used to build them. This is a reliable and objective indicator of model accuracy.

```
> print(rf_model)
Ranger result

Call:
ranger(formula = WRB ~ ., data = train_data_clean, num.trees = 500,      importance = "permutation")

Type:                Classification
Number of trees:     500
Sample size:         4082
Number of independent variables: 8
Mtry:                2
Target node size:    1
variable importance mode: permutation
Splitrule:           gini
OOB prediction error: 47.99 %
```

- **Confusion Matrix:** Shows how the model has classified OOB data, allowing you to see which classes of soils are confused with each other.

```
> print(conf_matrix)
Confusion Matrix and Statistics
```

Prediction	Reference							
	Albic Luvisols	Andosols	Arenosols	Calcaric Cambisols	Cambisols	Chernozems		
Albic Luvisols	17	0	0	6	2	0		
Andosols	0	6	0	0	1	0		
Arenosols	0	0	19	0	2	3		
Calcaric Cambisols	2	0	0	12	3	1		
Cambisols	14	9	3	38	271	1		
Chernozems	0	0	7	0	0	41		
distinctly contaminated soils	0	0	0	0	0	0		
Fluvisols	2	0	7	2	5	8		
Haplic Luvisols	7	1	2	6	10	10		
Histosols	0	0	0	0	0	0		
Leptosols	0	0	0	0	0	0		
Mollic Fluvisols and Mollic Gleysols	0	0	8	0	2	24		
Planosols and Stagnosols	7	0	0	7	17	0		
Podzols	0	0	0	0	8	0		
Rendzic Leptosols	5	0	0	0	31	0		

Step 2: Assessing the importance of variables

One of the most powerful advantages of the Random Forest is its ability to estimate **the importance of predictors (variable importance)**. The algorithm calculates how much each covariate on average contributes to the accuracy of the model across the forest. This allows us to understand which SCORPAN factors are key to the distribution of soils in our area.

```
# --- Get and plot Variable Importance from ranger model
---

# 1. Extract the importance scores from the model object.
# The importance() function works, but we need to
# handle its output.
importance_scores <- importance(rf_model)

# 2. Convert the named vector of scores into a data frame
# for plotting with ggplot2.
importance_df <- data.frame(
  Variable = names(importance_scores),
  Importance = importance_scores
) %>%
  # Arrange the variables by importance for a cleaner
  # plot
  arrange(Importance) %>%
  mutate(Variable = factor(Variable, levels = Variable))
# This keeps the sorted order in the plot

# 3. Create the variable importance plot using ggplot2.
# This is the modern equivalent of varImpPlot().
ggplot(importance_df, aes(x = Importance, y = Variable))
+
```

```
geom_col(fill = "steelblue") +
theme_bw() +
labs(
  title = "Variable Importance for Soil
Classification",
  x = "Importance (Permutation)",
  y = "Predictor"
)
```

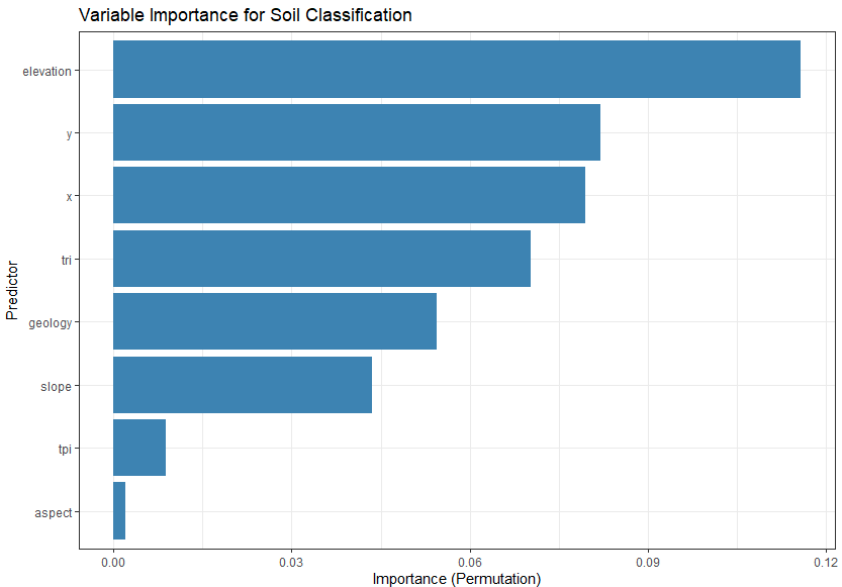


Fig. 2.13. Graph of the importance of variables. A bar chart is shown where predictors are sorted by their contribution to model accuracy

Graph analysis: This graph is key to interpreting the results. We can clearly see which predictors are "moving" in our model. For example, elevation, coordinates, tri and geology are the most important, which makes complete soil science sense. Instead, some of the indicators, such as aspect, can have a much smaller impact. This allows not only to build an accurate model, but also to get a scientific idea of the hierarchy of soil formation factors in the study area.

The random forest is a significant step forward compared to a single decision tree, offering higher accuracy and stability while maintaining

the possibility of deep analysis through the evaluation of the importance of variables.

In this section, we use the Random Forest for **classification**. It is worth noting that there is a powerful modification of it for regression problems, which not only predicts continuous values, but also allows us to estimate the uncertainty of these predictions. This method, known as **Quantile Regression Forests**, will be discussed in detail in Part III when we model the organic carbon content.

Chapter 9. Accuracy Assessment and Validation of Classification Models

After building our first machine learning models, we may be tempted to jump straight to mapping. However, this step would be premature and scientifically unsound. Before we can trust the model's predictions, we must conduct careful and objective **validation** – the process of evaluating how well the model performs on new, never-before-seen data (Congalton, 1991)

For validation, we will use a **test sample (test_data)**, which we prudently postponed at the very beginning. Since the model has never "seen" this data during its training, it is a perfect simulator of a real-world situation when the model encounters new data.

9.1. Confusion Matrix

The cornerstone of evaluating the accuracy of any classification model is **the mismatch matrix**, also known as the confusion matrix. This is a simple but extremely informative table that matches the **classes predicted by the model** with **the real (true) classes** from the test sample. It allows us to see not only the total number of errors, but also, much more importantly, *what kind of* errors these are.

Structure of the matrix of mismatches:

- **The strings** usually represent **true classes** (data from our observations).
- **The columns** represent **the predicted classes** (what the model "said").
- **The diagonal elements** (where true class = predicted class) show the number of **correctly classified** observations.
- **Extradiagonal elements** show **errors** - cases when the model "confused" one class with another.

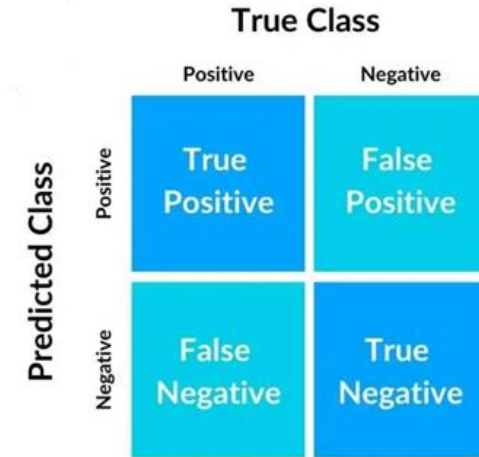


Fig. 2.14. Structure of the matrix of mismatches.

Practical implementation and interpretation

We create a matrix of inconsistencies for our Random Forest (`rf_model`) model, which we taught in the previous section.

Step 1: Make predictions on the test sample

First, we'll use the `predict()` function to get our model's predictions for `test_data`.

```
# Create and view the Confusion Matrix ---

# Make predictions on the unseen test data.
predictions <- predict(rf_model, data = test_data_clean)
```

Step 2: Creating and analyzing the matrix

Now, having a true value vector (`test_data$WRB`) and a prediction vector, we can create a matrix. The `confusionMatrix()` function from the `caret` package (Kuhn, 2008) is ideal for this, since it automatically calculates not only the matrix itself, but also a whole set of key accuracy metrics.

```
# Create the confusion matrix and associated statistics
# Use the confusionMatrix() function from the 'caret'
package.
# We compare the model's predictions with the actual
true values.
conf_matrix <- confusionMatrix(
  data = predictions$predictions,      # The predicted
  classes                                # The true classes
  reference = test_data_clean$WRB
)

# Print the detailed confusion matrix and all associated
statistics.
print(conf_matrix)
```

Output analysis: The output of `confusionMatrix()` is very detailed. First, we see the table itself, and below it is a block with statistics.

- **Overall Accuracy:** This is the simplest metric – the proportion of correctly classified observations. Although it is intuitive, it can be misleading when dealing with unbalanced data.
- **Kappa (Kappa Cohen Coefficient):** This is a much more reliable metric than general accuracy. Kappa shows how much better classification results are than random guessing results (Cohen, 1960). It takes into account the probability of random correct classification. Kappa values range from -1 to 1, where 1 is perfect consistency, 0 is random consistency, and negative values are worse than randomness.
 - < 0.2: Weak consistency
 - 0.2 - 0.6: Moderate consistency
 - 0.6 - 0.8: Substantial consistency
 - > 0.8: Near-perfect consistency
- **Statistics by Class:** `confusionMatrix()` also calculates metrics for each class separately, which is extremely important. The most important of them are **Sensitivity** and **Precision**, which in soil science are often called **Producer's Accuracy** and **User's Accuracy**, respectively.

The matrix of mismatches and the Kappa coefficient are fundamental tools for any classification task. They allow you to move from a simple statement "the model is 85% accurate" to a deep understanding of its

strengths and weaknesses: which classes it recognizes well, which ones it confuses, and how much its results are better than ordinary guessing.

9.2. Overall Accuracy Metrics (Producer's Accuracy, User's Accuracy)

The matrix of inconsistencies that we obtained in the previous subsection is a source for the calculation of a whole series of quantitative indicators that allow an objective and comprehensive assessment of the performance of our model (Congalton, 1991). The `confusionMatrix()` function from the `caret` package kindly calculates them for us, but for a deep understanding of the strengths and weaknesses of our map, we must be clear about what is behind each of these numbers. We consider the three most important groups of metrics.

```
# --- Extract Specific Accuracy Metrics ---

# 1. Extract Overall Accuracy from the confusion matrix
object.
# It's stored in the 'overall' component.
overall_accuracy <- conf_matrix$overall['Accuracy']

print("--- Overall Model Accuracy ---")
print(paste("Overall Accuracy:", round(overall_accuracy,
4)))
[1] "Overall Accuracy: 0.5195"

# 2. Extract Producer's and User's Accuracy.
# These are stored in the 'byClass' component of the
object.
accuracy_by_class <- as.data.frame(conf_matrix$byClass)

# Let's select and rename the relevant columns for
clarity.
# 'Sensitivity' is Producer's Accuracy.
# 'Pos Pred Value' is User's Accuracy.
# NOTE: Column names with spaces must be enclosed in
backticks (`).
class_accuracies <- accuracy_by_class %>%
  select(Sensitivity, `Pos Pred Value`) %>%
  rename(
```

```

    Producer_Accuracy = Sensitivity,
    User_Accuracy = `Pos Pred Value`
)

print("--- Accuracy Metrics by Class ---")
print(class_accuracies)

```

```

> print(class_accuracies)

```

	Producer_Accuracy	User_Accuracy
Class: Albic Luvisols	0.31481481	0.4594595
Class: Andosols	0.37500000	0.8571429
Class: Arenosols	0.41304348	0.5757576
Class: Calcaric Cambisols	0.16901408	0.4444444
Class: Cambisols	0.76988636	0.5113208
Class: Chernozems	0.46590909	0.5256410
Class: distinctly contaminated soils	0.00000000	NaN
Class: Fluvisols	0.41481481	0.4955752
Class: Haplic Luvisols	0.47058824	0.4000000
Class: Histosols	0.08333333	1.0000000
Class: Leptosols	0.00000000	NaN
Class: Mollic Fluvisols and Mollic Gleysols	0.57943925	0.5166667
Class: Planosols and Stagnosols	0.41304348	0.4634146
Class: Podzols	0.37878788	0.6756757
Class: Rendzic Leptosols	0.55555556	0.6451613

Overall Accuracy

This is the simplest and most intuitive metric. It is calculated as the ratio of the number of all correctly classified samples (the sum of the diagonal elements of the matrix) to the total number of samples in the test sample.

Overall Accuracy = $\frac{\text{Total Predictions} \times \text{Number of Correct Predictions}}{\text{Total Predictions}}$

Interpretation: This metric answers a simple question: "What percentage of samples from the test sample did our model classify correctly?". While this metric is useful for the overall impression, it can be **misleading**, especially when dealing with **unbalanced data**. Imagine that 90% of our territory is occupied by Cambisols. A model that simply always predicts Cambisol will have an overall accuracy of 90%, although it is completely helpless in determining all other, less common, but perhaps more important types of soil.

Producer's Accuracy

This metric, also known in machine learning as **Sensitivity** or **Recall**, evaluates accuracy from the point of view of the "maker" of the map. It

is calculated for each class separately.

Producer's Accuracy (for Class A)=Total Number of True Class A Samples/Number of Samples Correctly Classified as Class A.

Interpretation: This metric answers the question: "Of all the true samples of a certain type of soil that exist on the ground (in our test sample), what percentage was our map able to correctly identify?". This is an indicator of how well the map "finds" a particular class. Low accuracy of the manufacturer means that the model misses many samples of this class, mistakenly attributing them to others (*omission errors*) matrices of inconsistencies (Congalton, 1991).

User's Accuracy

This metric, known as **Precision in ML**, evaluates the quality of a map from the point of view of the end "user". It is also calculated for each class.

User's Accuracy (for Class A)=Total number of samples assigned Class A by the model/Number of samples correctly classified as Class A.

Interpretation: This metric answers the practical question: "If I go to a point that the map marks as a certain type of soil, what is the probability that there is actually that type of soil there?". This is an indicator of the reliability of the forecast on the map. Low user accuracy means that the map includes within the boundaries of a certain class many areas that actually belong to other classes (*inclusion errors, commission errors*). The calculation is carried out **according to the lines** of the matrix of discrepancies.

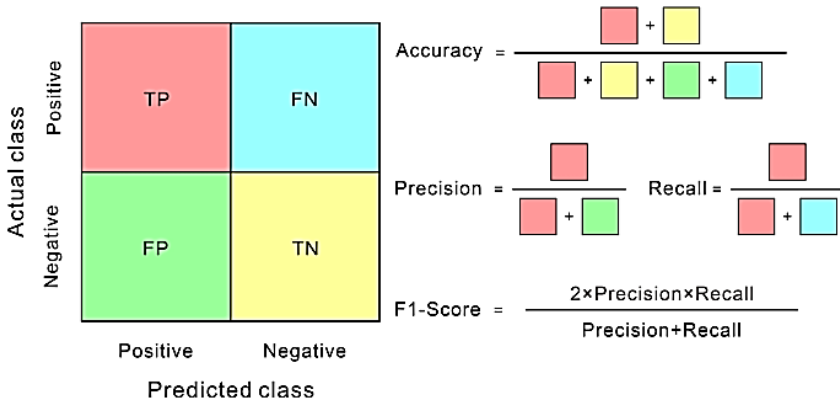


Fig. 2.15. Accuracy calculation scheme adopted from Zhong et al. (2023).

Analysis: There is a trade-off between these two metrics. The model can have high producer accuracy for Chernozems (find almost all true Chernozems), but low user accuracy (while mistakenly attributing many other soils to Chernozems).

That is why the accuracy analysis by classes provided by `confusionMatrix()` is much more informative than a single digit of total accuracy. It allows us to diagnose exactly which types of soils our model predicts well and which have problems, which allows us to return to the stage of selecting covariate or adjusting the model to improve the results.

9.3. Kappa coefficient

In the previous section, we took a closer look at Overall Accuracy, as well as manufacturer's and user's accuracy. We have found that overall precision, while intuitive, can be misleading, especially when the classes in our data are unbalanced. A model can achieve high overall accuracy by simply "guessing" the most common class while ignoring all rare but important classes. We need a metric that can account for this effect and estimate how much the performance of our model is better than simple random guessing.

This is exactly why **Cohen's Kappa Coefficient**, or simply Kappa, **was developed**. This is a statistical indicator that measures the degree of agreement between two estimators (in our case, between model

predictions and true data), while taking into account the probability that this consistency arose **by chance**.

The concept of random consistency

Imagine that you give two soil scientists who know nothing about the territory a set of samples and ask them to assign each sample one of the possible soil types at random. Even with a completely random selection, purely according to the law of probability, some of their classifications will match. This is **random consistency**. Kappa answers the question: "How much is the consistency of our model with reality higher than this basic one, random consistency?".

The formula for calculating Kappa is as follows:

$$\text{Cohen's Kappa} = \frac{\text{Accuracy} - \text{RAccuracy}}{1 - \text{RAccuracy}}$$

where:

- **Accuracy** is an observable consistency that is nothing more than our **Overall Accuracy**.
- **RAccuracy** is a hypothetical probability of random consistency. It is calculated based on the row and column totals of our discrepancy matrix.

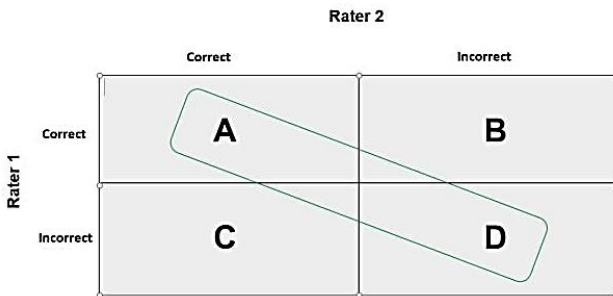


Fig. 2.16. N x N grid used to interpret results of raters ([Image: Kurtis Pykes](#))

Interpretation of Kappa Meanings

The Kappa value ranges from -1 to +1.

- kappa=1: Perfect, complete consistency.
- Kappa 0-1: Consistency is better than random.

- $\kappa=0$: The consistency is exactly at the randomness level. The model does not have any predictive power.
- $\kappa \leq 0$: Consistency is worse than random (which is rare, but indicates serious problems with the model).

To interpret the strength of coherence, the scale of Landis and Koch (1977) is often used:

Meaning of Kappa	Interpretation
< 0.00	Weak consistency
$0.00 - 0.20$	Barely noticeable
$0.21 - 0.40$	Satisfactory
$0.41 - 0.60$	Moderate
$0.61 - 0.80$	Significant
$0.81 - 1.00$	Almost perfect

Practical analysis

Fortunately, we don't have to calculate Kappa manually. The `confusionMatrix()` function from the `caret` package does this for us. We go back to its output, which we got in subsection 9.1.

```
Overall Statistics

      Accuracy : 0.5195
      95% CI   : (0.4925, 0.5463)
  No Information Rate : 0.2586
  P-Value [Acc > NIR] : < 2.2e-16

      Kappa : 0.4329

  McNemar's Test P-value : NA
```

For example, if we got $\kappa : 0.43$, it means that our model has **moderate consistency** with real data, which for a case study is a good result for a complex soil mapping task (although in real model tasks they try to achieve an indicator greater than 0.60-0.70). This is a much more significant indicator than, say, an overall accuracy of 52%, because it confirms that high precision is not just an artifact of unbalanced classes.

Why is Kappa so important? Consider an example with dominant Rendzic Leptosols. A model that always predicts " Rendzic Leptosols "

will have an Overall Accuracy = 0.9. However, when we calculate Kappa, the probability of randomly guessing "Rendzic Leptosols" will also be very high, around 0.81. As a result, the Kappa value will be close to zero, which will honestly show us that the model hasn't really learned anything.

Thus, the Kapp coefficient is a must-have tool in the arsenal of a digital soil scientist. It provides a rigorous, objective and class imbalance-resistant assessment, which allows us to be sure that our model has indeed captured the real relationships between soils and environmental factors.

9.4. Practical validation

Quantitative metrics such as overall accuracy and Kapp's ratio provide us with an important, generalized assessment of the model's performance. They provide an answer to the question, "How well does the model perform overall?". However, for a deep understanding and, most importantly, to further improve our performance, we must dive deeper and answer the question, "**Where and why is our model wrong?**". This process of analyzing the nature of errors is **practical validation**.

Practical validation is a bridge between statistics and soil science. It consists in returning to our matrix of inconsistencies and considering it not as a source of numbers, but as a diagnostic tool. We are interested **in non-diagonal elements** – those cells where the model confused one type of soil with another.

Error analysis based on a matrix of mismatches

We take another look at the output of the discrepancy matrix that we got:

```
> print(conf_matrix)
Confusion Matrix and Statistics
```

Prediction	Reference							
	Albic Luvisols	Andosols	Arenosols	Calcaric Cambisols	Cambisols	Chernozems		
Albic Luvisols	17	0	0	6	2	0		
Andosols	0	6	0	0	1	0		
Arenosols	0	0	19	0	2	3		
Calcaric Cambisols	2	0	0	12	3	1		
Cambisols	14	9	3	38	271	1		
Chernozems	0	0	7	0	0	41		
distinctly contaminated soils	0	0	0	0	0	0		
Fluvisols	2	0	7	2	5	8		
Haplic Luvisols	7	1	2	6	10	10		
Histosols	0	0	0	0	0	0		
Leptosols	0	0	0	0	0	0		
Mollic Fluvisols and Mollic Gleysols	0	0	8	0	2	24		
Planosols and Stagnosols	7	0	0	7	17	0		
Podzols	0	0	0	0	8	0		
Rendzic Leptosols	5	0	0	0	31	0		

Interpretation from the point of view of soil science:

- Error: 14 true Cambisols were classified as Albic Luvisols.
- **Analysis:** Why could the model confuse them? Both Cambisols and Albic Luvisols in Slovakia are often formed on similar parent rocks in the foothills. The key difference between Albic Luvisols is the presence of a well-defined illuvial (argic) clay accumulation horizon. characteristic of Cambisols.
- **Possible solution:** Add new covariates to the model that better reflect the stability of the landscape, such as a flux power index (SPI) or a geomorphological map.
- Error: 7 true Chernozems were classified as Luvisoli.
- **Analysis:** This is also an understandable error. Chernozems and Arenosols can occupy similar geographical zones on flat lowlands. Both types of soils can have a dark humus horizon. Probably, the model was not able to clearly distinguish them from the available set of climatic and relief predictors.
- **Possible solution:** Add covariates that better reflect the parent rock (for example, climatic indicators that better reflect the continentality of the climate, favorable for steppe processes).

Spatial analysis of errors

In addition to analyzing the matrix itself, it is extremely useful to visualize the errors on the map. We can create a map of our test points, marking those that have been classified correctly and those where the

model has made a mistake.

```
# --- Load Necessary Libraries ---
# We will use 'ranger' for modeling, 'caret' for the
# confusion matrix, and 'sf' for spatial operations.
# This code will also install them if they are not
# already present.
packages <- c("ranger", "dplyr", "caret", "rsample",
"ggplot2", "sf")
for (pkg in packages) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg)
    library(pkg, character.only = TRUE)
  }
}

# --- Spatial Visualization of Errors ---

# 1. Create a dataframe with true and predicted results,
# keeping the ID from the original test set.
results_df <- data.frame(
  ID = test_data$ID,
  WRB_true = test_data$WRB,
  WRB_pred = predictions$predictions
)

# 2. Get the coordinates for each ID from the complete
# dataset.
# This avoids issues with joining to external spatial
# files.
locations_df <- final_dataset_clean %>%
  select(ID, x, y) %>%
  distinct(ID, .keep_all = TRUE)

# 3. Join results with locations and convert to a spatial
# 'sf' object.
results_sf <- left_join(results_df, locations_df, by =
"ID") %>%
  # Convert to sf object, specifying coordinate columns
  # and the original CRS
  st_as_sf(coords = c("x", "y"), crs =
st_crs(slovakia_boundary)) %>%
  mutate(is_correct = (WRB_true == WRB_pred))

# 4. Filter out any rows that have missing geometry after
# the join.
```

```

results_sf_clean <- results_sf %>%
  filter(!st_is_empty(.))

# 5. Visualize the errors on a map with custom styling.
# We split the data to apply different aesthetics to
correct and incorrect points.
correct_points <- results_sf_clean %>% filter(is_correct
== TRUE)
incorrect_points <- results_sf_clean %>%
filter(is_correct == FALSE)

ggplot() +
  # Add the country boundary as a background
  geom_sf(data = slovakia_boundary, fill = "gray95") +
  # Add the correctly predicted points (green with a grey
border)
  # We use shape = 21 which has both fill and color
aesthetics.
  geom_sf(data = correct_points, fill = "green", color =
"grey40", shape = 21, size = 3) +
  # Add the incorrectly predicted points (red and half
the size)
  geom_sf(data = incorrect_points, color = "red", size =
1.5) +
  theme_bw() +
  labs(
    title = "Spatial Distribution of Classification
Errors",
    subtitle = "Green: Correct Predictions, Red:
Incorrect Predictions"
  )

```

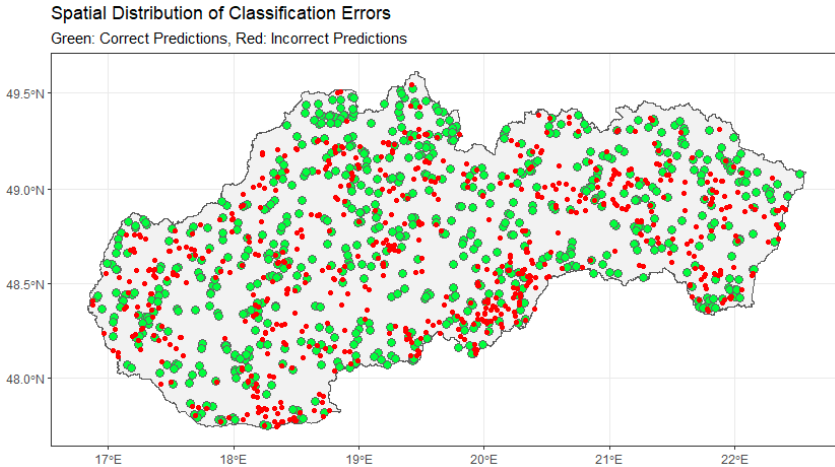


Fig. 2.18. Map of the spatial distribution of errors

Map analysis: Such a map can reveal spatial patterns in errors. Are the errors concentrated in specific geographic regions (e.g., mountain valleys or at the boundary of different geological formations)? If so, it may indicate that our model does not perform well under these specific conditions, and we lack predictors to describe these unique soil formation factors.

Practical validation transforms the process of evaluating accuracy from a simple statement of fact ("the model is accurate at X%") into an iterative process of scientific research. It allows us not only to evaluate, but also to understand and, most importantly, purposefully improve our predictive soil map.

Chapter 10. Creation and interpretation of predictive maps of soil types

We have come a long way: from data preparation and exploratory analysis to training and careful validation of our machine learning models. We have made sure that our Random Forest model has sufficient predictive power and its accuracy far exceeds random guessing. Now it is time for the final, most interesting stage – applying our trained model to the entire study area to create **a predictive soil map**.

10.1. Spatial forecasting

Spatial forecasting is the process of using a trained model to assign a predicted value (in our case, a soil class) to each individual pixel in our harmonized raster covariate stack. Essentially, we force the model to "look" at every pixel of our territory, analyze the values of altitude, slope, geology, and other predictors at that point, and, based on the rules it learned in the training sample, make the most likely forecast for the type of soil.

This process is the digital equivalent of a soil scientist-cartographer extrapolating his knowledge from a few point observations to the entire landscape. However, unlike humans, the model does so objectively, reproducibly and with extreme detail for millions of pixels.

Practical implementation with terra

Fortunately, modern tools such as the terra package make this potentially complex computing process extremely simple. terra has its own, highly optimized predict() function that "understands" how to work with raster objects and models trained with popular packages such as randomForest.

The predict() function automatically performs the following steps:

- Takes a learned model (rf_model).

Takes a multilayer raster stack of covariate (final_stack_masked).

For each pixel, it collects the values from all the layers into a vector.

Feeds this vector to the input of the model.

Receives the forecast and writes it to the corresponding pixel of the new,

original raster.

The terra package efficiently manages memory by processing large rasters in chunks, which makes it possible to work even with very large areas that do not fit into the computer RAM.

```
# --- Load Necessary Libraries ---
# We will use 'ranger' for modeling, 'caret' for the
# confusion matrix, and 'sf' for spatial operations.
# This code will also install them if they are not
# already present.
packages <- c("ranger", "dplyr", "caret", "rsample",
"ggplot2", "sf", "terra")
for (pkg in packages) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg)
    library(pkg, character.only = TRUE)
  }
}

# Assume 'final_dataset_clean', 'slovakia_boundary', and
# the original 'soil_points' sf object are loaded.
# Assume 'rf_model' is trained and loaded.
# Assume 'final_stack_masked' is loaded.

# --- Add Coordinate Layers to the Raster Stack ---

# 1. Create two new raster layers with the same
# dimensions/CRS as our stack.
# The 'init' function will populate them with
# coordinate values.
x_coord_raster <- init(final_stack_masked, "x")
y_coord_raster <- init(final_stack_masked, "y")

# 2. Rename the new layers to 'x' and 'y' to match the
# model's expectations.
names(x_coord_raster) <- "x"
names(y_coord_raster) <- "y"

# 3. Combine the original stack with the new coordinate
# layers.
final_stack_with_coords <- c(final_stack_masked,
x_coord_raster, y_coord_raster)

# 4. Verify that the names now match the model's
```

```

predictors.
print("Names in the final raster stack:")
names(final_stack_with_coords)
print("Names expected by the model:")
print(rf_model$forest$independent.variable.names)

# --- Spatial Prediction ---
# Now we use the new stack that includes the coordinate
layers.
predicted_map <- predict(final_stack_with_coords,
  rf_model, filename = "results/predicted_soil_map.tif",
  overwrite = TRUE)

print("Prediction complete. The map has been saved to
results/predicted_soil_map.tif")

```

Process Analysis: That's all! With a single command, we applied a complex model to millions of pixels. The result is a new, single-layer raster `predicted_map` object stored on disk. This is a **categorical raster**, where the value of each pixel is a numerical code corresponding to the predicted soil class (e.g. 1=Cambisol, 2=Chernozem, etc.).

Rendering the final map

Now we visualize the result. Terra has powerful built-in visualization tools that automatically create a legend for category stories.

```

# --- Visualize Final Map (Masked) ---

# 1. Ensure the boundary layer is loaded (assuming it's
named slovakia_boundary)
# and has the same CRS as the predicted map.
slovakia_boundary_proj <- st_transform(slovakia_boundary,
  crs = crs(predicted_map))

# 2. Mask the predicted map using the country boundary.
# This sets all pixels outside the polygon to NA.
predicted_map_masked <- mask(predicted_map,
  slovakia_boundary_proj)

# 3. Convert the final raster to a dataframe for ggplot2
visualization.
map_df <- as.data.frame(predicted_map_masked, xy = TRUE)
# The layer name might be complex, so we rename it for

```



```

simplicity.
colnames(map_df)[3] <- "soil_type"
# Ensure it's treated as a categorical variable
map_df$soil_type <- as.factor(map_df$soil_type)

# 4. Create the final map using ggplot2 for better
aesthetics.
ggplot() +
  # Add the raster layer
  geom_raster(data = map_df, aes(x = x, y = y, fill =
soil_type)) +
  # Add the country boundary on top
  geom_sf(data = slovakia_boundary_proj, fill = NA, color
= "black", size = 0.5) +
  # Use the Viridis color palette, which is popular in
QGIS and colorblind-friendly
  scale_fill_viridis_d(option = "D", name = "Soil Type
(WRB)") +
  theme_bw() +
  labs(
    title = "Predictive Map of Soil Types in Slovakia",
    x = "Longitude",
    y = "Latitude"
  ) +
  coord_sf(crs = crs(predicted_map))

```

Map analysis: On the final map, we can see the spatial patterns of soil distribution, which our model "studied". The detail of the map is limited only by the resolution of our initial covariates. As you can see, the predictive map predicted the distribution of the main soils of Slovakia very well. At the same time, we note that we used the minimum possible set of covariates, and when they are expanded, the accuracy of the forecast will be much higher.

This stage is the triumph of the entire DSM workflow. It transforms an abstract statistical model into a concrete, spatially explicit and useful product, ready for further analysis, interpretation and use in sustainable land use, soil protection and agronomic planning tasks.

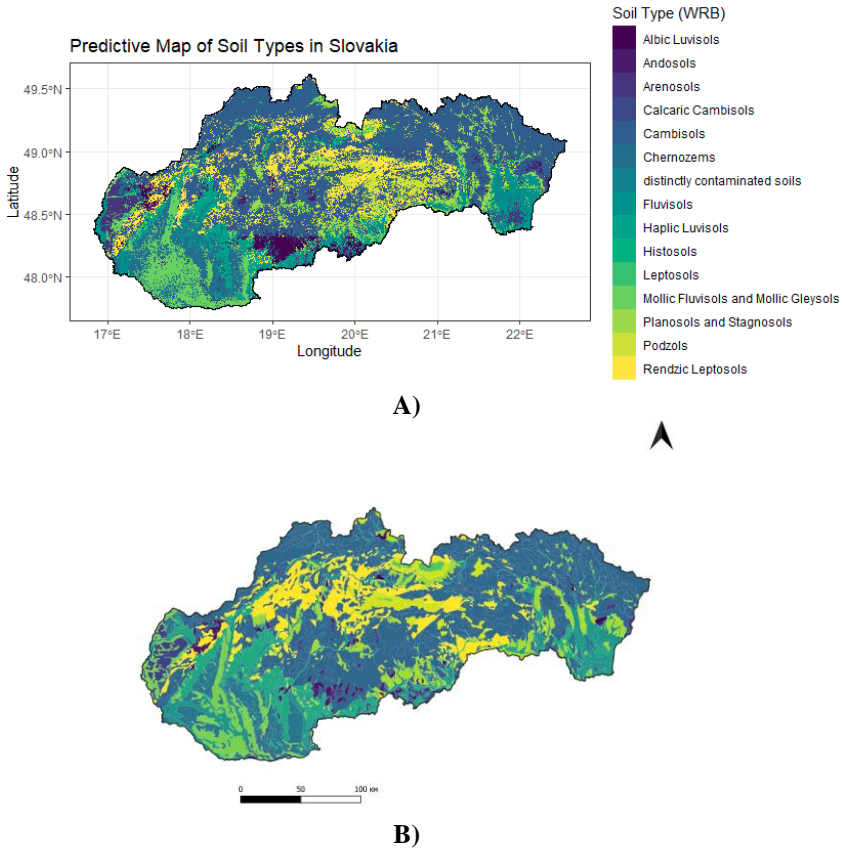


Fig. 2.19. Final predictive map of soil types (A). Each color on the map corresponds to the predicted dominant soil type for a given cell. For comparison, the original soil map (B) is given

10.2. Map Interpretation

Creating a beautiful, smoothed predictive map is the technical completion of the DSM workflow. However, the scientific work does not end there. The final and perhaps most important step is **the interpretation of the map**. This is the process in which we go from "what" (what the map shows) to "why" (why the map shows exactly

these patterns). Interpretation is an intelligent synthesis where we combine the results of our statistical modeling with our fundamental knowledge of the field Soil Science, Geography and Ecology.

A map is not an absolute truth, but a scientific hypothesis about the spatial distribution of soils, expressed in graphic form. Our task is to critically evaluate this hypothesis, understand its strengths and realize its limitations.

Relationship of spatial patterns with SCORPAN factors

The very first step of interpretation is to analyze spatial patterns on the map and correlate them with soil formation factors that we used as predictors. This is where the graph of the importance of variables, which we obtained at the stage of training the Random Forest model (subsection 8.3), comes in handy.

1. Analyzing the dominant factors: We look at our map and remember which predictors were the most important. Suppose they were elevation (altitude) and geology (geology). Do we see this on the map?

- **Altitudinal gradient:** We are likely to see a distinct zonality corresponding to the relief. For example, the warmest and driest southern lowlands (Danube Lowlands) are occupied by **Chernozems** and **Fluvisemes**. **Podzols** and **Leptosols** (skeletal soils). This pattern is a classic example of vertical soil zoning and confirms that our model has successfully captured the relationship between soil and relief.
- **Parent rock influence:** If we superimpose our map on the geologic one, we can see other patterns. For example, **the ranges of the Rendzin** (Rendzic Leptosols) on the map are likely to closely coincide with the outcrops of limestone rocks, which is their classic formation condition.

2. Consideration of constraints and uncertainties: A scientifically competent interpretation always includes a discussion of limitations.

- **Scale and resolution:** It is important to remember that our

map is a model, not a reality. The forecast for a pixel measuring 100x100 meters, is an average characteristic for this area. It does not reflect the micro-variability of the soil inside this pixel.

- **Model accuracy:** We must return to our matrix of inconsistencies (Chapter 9). If we know that the model has often confused, for example, **Cambisole** and **Luvisoli**, then on the map in the transition zones between these two types of soils, we must interpret the boundary not as a clear line, but as an area with increased uncertainty.
- **"Digital soil bodies":** Unlike traditional maps, where the cartographer draws clear polygonal contours, our map is continuous. The smoothing we applied makes the contours more realistic, but they are still the result of statistical processing rather than field delineation.

Practical application

The final stage of interpretation is the answer to the question: "How can this map be used?". Our predictive map can serve:

- **The basis for updating** existing, more generalized soil maps.
- **Inputs** for erosion models, hydrological modeling, or potential yield estimation.
- **A tool for planning** sustainable land use, identifying lands in need of protection or optimizing agricultural practices.
- Thus, interpretation is not just a description of a map, but an in-depth analysis that combines statistical results with expert knowledge. It is this synthesis that turns our digital product into a true scientific tool, contributing to a better understanding and management of invaluable soil resources.

PART III. PREDICTIVE MODELING OF SOIL CHARACTERISTICS

Chapter 11. Continuous Variable Modeling: Organic Carbon Content

11.1. Differences between Modeling of Continuous and Categorical Variables

In the second part of this tutorial, we have successfully gone all the way through digital mapping for **the categorical variable** – soil type. We have trained the model to assign a specific label or class to each pixel on the map ("Cambisol", "Chernozem", etc.). This task, known as **classification**, is fundamental for creating soil maps. However, for many practical tasks of agronomy, ecology and sustainable land use, we need to know not only the type of soil, but also the type of soil, but also its quantitative characteristics. For example, what exactly is the organic carbon content, what is the cation exchange capacity, or what is the folding density?

The answer to these questions is provided by another type of guided machine learning – **regression**. In this part of the book, we will focus on regression modeling, choosing one of the most important properties of soil – **Soil Organic Carbon (SOC)** – as our **target variable**.

The transition from classification to regression changes not so much the overall DSM workflow as key aspects in the modeling and evaluation phases. We look at these fundamental differences.

1. Nature of the target variable

This is the most important difference that defines everything else.

- **Classification:** The target variable is **categorical** (in R – factor). It has a limited set of discrete, disordered levels (for example, 8 types of soils). The model predicts **belonging** to one of these classes.
- **Regression:** The target variable is **continuous** (in R – numeric). It can take any numerical value within a certain range (e.g. SOC content can be 1.2%, 3.45%, 5.8%, etc., or 15, 50, 126 t/ha). The model predicts **a specific numerical value**.

- **2. Algorithms and their settings**
- While many algorithms, like Decision Trees and Random Forest, can solve both types of problems, their inner workings and settings are different.
- **Decision trees (rpart):** For classification, we used method = "class". For regression, we will use method = "anova". The criterion for dividing nodes changes from "purity" of classes to minimizing the sum of squares of deviations from the mean in newly formed groups.
- **Random Forest (randomForest):** The algorithm automatically determines the type of task by the type of target variable. For regression, the final prediction for the new observation is not made by "voting" the trees, but by **averaging the** predictions from all the trees in the forest.

3. Metrics to Evaluate Accuracy

This is perhaps the most obvious difference. Metrics based on the comparison of discrete labels are completely unsuitable for evaluating the accuracy of numerical predictions.

- **Classification:** We relied on **the Discrepancy Matrix**, from which we obtained **the Overall Accuracy**, **the Manufacturer/User Accuracy**, and, most importantly, **the Kappa Coefficient**.
- **Regression:** To assess the accuracy of regression models, a completely different set of metrics is used, based on the analysis of **errors (residuals)** - the difference between true and predicted values. The key metrics are:
 - **Coefficient of determination (R2):** Shows what proportion of variability (variability) of the target variable our model explains. Values from 0 to 1; the closer to 1, the better.
 - **Root Mean Squared Error (RMSE):** This is essentially the standard deviation of model errors. It is measured in the same units as the target variable (e.g., as a percentage of SOC), making it easy to interpret.
 - **Biased (Bias or Mean Error):** Shows whether the model

has a systematic tendency to overestimate or underestimate forecasts. The ideal value is 0.

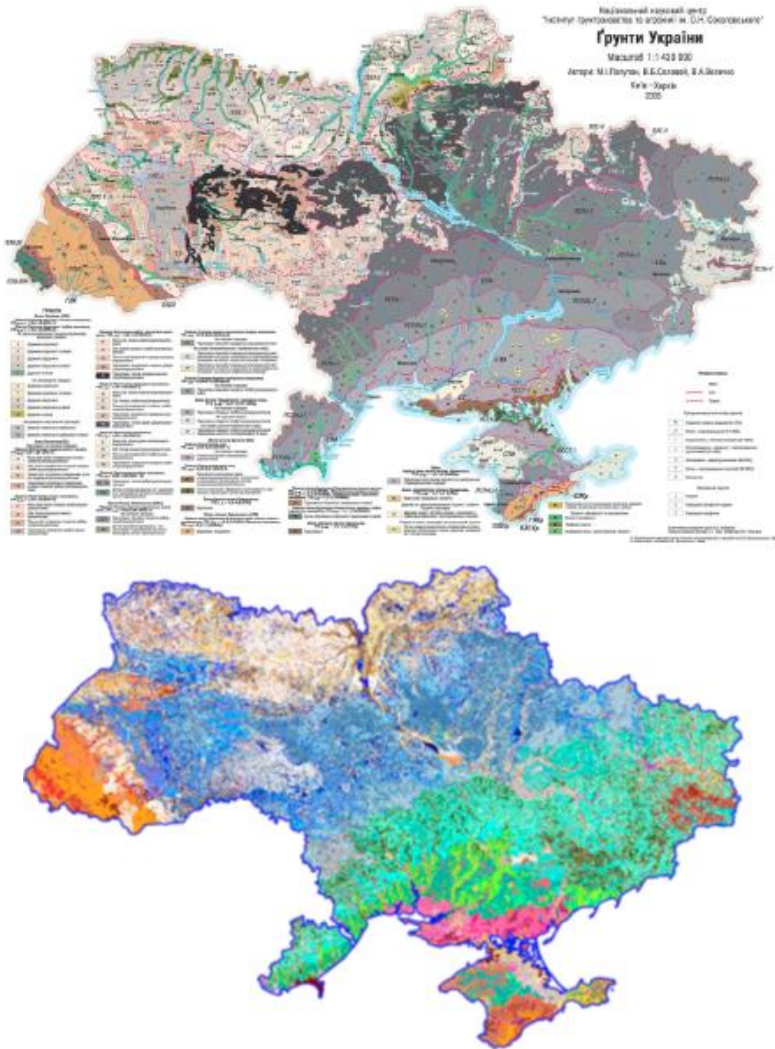


Fig. 3.1. Comparison of a classic SOC map of Ukraine (above) and a map based on regression models of the soil characteristics (down).

Thus, while the general logic of SCORPAN and the DSM workflow remain the same, the transition to continuous variable modeling requires a change in the tooling in the training phases and, especially, model validation. In the following subsections, we will apply these new approaches to create a detailed map of organic carbon content.

11.2. Focus on Soil Organic Carbon (SOC)

Moving on to regression modeling, we chose **Soil Organic Carbon (SOC) as our target variable**. This choice is not accidental. SOC is perhaps one of the most important and most studied properties of soil, and there are several fundamental reasons for this. Understanding its role will help us better interpret the results of our simulations and realize the practical significance of the maps created.

Why is SOC so important?

The organic carbon content is an integral indicator located at the intersection of key environmental, agronomic and climatic processes.

- **Soil Health and Fertility Indicator:** Soil organic matter, of which carbon is the main constituent, is the foundation for most fertility processes. It improves soil structure by increasing soil aggregation and erosion resistance; increases the ability to retain moisture, which is critical in climate change; is a source of nutrients for plants (nitrogen, phosphorus, sulfur); and supports the biodiversity of soil microorganisms. SOC maps are indispensable for accurate agriculture and sustainable management of agroecosystems.

A key component of the global carbon cycle: Soils are the largest terrestrial reservoir of carbon on the planet, containing several times more carbon than the entire atmosphere and biomass combined. Even small changes in soil carbon stocks can have a significant impact on the concentration of greenhouse gases in the atmosphere. Therefore, accurate maps of the spatial distribution of SOC are critical for modeling global climate change, assessing the potential of soils to sequester (sequester) carbon and developing national strategies to mitigate the effects of climate change.

SOC Data Features

As a continuous variable, SOC content data have certain statistical properties that we must consider before modeling. One of the most characteristic features of many soil properties, and SOC in particular, is **the positive-asymmetric (right-handed) distribution**.

This means that most of the values are relatively low, but a small number of very high values ("long right tail") are present in the data. Such high values are often observed, for example, in peat soils (histosols) or in the powerful humus horizons of the Chernozems.

We visualize a typical SOC distribution using a histogram.

```
# --- Load Necessary Libraries ---
# This script will install packages if they are not
# already present.
packages <- c("sf", "dplyr", "ggplot2")
for (pkg in packages) {
  if (!require(pkg, character.only = TRUE)) {
    install.packages(pkg)
    library(pkg, character.only = TRUE)
  }
}

# --- Chapter 11: Modeling SOC ---
# --- Section 11.3: Data Transformation ---

# 1. Load the soil point data
# We assume the data is in the 'gis_data' subfolder.
soil_points <-
  st_read("gis_data/slovakia_soil_points_3857.gpkg")

# 2. Prepare the data for transformation and modeling
# We will use SOC_t_ha as our target variable.
modeling_data <- soil_points %>%
  select(SOC_t_ha) %>%      # Select only the target
  variable
  as.data.frame() %>%      # Convert sf object to a
  regular dataframe
  select(-geom)            # Remove the empty geometry
  column

# Plot 1: Histogram of original SOC stock data
p_original <- ggplot(modeling_data_transformed, aes(x =
  SOC_t_ha)) +
```

```
geom_histogram(bins = 30, fill = "darkolivegreen",
color = "black") +
  theme_bw() +
  labs(title = "Before Transformation", x = "SOC Stock,
t/ha", y = "Frequency")
```

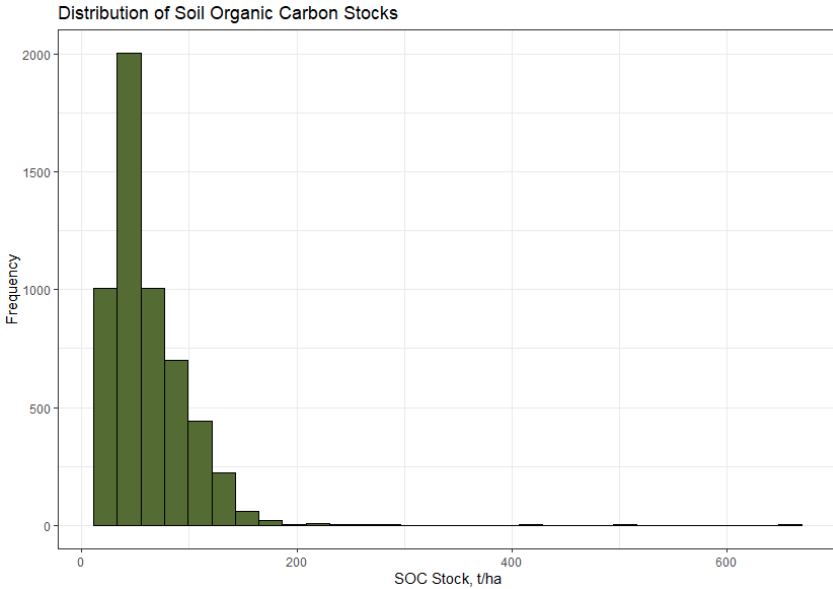


Fig. 3.2. A histogram illustrating a typical positive-asymmetric distribution of SOC content. A large number of low values and a long "tail" with few high values are visible

Graph analysis: This asymmetry can create problems for some statistical models that work better with data that have a normal (symmetrical) distribution. In addition, the presence of extremely high values can disproportionately affect the model's learning process.

Because of this feature, standard practice before modeling SOC_s (and many other ground properties) is **to transform the data** in order to make their distribution more symmetrical. The most common and efficient method for this is **logarithmic transformation**. We will discuss this step in detail in the next subsection, as it is key to building a reliable regression model.

11.3. Preparing Data for SOC Modeling (Logarithmic Transformation)

In the previous subsection, we found that organic carbon content (SOC) and many other soil properties often have a **positive-asymmetric distribution**. This means that most values are relatively low, but there are a small number of extremely high values, which creates a "long right tail" on the histogram. Such asymmetry can negatively affect the performance of regression models, especially linear ones, which often make assumptions about the normality of the error distribution. Even for more flexible models like Random Forest, strong asymmetries and the presence of outliers can make the learning process difficult.

To solve this problem, standard practice is **data transformation** – applying a mathematical function to our target variable in order to make its distribution more symmetric, similar to normal (bell-shaped). For positive-asymmetric data, the most common and most effective method is **logarithmic transformation**.

How Does a Logarithmic Transformation Work?

The natural logarithm (`log()` in R) has the property of "compressing" large values and "stretching" small ones. When we apply it to our asymmetric distribution, it effectively "pulls in" the long right tail, making the overall distribution much more symmetrical.

Important note: the logarithm is not defined for zero and negative values. Since the SOC content cannot be negative, the main problem is zeros. If there are zero SOC values in our data, the standard approach is to add a small constant before logarithm, such as `log(soc_percent + 1)`.

Practical implementation and visualization of the effect

We apply the logarithmic transformation to our SOC data and visually evaluate its effect by comparing the histograms before and after the transformation. We'll use the `mutate()` function with `dplyr` to create a new column with transformed values.

```
# 3. Apply the Logarithmic Transformation
# Create a new column with log-transformed SOC stock
values.
```

```

modeling_data_transformed <- modeling_data %>%
  mutate(log_soc = log(SOC_t_ha))

# --- 4. Visualize the effect of the transformation ---

# Plot 2: Histogram of log-transformed SOC stock data
p_transformed <- ggplot(modeling_data_transformed, aes(x
= log_soc)) +
  geom_histogram(bins = 30, fill = "skyblue", color =
"black") +
  theme_bw() +
  labs(title = "After Log Transformation", x = "log(SOC
Stock)", y = "Frequency")

# Combine the two plots side-by-side using the patchwork
package
p_original + p_transformed

```

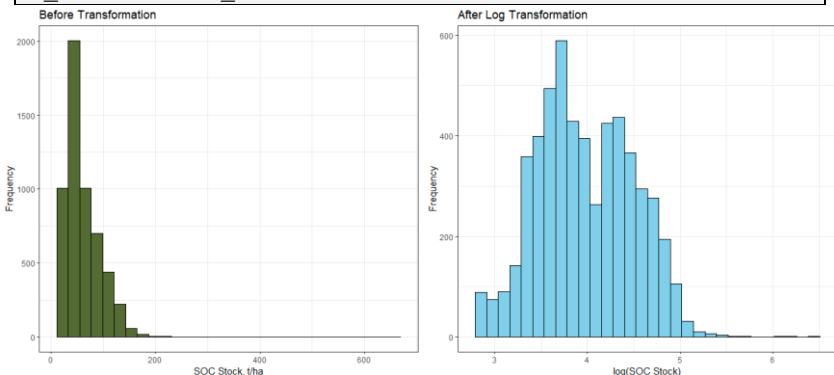


Fig. 3.3. Comparison of the distribution of SOC contents before (left) and after (right) logarithmic transformation. You can clearly see how the asymmetrical distribution turns into a symmetrical, bell-shaped one

Graph analysis: The result is obvious. The left histogram shows a strongly asymmetric distribution of the original data. The right histogram, showing the distribution of logarithmic values, is much more symmetric and resembles a normal distribution. It is on these transformed data (`log_soc`) that we will train our regression model.

The need for reverse transformation

This is **a critical point** that should not be overlooked. Our model will learn to predict **the logarithm of the SOC**. Therefore, its predictions

(for both the test sample and the final map) will be on a logarithmic scale. These values do not have a direct physical interpretation.

To get the predictions in original, understandable units (SOC percentages), we must perform **an inverse transformation**. The inverse function to the natural logarithm is **the exponent (exp() in R)**

```
SOC_predicted=exp(log(SOC_predicted))
```

We will apply this operation in the final stage, after receiving the predictions from the model, to create a map on which the values can be easily interpreted.

11.4. Exploratory Data Analysis

Once we have prepared and transformed our target variable, we should not immediately move on to building a complex model. An important intermediate step is **Exploratory Data Analysis (EDA)**. The purpose of the EDA is to investigate the relationships between our target variable (now log_soc) and our predictors.

- ✓ Are there any statistically significant relationships between our covariates and SOC content at all?
- ✓ Which of the predictors look the most promising?
- ✓ What is the nature of these relationships (linear, non-linear)?

The answers to these questions give us a deep understanding of the data, confirm the validity of the choice of our predictors according to the SCORPAN model, and help in the further interpretation of the simulation results. For EDA, we will use simple but powerful visual and statistical methods.

Correlation analysis for continuous predictors

The first step is to estimate **the linear relationship between our target variable and all continuous predictors**. The Pearson correlation coefficient (r) measures the strength and direction of the linear relationship between two variables. It ranges from -1 (perfect negative relationship) to +1 (perfect positive relationship), where 0 means no linear relationship.

We calculate and visualize the correlation matrix for our data.

```
# 4. Load the raster stack of environmental covariates
covariate_stack <-
rast("gis_data/all_slovakia_terrain_derivatives.tif")

# 5. Create and add coordinate layers to the stack
# This is necessary if the model was trained with x and y
as predictors.
x_coord_raster <- init(covariate_stack, "x")
y_coord_raster <- init(covariate_stack, "y")
names(x_coord_raster) <- "x"
names(y_coord_raster) <- "y"
full_stack <- c(covariate_stack, x_coord_raster,
y_coord_raster)

# 6. Extract covariate values for each point
# Ensure CRS match before extraction
soil_points_proj <- st_transform(soil_points, crs =
crs(full_stack))
extracted_data <- extract(full_stack, soil_points_proj)

# 7. Prepare the final dataset for exploratory analysis
# We combine the transformed SOC data with the extracted
predictor values.
exploratory_dataset <- modeling_data_transformed %>%
  bind_cols(extracted_data) %>%
  mutate(geology = as.factor(geology)) %>% # Ensure
geology is a factor
  na.omit() # Remove any rows with missing values

# 8. Prepare a numeric-only dataset for the correlation
matrix
numeric_dataset <- exploratory_dataset %>%
  select_if(is.numeric) %>%
  select(-ID, -SOC_t_ha) # Remove ID and original SOC
column

# 9. Calculate the correlation matrix
cor_matrix <- cor(numeric_dataset)

# 10. Visualize the correlation matrix for numeric
variables
corrplot(cor_matrix, method = "circle", type = "upper",
order = "hclust",
```

```
tl.col = "black", tl.srt = 45)
```

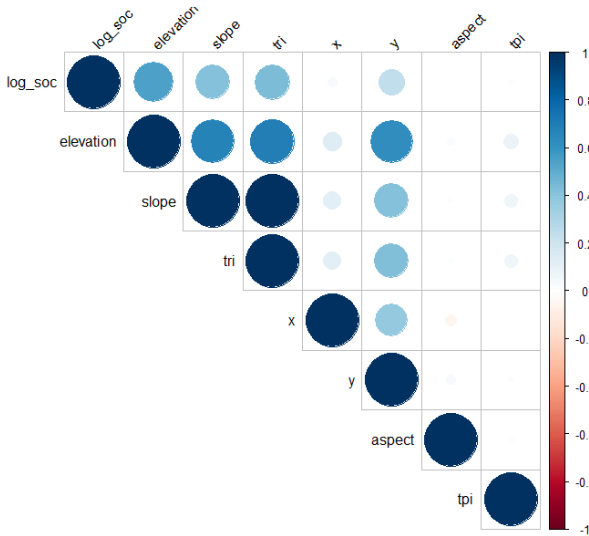


Fig. 3.4. A correlogram that visualizes the correlation matrix. The size and color of the circles show the strength and direction of correlation between the variables

Graph analysis: We are most interested in the first row (or column) corresponding to `log_soc`. We can see, for example, a strong positive correlation with `elevation` (large blue circle) and `tri`. This makes complete soil science sense: mountainous areas (higher altitude) are usually colder and wetter, which slows down the decomposition of organic matter.

Analysis of relationships with categorical predictors

Correlation does not work for categorical variables such as geology. To assess their relationship to `log_soc`, it is best to use **box plots**, which allow you to visually compare the SOC distribution for each category of the parent rock.

```
# 11. Create an optimized box plot for the 'geology'
variable with many categories
# We will reorder the geology factor based on the median
log_soc value.
# This makes the plot much more interpretable.
```

```

ggplot(exploratory_dataset, aes(x = reorder(geology,
log_soc, FUN = median), y = log_soc)) +
  geom_boxplot(fill = "purple", alpha = 0.7) +
  theme_bw() +
  labs(
    title = "Distribution of log(SOC) by Parent
Material",
    subtitle = "Categories are ordered by median SOC
value",
    x = "Parent Material (Geology)",
    y = "log(SOC Stock)"
  ) +
  # Remove x-axis text labels as they would be unreadable
  theme(axis.text.x = element_blank(),
        axis.ticks.x = element_blank())

```

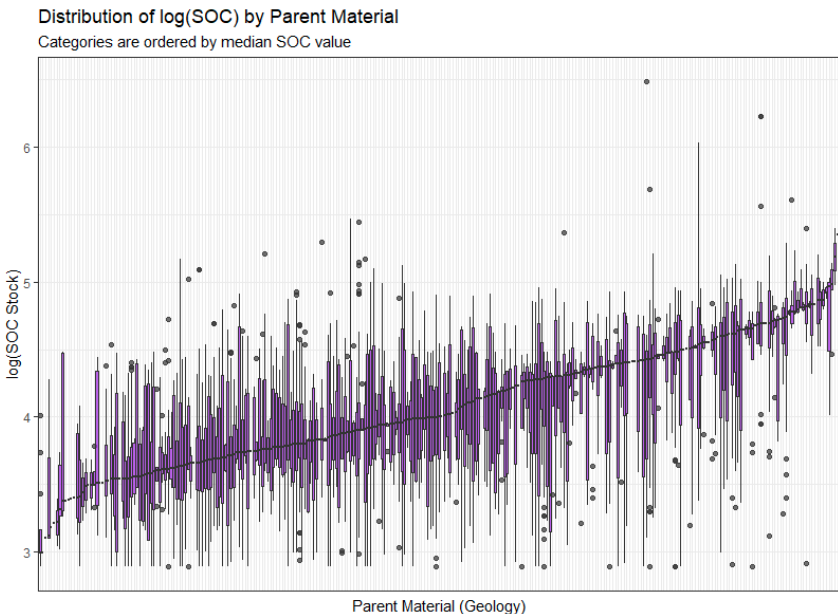


Fig. 3.5. Box diagrams comparing the log(SOC) distribution for different geological classes

Graph analysis: This graph can clearly show that the median values and variability of SOC differ significantly between types of parent

rocks. For example, on loess rocks that are rich in nutrients, the SOC content may be systematically higher than on poor sandstones. This confirms that geology is an important predictor.

Exploratory analysis is an indispensable stage that allows you to "get acquainted" with the data. It confirms that there are logical, earth-science-interpretable relationships between our predictors and the target variable. This gives us confidence that our approach is sound and we can proceed to build a regression model with the expectation of obtaining meaningful results.

Chapter 12. Regression Models: Random Forest and Cubist

12.1. Random forest for regression

In Part II, we successfully applied the Random Forest algorithm to **classify** soil types. We saw how an ensemble of hundreds of decision trees, each of which "votes" for a specific class, allows us to achieve high accuracy and stability of the forecast. Now we adapt this same powerful approach to solve the **regression** problem – predicting a continuous quantitative variable, namely our logarithmic organic carbon (log_soc) content.

The conceptual basis of the algorithm remains the same: we also build a "forest" of a large number of trees on bootstrap data samples, using a random subset of predictors at each step. However, key aspects of its inner workings and the way in which the final forecast is obtained change dramatically.

Key Differences of Regression Random Forest

- **Criterion for splitting nodes:** If in classification trees the algorithm looked for a division that maximizes the "purity" of classes in child nodes (for example, by the Gini index), then in regression trees it looks for a division that minimizes **the variability (variance)** of the target variable. In other words, it tries to divide the data so that in each of the newly formed groups, the SOC values are as similar to each other as possible.

Prediction: Instead of "voting" for the most popular class, the final prediction for a new observation in the regression Random Forest is obtained by **averaging** the predictions from all the individual trees in the forest. Each tree gives its own numerical prediction, and the end result is their arithmetic mean.

Practical implementation and analysis of the model

Fortunately, the `randomForest` package is versatile. The `randomForest()` function automatically detects the type of task (classification or regression) by the type of your target variable. Since `log_soc` is numeric, the function will automatically switch to regression mode.

Step 1: Data Preparation and Model Training

We will use the same approach to breaking down data into training and test samples as before.

```
# --- Chapter 12: Regression Modeling ---

# 1. Split data into training (75%) and testing (25%)
sets
# We use the full exploratory dataset for this
set.seed(456)
data_split <- initial_split(exploratory_dataset, prop =
0.75)
train_data <- training(data_split)
test_data <- testing(data_split)

# 2. Train the Random Forest regression model using
ranger
# ranger automatically detects the regression task from
the numeric target variable.
set.seed(456)
rf_model_reg <- ranger(
  formula = log_soc ~ . - SOC_t_ha - ID, # Predict
log_soc using all other variables except the original SOC
and ID
  data = train_data,
  num.trees = 500,
  importance = "permutation"
)

# 3. Print the model summary
# It will show the R-squared value based on OOB data.
print(rf_model_reg)
```

Ranger result

```
Call:
ranger(formula = log_soc ~ . - SOC_t_ha - ID, data = train_data,      num.trees = 500, importance = "permutation")
```

Type:	Regression
Number of trees:	500
Sample size:	4082
Number of independent variables:	8
Mtry:	2
Target node size:	5
Variable importance mode:	permutation
Splitrule:	variance
OOB prediction error (MSE):	0.1601517
R squared (OOB):	0.3913842

Output Analysis: The print() output for the regression model

provides a different set of metrics:

- Type of forest: regression.
- Mean of squared residuals (MSE): The mean square of errors calculated on OOB data. This is an indicator of the average value of the model error. The smaller it is, the better.
- % Var explained: The percentage of variation explained. It is analogous to the coefficient of determination (R^2) for OOB data. It shows what proportion of variability in SOC content in the data our model was able to explain using predictors. A value of 39% means that the model explains 39% of the variability of the SOC, which is not the highest result, but given the minimal set of predictors is quite good.

Estimating the importance of variables

As in classification, we can estimate which predictors contribute the most to the accuracy of the regression model. The metrics of importance here are also different:

- **%IncMSE:** Percentage increase in standard error (MSE). Shows how many percent the average model error will increase if you "shuffle" the value of a given predictor, destroying its relationship with the target variable. **This is the most important and reliable indicator.**
- **IncNodePurity:** Increase in the "purity" of nodes, measured by decreasing the sum of the squares of the remainders.

```
# 4. Get and plot Variable Importance
# Extract the importance scores from the model object
importance_scores <- importance(rf_model_reg)

# Convert the named vector of scores into a data frame
# for plotting with ggplot2
importance_df <- data.frame(
  Variable = names(importance_scores),
  Importance = importance_scores
) %>%
  # Arrange the variables by importance for a cleaner
  plot
  arrange(Importance) %>%
  mutate(Variable = factor(Variable, levels = Variable))
# This keeps the sorted order in the plot
```

```
# Create the variable importance plot using ggplot2
ggplot(importance_df, aes(x = Importance, y = Variable))
+
  geom_col(fill = "darkred") +
  theme_bw() +
  labs(
    title = "Variable Importance for SOC Regression",
    x = "Importance (Permutation)",
    y = "Predictor"
  )
```

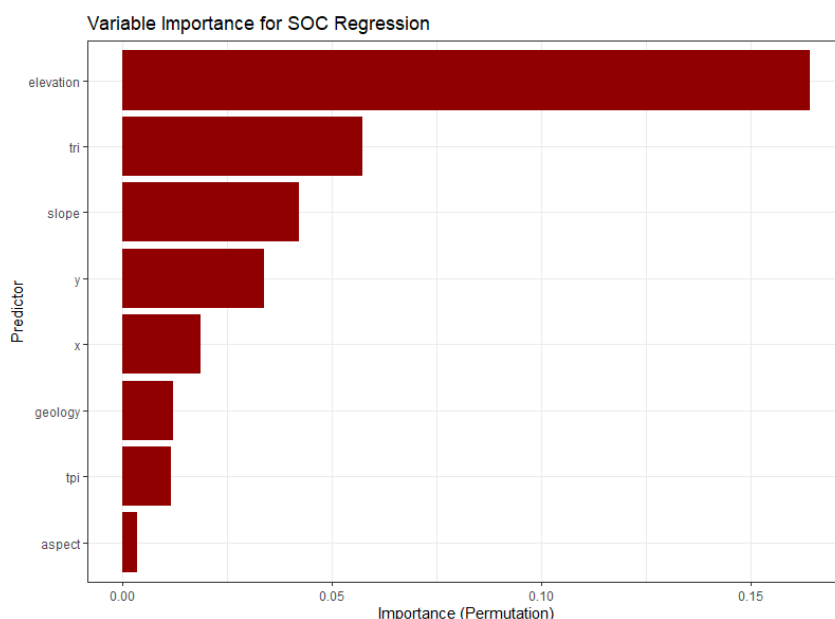


Fig. 3.6. Graph of the importance of variables for a regression model.
Predictors are sorted by their effect on model accuracy

Graph analysis: This graph allows us to draw conclusions about the key factors that control the distribution of organic carbon in our area. For example, we may find that elevation, tri (Terrain Ruggedness Index) and slope are much more important than, say, the aspect (slope exposure).

The Random Regression Forest is an extremely powerful and flexible

tool. It does not make rigid assumptions about the linearity of relationships, can work with a large number of predictors, and is resistant to outliers. This makes it one of the best "ready-to-use" algorithms for predictive mapping of continuous soil properties.

12.2. The Cubist model

The random forest is an extremely powerful and versatile tool for regression. However, its main drawback is often considered to be low **interpretation**. We can find out which predictors are important, but the model itself remains a "black box" – we cannot easily understand *how* exactly it turns the predictor values into the final prediction. This can be a significant limitation.

Fortunately, there are alternative approaches that combine high accuracy with excellent interpretation. One of the most famous and efficient such algorithms is **Cubist** (Kuhn & Quinlan, 2025). This algorithm, developed by Ross Quinlen (author of the famous C4.5 decision trees), is a unique hybrid that combines **rule-based models** and **linear regression**.

How does Cubist work?

Cubist is a complex but intuitive algorithm that works in two main stages:

- **Creating Rules:** In the first step, Cubist works like a decision tree. It recursively divides the predictor space to create a set of comprehensive and mutually exclusive **rules**. Each rule is essentially a combination of if-and-then conditions that defines a specific subset of data. For example, one rule might look like this: IF elevation \leq 550m AND geology = "Les".

Building linear models: This is a key difference from conventional trees. For each subset of the data defined by the rule, Cubist does not just calculate the mean, but builds **a separate multiple linear regression model**. That is, each rule has its own unique equation that relates the target variable to predictors. This allows the model to capture local linear dependencies, which may be different in different parts of the landscape.

The final prediction for a new observation is made by determining

which rule it falls under and then applying the corresponding linear equation.

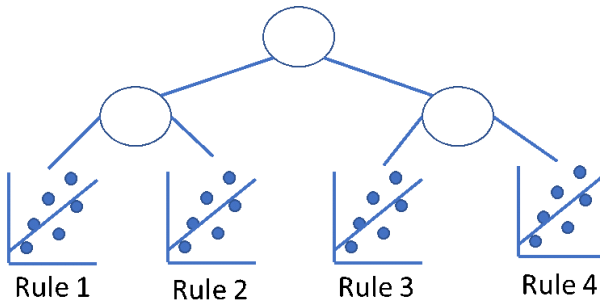


Fig. 3.7. Conceptual diagram of the Cubist model. It shows how the predictor space is divided into several regions (rules), and each region builds its own linear regression model ([from Jacey Heuer, 2019](#))

Practical implementation with the Cubist package

We build a Cubist model based on our data. The Cubist package has a slightly different syntax than `randomForest`: it requires the predictors and the target variable to be passed as separate objects.

```
# Load the Cubist library
# Cubist package on CRAN: https://cran.r-project.org/web/packages/Cubist/index.html
install.packages("Cubist")

library(Cubist)
library(dplyr)

# We use the same train_data and test_data from the RF
example

# Cubist requires predictors (x) and the target (y) to be
separate
# Let's prepare the training data
train_predictors <- train_data %>% select(-log_soc)
train_target <- train_data$log_soc

# Train the Cubist model
# 'committees' is an important hyperparameter, similar to
boosting.
```

```
# It creates multiple models and averages their
# predictions, improving accuracy.
cubist_model <- cubist(x = train_predictors, y =
train_target, committees = 5)

# Print the model summary to see the rules and
# performance
summary(cubist_model)
```

```
> summary(cubist_model)
```

```
call:
cubist.default(x = train_predictors, y = train_target, committees = 5)
```

```
Cubist [Release 2.07 GPL Edition] Tue Aug 19 18:58:48 2025
```

```
-----
Target attribute `outcome`

Read 4082 cases (9 attributes) from undefined.data

Model 1:

Rule 1/1: [334 cases, mean 3.574414, range 2.890372 to 4.532599, est err 0.257179]

  if
    elevation <= 221.8894
    geology in {25, 45, 48, 89, 90, 92, 98, 100, 103, 112, 129, 148, 149,
173, 186, 201, 202, 203, 214, 216, 220, 227, 230, 232, 233,
236, 244, 279, 282, 283, 284, 285, 288, 290, 291, 300, 305,
310, 319, 324, 330, 333}
  then
    outcome = 3.963969 - 0.00231 elevation

Rule 1/2: [501 cases, mean 3.797717, range 2.890372 to 5.123833, est err 0.360335]

  if
    elevation > 221.8894
    geology in {6, 20, 21, 58, 63, 80, 81, 86, 100, 107, 111, 116, 122, 129,
134, 149, 154, 172, 178, 180, 182, 186, 192, 201, 207, 212,
213, 217, 221, 228, 231, 236, 268, 269, 278, 292, 293, 296,
298, 303, 307, 323, 335, 336, 340, 342, 343}
  then
    outcome = 3.169425 + 0.00092 elevation + 0.0107 tri - 0.018 tpi

Rule 1/3: [956 cases, mean 3.835163, range 2.890372 to 5.44846, est err 0.282122]
```

Output Analysis: The output summary() is extremely informative and is the main advantage of Cubist.

- **Performance Score:** At the beginning, a mean and relative error score calculated using internal cross-validation is output.
- **Importance of Variables:** Cubist shows how often each variable has been used in **rule conditions** and in **linear models**. This provides a deep understanding of the role of each predictor.

- **Rules:** The most interesting part. Cubist prints each rule and its corresponding linear model.
- **Example of a rule interpretation:** Suppose the output for one of the rules looks like this:

This output indicates that”: "For areas located at an altitude of up to 221.88 meters on loess or alluvial sediments (334 such cases in our data), the content of log_soc can be predicted using the equation " $3.96 - 0.002 \cdot \text{elevation} \dots$ ".

Cubist is a great alternative to the Random Forest, especially when the interpretation of the model is as much a priority as its accuracy. In the next section, we will perform a formal validation of both models on a test sample in order to objectively compare their predictive power.

Chapter 13. Validation of regression models and uncertainty analysis

Having built two powerful regression models – the Random Forest and the Cubist – we, as in the case of classification, must carry out their objective and comprehensive validation. Our goal is not just to choose the "best" model, but also to deeply understand how accurate its predictions are, what kind of mistakes it makes, and whether it has systematic tendencies to over- or underestimate. To do this, we will use a set of specialized metrics designed specifically to evaluate continuous forecasts.

13.1. Key metrics for regression (R^2 , RMSE, Bias)

As we have already noted, metrics from the classification, such as the matrix of mismatches and the Kapp coefficient, are completely unsuitable for regression. Instead of analyzing whether the class is correctly guessed, we analyze **errors (residuals)** – the difference between the true, observed value (y) and the predicted value (\hat{y}).

$$\text{residuals} = y - \hat{y}$$

Analyzing the distribution of these errors in the test sample is the basis for calculating all key regression metrics.

Coefficient of determination (R^2)

The coefficient of determination, or **R-squared**, is one of the most popular metrics. It shows what **fraction (proportion) of the total variability (variance) of the target variable our model was able to explain** using predictors.

R^2 ranges from 0 to 1 (or 0% to 100%).

- ✓ $R^2=1$: Ideal model that explains 100% data variability.
- ✓ $R^2=0$: A completely useless model that explains 0% variability (its predictions are no better than just an average of all data).

Interpretation: If we obtained $R^2=0.82$, this means that 82% of the variability in the SOC content in our test sample is due to the SCORPAN factors included in our model. The remaining 18% is "unexplained"

variability due to noise in the data or factors that we did not take into account.

Root Mean Squared Error (RMSE)

If R^2 is a relative measure of model quality, then RMSE is **an absolute measure of the magnitude of error**. It is essentially the standard deviation of model errors, and most importantly, it is measured **in the same units as our target variable**.

$$\text{RMSE} = \sqrt{[\Sigma(P_i - O_i)^2 / n]}$$

Interpretation: If our target variable is \log_soc , then the RMSE will be measured in units of $\log(\%)$. If we got $\text{RMSE} = 0.3$, it means that, on average, the predictions of our model deviate from the true values by ± 0.3 units of $\log(\%)$. usually, the best.

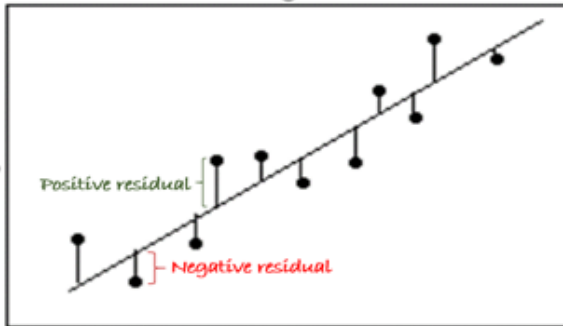


Fig. 3.8. Illustration of errors (residuals) for a regression model. A scatter plot with a regression line is shown. Vertical segments from points to a line are errors, and RMSE is a generalized measure of their average value

Biases (Bias or Mean Error, ME)

This metric shows whether the model has **systematic error**. It is calculated as the simple arithmetic mean of all errors.

$$\text{Bias}(\hat{\theta}) = E(\hat{\theta}) - \theta$$

Interpretation:

- **Bias > 0:** The model systematically **underestimates** forecasts (true values are on average greater than predicted).
- **Bias < 0:** The model systematically **overestimates** forecasts.

- **Bias \approx 0:** The model is **unbiased**, that is, its errors are random and cancel each other out on average. This is a perfect result.

Practical calculation and comparison of models

We now apply these metrics to compare our `rf_model_reg` and `cubist_model` models on the test sample.

```
# --- Chapter 13: Model Validation ---

# 1. Make predictions on the test data for both models
# For ranger, we need to access the '$predictions'
# element and use the 'data' argument
rf_preds <- predict(rf_model_reg, data =
test_data)$predictions
# For Cubist, we need to provide only the predictor
# columns
cubist_preds <- predict(cubist_model, newdata = test_data
%>% select(-log_soc, -SOC_t_ha, -ID))

# 2. Create a results dataframe to hold true values and
# predictions
results_df <- tibble(
  true_log_soc = test_data$log_soc,
  rf_pred = rf_preds,
  cubist_pred = cubist_preds
)

# 3. Calculate metrics for the Random Forest model
rf_metrics <- results_df %>%
  metrics(truth = true_log_soc, estimate = rf_pred)
print("--- Random Forest Validation Metrics ---")
print(rf_metrics)

# 4. Calculate metrics for the Cubist model
cubist_metrics <- results_df %>%
  metrics(truth = true_log_soc, estimate = cubist_pred)
print("--- Cubist Validation Metrics ---")
print(cubist_metrics)
```

```

> print("--- Random Forest Validation Metrics ---")
[1] "--- Random Forest Validation Metrics ---"
> print(rf_metrics)
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>    <chr>        <dbl>
1 rmse    standard      0.396
2 rsq     standard      0.431
3 mae     standard      0.317
> print("--- Cubist Validation Metrics ---")
[1] "--- Cubist Validation Metrics ---"
> print(cubist_metrics)
# A tibble: 3 × 3
  .metric .estimator .estimate
  <chr>    <chr>        <dbl>
1 rmse    standard      0.403
2 rsq     standard      0.413
3 mae     standard      0.317

```

Output Analysis: The yardstick package automatically calculates a set of standard metrics, including rmse, rsq(R²), and mae (Mean Absolute Error, similar to RMSE, but less sensitive to emissions). We can easily compare tables for both models. For example, we can see that Cubist has a slightly higher RMSE, suggesting lower accuracy, while Random Forest has a single R². Quantitative analysis allows us to make an informed choice in favor of one or another model for the final spatial forecasting.

13.2. Visual diagnostics

Quantitative metrics such as R² and RMSE provide us with important but very concise information about model performance. They summarize millions of details about errors into a single number. To truly understand *how* and *where* our model goes wrong, we need to go beyond these generalizations and resort to **visual diagnostics**. Graphical error analysis is an indispensable tool that allows you to identify systematic problems, such as bias or heteroscedasticity (unevenness of errors), which may not be noticeable when analyzing numerical metrics alone.

The most informative and common visual diagnostic tool for regression models is **the scatter plot "Observed vs. Predicted" values**.

Chart "Observed vs. Predicted"

This graph is built very simply:

- On the X axis, true, observed values **from the test sample** (**true_log_soc**) are deposited.
- Along the Y axis, the values predicted by the model **for the same points** (**rf_pred**, **cubist_pred**) are postponed.

Interpretation of the graph:

- **Perfect model:** If our model were perfect, all the points on this graph would lie exactly on **the 1:1 line** (the line where $y = x$). This line represents perfect consistency.
- **Real model:** In reality, the points will always be scattered around this line. **The degree of scattering** visually shows us the magnitude of the errors (the denser the points are grouped around the line, the smaller the RMSE).
- **Systematic deviations:** The most important thing is to look for systematic patterns in the deviations. For example, if for high SOC values the points are systematically *below* the 1:1 line, it means that the model **underestimates** for soils with high organic content.

We plot this graph for our Random Forest model.

```
# 5. Visual Diagnostics: Observed vs. Predicted Plots
# Create the plot for the Random Forest model
p_rf <- ggplot(results_df, aes(x = true_log_soc, y =
rf_pred)) +
  geom_point(alpha = 0.6, color = "darkblue") +
  geom_abline(slope = 1, intercept = 0, color = "red",
linetype = "dashed", size = 1) +
  geom_smooth(method = "lm", se = FALSE, color = "black")
+
  theme_bw() +
  labs(
    title = "Observed vs. Predicted (Random Forest)",
    subtitle = "Red line is the ideal 1:1 prediction",
    x = "Observed log(SOC)",
    y = "Predicted log(SOC)"
  ) +
  coord_equal()

# Create the plot for the Cubist model
p_cubist <- ggplot(results_df, aes(x = true_log_soc, y =
```

```

cubist_pred)) +
  geom_point(alpha = 0.6, color = "darkgreen") +
  geom_abline(slope = 1, intercept = 0, color = "red",
linetype = "dashed", size = 1) +
  geom_smooth(method = "lm", se = FALSE, color = "black")
+
  theme_bw() +
  labs(
    title = "Observed vs. Predicted (Cubist)",
    subtitle = "Red line is the ideal 1:1 prediction",
    x = "Observed log(SOC)",
    y = "Predicted log(SOC)"
  ) +
  coord_equal()

# Display plots side-by-side for comparison
p_rf + p_cubist

```

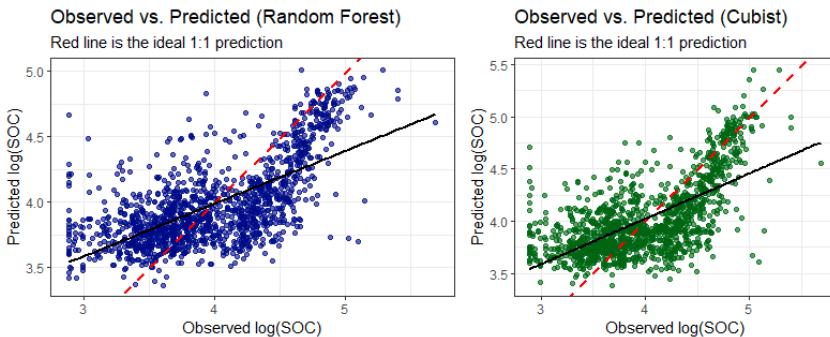


Fig. 3.9. Scatter plot "Observed vs. Predicted" for the regression model. The dots show individual predictions, the red dotted line shows perfect consistency

Graph analysis: This graph provides much more information than just the number R^2 . We can visually evaluate:

- **Overall accuracy:** How tightly the points fit the 1:1 line.

Bias: If a trend line (black solid) systematically passes above or below the red line, this indicates an overall bias.

Nonlinear errors: If the points form a curved rather than straight cloud, this may indicate that the model does not pick up relationships well in extreme ranges of values. For example, a model may work well for average SOC values, but systematically err for very low or very high values.

Such visual analysis is a mandatory addition to quantitative metrics. It allows us not only to state accuracy, but to deeply understand the behavior of our model, identify its weaknesses and, if necessary, return to the stage of feature engineering or model selection to improve the results.

13.3. Quantification of forecast uncertainty (quantile regression forests)

So far, when evaluating our regression models, we have focused on the accuracy **of the point forecast** – the single "best" value that the model produces for each observation. Metrics like RMSE tell us how wrong this point prediction is, on average. However, in the real world, in order to make informed decisions, we often need to know not only the most likely value, but also **how confident we are in that prediction**. This is **the concept of uncertainty**.

Imagine two situations:

- The model predicts a SOC content of 3.5%, and we are 90% sure that the true value lies in the range of [3.4%, 3.6%]. This is a very reliable forecast.

The model predicts the same 3.5%, but the 90% confidence interval is [1.5%, 5.5%]. This is a very unreliable forecast, although the point value is the same.

The standard Random Forest, by averaging tree predictions, loses information about the spread of these predictions and cannot provide us with this confidence interval. Fortunately, there is a powerful modification of it designed specifically for this purpose – **Quantile Regression Forests (QRF)**.

How does QRF work?

The idea of QRF is ingeniously simple. Instead of simply averaging predictions from 500 trees in a forest, QRF **stores all 500 predictions** for each new observation. Thus, for each point, we get not a single number, but a whole **empirical distribution** of probable values.

With this distribution, we can calculate any **quantile**. A quantile is a value below which a certain percentage of data lies.

- **0.05 quantile (5th percentile):** a value below which is 5% of the predictions from all trees.
- **0.5 Quantile (50th percentile):** This is **the median**, which is a more emission-tolerant alternative to the mean.
- **0.95 quantile (95th percentile):** a value below which 95% of predictions are located.

By calculating the lower (e.g., 0.05) and upper (e.g., 0.95) quantiles, we get a **90% Prediction Interval**. This is our quantification of uncertainty.

Practical implementation with ranger

To build the QRF, we will use a modern, fast ranger package, which is an efficient implementation of the Random Forest and supports quantile regression.

```
# --- Chapter 13.3: Quantifying Uncertainty with QRF ---

# 1. Train a Quantile Regression Forest model
# The key is to set the argument quantreg = TRUE
set.seed(456)
grf_model <- ranger(
  formula = log_soc ~ . - SOC_t_ha - ID,
  data = train_data,
  num.trees = 500,
  quantreg = TRUE,
  importance = "permutation"
)

# 2. Make predictions on the test set to get the
# quantiles
# We specify which quantiles we are interested in (5th,
# 50th/median, 95th)
grf_preds <- predict(grf_model, data = test_data, type =
"quantiles", quantiles = c(0.05, 0.5, 0.95))

# 3. The result is a matrix, let's convert it to a
# dataframe
grf_results_df <- as.data.frame(grf_preds$predictions)
colnames(grf_results_df) <- c("q05", "q50_median", "q95")

# 4. Combine with true values for plotting
final_grf_results <- bind_cols(
```

```

true_log_soc = test_data$log_soc,
qrf_results_df
)

```

Visualization of uncertainty

Now, having for each point not only a forecast (median), but also the limits of the confidence interval, we can create a much more informative diagnostic graph.

```

# 5. Visualize the QRF predictions with uncertainty
intervals
ggplot(final_qrf_results, aes(x = true_log_soc, y =
q50_median)) +
  # Add the 90% prediction interval as a shaded ribbon
  geom_ribbon(aes(ymin = q05, ymax = q95), fill =
"skyblue", alpha = 0.5) +
  # Add the median prediction points
  geom_point(alpha = 0.6, color = "darkblue") +
  # Add the ideal 1:1 line
  geom_abline(slope = 1, intercept = 0, color = "red",
linetype = "dashed") +
  theme_bw() +
  labs(
    title = "QRF Validation with 90% Prediction
Interval",
    x = "Observed log(SOC)",
    y = "Predicted log(SOC) (Median)"
  ) +
  coord_equal()

```

Graph analysis: This graph is extremely informative. We see not only how close the median forecast is to the ideal line, but also the **width of the predictive interval**. We gain a deep understanding of its limitations.

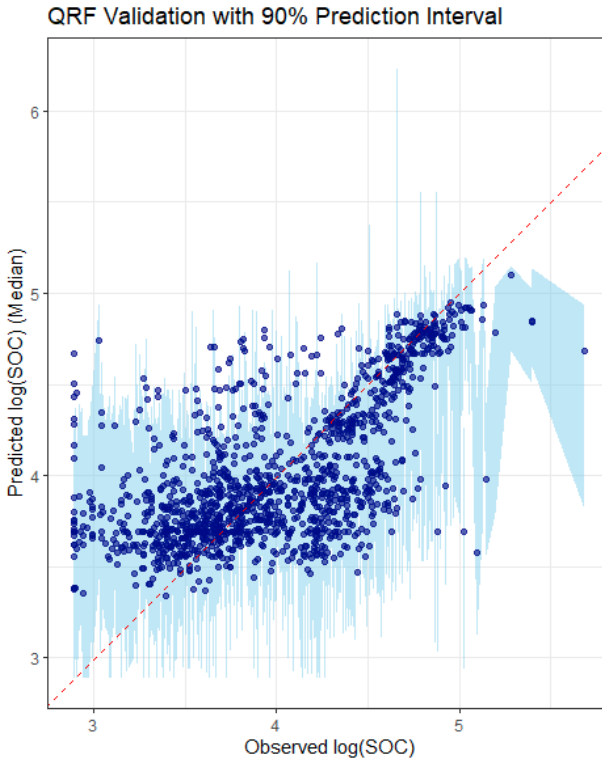


Fig. 3.10. Chart "Observed vs. Predicted" for QRF. The dots show the median forecast and the blue bar shows the width of 90% of the predictive interval, visualizing the uncertainty

The ability to quantify uncertainty is one of the greatest advantages of the modern DSM. It allows us to create not just one "best forecast" map, but a whole set of maps: a map of the median forecast, a map of the lower and upper limits of the confidence interval, and, most importantly, **a map of the width of the predictive interval**, which directly shows in which parts of our territory the forecasts are the most reliable, and where additional field research is needed.

Chapter 14. Construction of final maps and their practical application

14.1. Creating final maps (forecast, interval boundaries, uncertainty)

We have successfully trained and thoroughly validated our regression models, choosing Quantile Regression Forests (QRF) as a powerful tool that allows not only to make point predictions, but also to quantify their uncertainty. Now it's time to apply this learned model to our entire study area, turning abstract statistical dependencies into a set of concrete, spatially explicit, and extremely useful maps.

Unlike the classification, where we created a single map of predicted classes, QRF allows us to generate a whole package of cartographic products:

- **Median prediction map:** Our "best guess" for the SOC content in each pixel.

Predictive interval boundary maps: Maps of the lower (e.g., 5th percentile) and upper (e.g., 95th percentile) boundaries that delineate the range of probable values.

Uncertainty Map: A map that directly visualizes the width of the predictive interval, showing where our predictions are most reliable and where they are least certain.

Practical implementation of spatial forecasting

The spatial prediction process for regression is similar to what we did for classification. We'll be using the `predict()` function from the `terra` package, which integrates nicely with models trained with `ranger`.

```
# --- Chapter 14: Spatial Prediction (Memory-Safe
# Version) ---

# 1. Prepare the full raster stack for prediction
# First, load the boundary to mask the predictors
slovakia_boundary <-
st_read("gis_data/slovakia_boundary.gpkg")
slovakia_boundary_proj <- st_transform(slovakia_boundary,
crs = crs(full_stack))
```

```

# Mask the predictor stack to only include pixels within
the boundary
final_stack_masked <- mask(full_stack,
slovakia_boundary_proj)

# Ensure the names match the model predictors
predictor_names <-
grf_model$forest$independent.variable.names
final_stack_for_pred <-
final_stack_masked[[predictor_names]]

# 2. Spatially predict by processing the raster in chunks
(tiles)
# Create a grid of 8 tiles (4 rows, 2 columns) covering
the raster extent
tiles <- st_make_grid(st_bbox(final_stack_for_pred), n =
c(4, 2))
tile_files <- character() # To store the paths of the
predicted tiles

# 3. Loop through each chunk, predict, and save to a
temporary file
for (i in 1:length(tiles)) {
  print(paste("Processing chunk", i, "of",
length(tiles)))

  # Crop the raster stack to the chunk's extent
  chunk_raster <- crop(final_stack_for_pred, tiles[i],
snap = "out")

  # Define a temporary filename for the predicted chunk
  temp_filename <- file.path("results",
paste0("temp_tile_", i, ".tif"))

  # Spatially predict on the smaller chunk
  q_values <- c(0.05, 0.5, 0.95)
  predict(
    chunk_raster,
    grf_model,
    fun = function(model, ...) predict(model, ..., type =
"quantiles", quantiles = q_values)$predictions,
    filename = temp_filename,
    overwrite = TRUE,
    wopt = list(memfrac = 0.7, datatype = "FLT4S")
  )
}

```

```

# Store the path to the created tile file
tile_files[i] <- temp_filename
}

# 4. Merge all the predicted tiles back into a single
raster
# Create a SpatRasterCollection from the list of file
paths
tile_collection <- sprc(tile_files)

# Merge the collection into one final map
predicted_quantiles_map <- merge(tile_collection,
filename = "results/predicted_soc_quantiles.tif",
overwrite = TRUE)

# 5. Clean up temporary tile files
file.remove(tile_files)

# 6. Rename the layers of the resulting multi-layer
raster for clarity
names(predicted_quantiles_map) <- c("soc_q05",
"soc_q50_median", "soc_q95")

print("Spatial prediction of quantiles is complete.")

```

Process analysis: We have received a new, three-layer raster file. Each layer corresponds to one of the calculated quantiles. Now we can easily manipulate these layers. It is **worth warning** that this step is quite long and demanding on the RAM and processor of the computer. Therefore, it is written taking into account these features.

Uncertainty map calculation and visualization

The most direct measure of uncertainty is the width of the 90% predictive interval (PI), which is calculated as the difference between the 95th and 5th percentiles.

```

# --- Chapter 14.1: Create Final Maps ---

# 1. Calculate the Prediction Interval Width (PIW)
# This is the difference between the 95th and 5th
percentile predictions.
piw_map <- predicted_quantiles_map$soc_q95 -
predicted_quantiles_map$soc_q05

```

```

names(piw_map) <- "prediction_interval_width"

# 2. Extract the median prediction map
median_prediction_map <-
predicted_quantiles_map$soc_q50_median

# 3. Mask both maps to the country boundary for clean
visualization
# Ensure boundary is in the correct projection
slovakia_boundary_proj <- st_transform(slovakia_boundary,
crs = crs(median_prediction_map))
median_map_masked <- mask(median_prediction_map,
slovakia_boundary_proj)
piw_map_masked <- mask(piw_map, slovakia_boundary_proj)

# 4. Save the final, interpretable maps
# Save the median prediction map
writeRaster(median_map_masked,
"results/soc_log_median_prediction.tif", overwrite =
TRUE)

# Save the uncertainty map
writeRaster(piw_map_masked,
"results/soclog_uncertainty_map.tif", overwrite = TRUE)

# 5. Plot the two key maps separately with better color
schemes
# Define color palettes
prediction_palette <- brewer.pal(9, "YlGnBu")
uncertainty_palette <- brewer.pal(9, "YlOrRd")

# Plot the median prediction map
plot(median_map_masked,
      main = "Median Prediction of log(SOC)",
      col = prediction_palette)
plot(st_geometry(slovakia_boundary_proj), add = TRUE,
border = "black")

# Plot the uncertainty map
plot(piw_map_masked,
      main = "90% Prediction Interval Width",
      col = uncertainty_palette)
plot(st_geometry(slovakia_boundary_proj), add = TRUE,
border = "black")

```

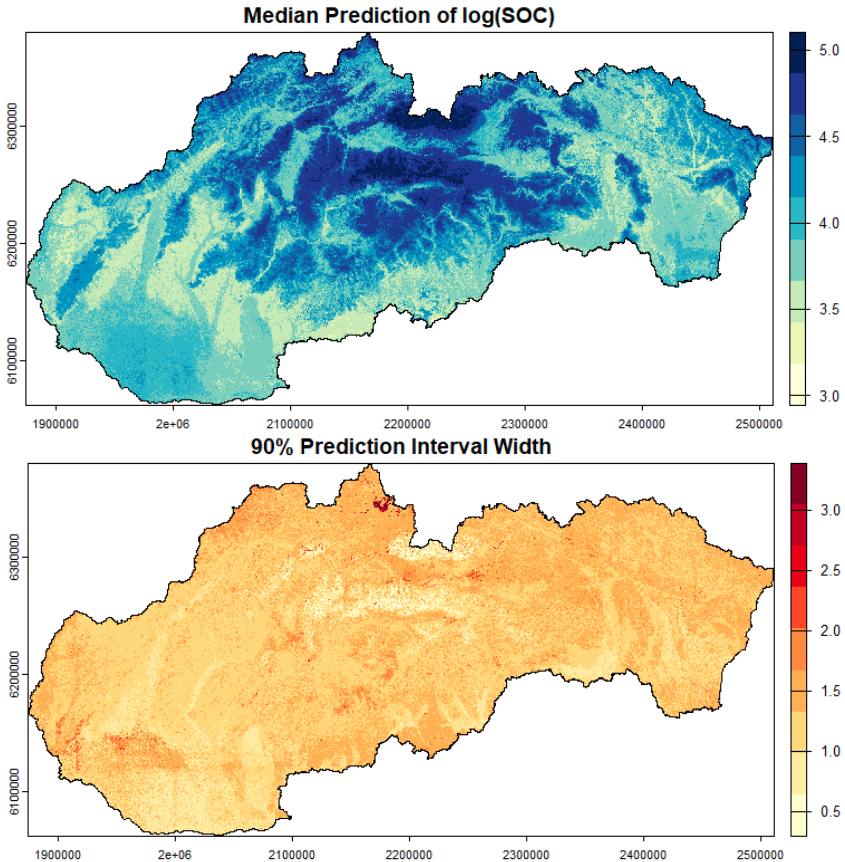



Fig. 3.11. Key results of regression modeling. Above is a map of the median forecast $\log(\text{SOC})$. Below is an uncertainty map (the width of the predictive interval), where lighter colors indicate higher confidence, and darker colors indicate higher uncertainty

Map analysis:

- **The median forecast map** shows the expected spatial patterns: higher $\log(\text{SOC})$ values in mountainous and forest areas and lower values for arable land in lowlands.
- **The uncertainty map** provides unique additional information. We can see that the uncertainty is not the same across the territory. It can be higher, for example, in high-

altitude areas or in areas with rare combinations of parent rocks and terrain, where we had little training data. This map is an invaluable tool for planning future field studies: to improve the model, new samples should be taken precisely in the areas with the highest uncertainty.

At this stage, we have received a powerful set of maps reflecting not only our knowledge of the distribution of SOC_s, but also the limits of this knowledge. However, before using them, we must perform the last, critical step – reverse transformation – to return the values to their original, interpreted units.

14.2. Reverse Conversion

At the moment, we have a complete set of spatial predictions: a map of the median value, maps of the boundaries of the predictive interval and an uncertainty map. However, there is one last but extremely important problem: all of these maps are on **a logarithmic scale**. whether he is an agronomist, an environmentalist or a politician.

To make our results useful and understandable, we must perform **a back-transformation** – returning all our predictions from the logarithmic scale to the original units of measurement (t/ha). This step is an integral part of any simulation using data transformation.

Mathematical basis

The inverse of the mathematical operation to the natural logarithm (`log()`) is **the exponent (`exp()`)**. By applying the `exp()` function to our logarithmic predictions, we return them to the original scale.

```
SOC_original=exp(log(SOC_predicted))
```

Practical implementation with terra

Due to the power of map algebra in the terra package, applying this feature to our raster layers is extremely simple. We can apply the `exp()` function directly to our entire multi-layered `SpatRaster` object.

```
# --- Chapter 14.2: Back-transformation ---
# 1. Apply the exponential function to all layers of the
```

```
raster stack
# This converts the log-transformed predictions back to
the original scale (t/ha)
backtransformed_soc_maps <- exp(predicted_quantiles_map)
```

Process analysis: We now have a new three-layer raster, where the pixel values represent:

- ✓ soc_q05: lower bound of 90% of the predictive interval for SOC, %.
- ✓ soc_q50_median: median SOC projection, %.
- ✓ soc_q95: upper limit of 90% of the predictive interval for SOC, %.

Recalculation and analysis of uncertainty in the original scale

An important point: we cannot simply exponentiate our old uncertainty map (piw_map). Due to the nonlinearity of the logarithmic function, the width of the predictive interval must be **recalculated** based on the already transformed limits.

```
# 2. Recalculate the Prediction Interval Width on the
original scale
piw_map_original_scale <-
backtransformed_soc_maps$soc_q95 -
backtransformed_soc_maps$soc_q05
names(piw_map_original_scale) <- "uncertainty_soc_t_ha"

# 3. Extract the final median prediction map
final_median_map <-
backtransformed_soc_maps$soc_q50_median
names(final_median_map) <- "Median_SOC_t_ha"

# 4. Mask the final maps to the country boundary for
clean visualization
final_median_map_masked <- mask(final_median_map,
slovakia_boundary_proj)
piw_map_original_scale_masked <-
mask(piw_map_original_scale, slovakia_boundary_proj)

# 5. Save the final, interpretable maps
# Save the median prediction map
writeRaster(final_median_map_masked,
"results/soc_median_prediction.tif", overwrite = TRUE)
```

```

# Save the uncertainty map
writeRaster(piw_map_original_scale_masked,
"results/soc_uncertainty_map.tif", overwrite = TRUE)

# 6. Plot the final, interpretable maps
# Define color palettes
prediction_palette <- brewer.pal(9, "YlGn")
uncertainty_palette <- brewer.pal(9, "YlOrRd")

# Plot the median prediction map
plot(final_median_map_masked,
      main = "Median Prediction of SOC Stock (t/ha)",
      col = prediction_palette)
plot(st_geometry(slovakia_boundary_proj), add = TRUE,
border = "black")

# Plot the uncertainty map
plot(piw_map_original_scale_masked,
      main = "90% Prediction Interval Width (t/ha)",
      col = uncertainty_palette)
plot(st_geometry(slovakia_boundary_proj), add = TRUE,
border = "black")

```

Map analysis: After the reverse conversion, we can make an important observation. If on the uncertainty map on the logarithmic scale the error spread was more or less the same (homoscedastic), then on the map on the original scale, the uncertainty is likely to be heteroscedastic. This means that **the absolute error of the forecast (in t/ha)** is significantly higher in areas with high stocks organic carbon. For example, for soil with a forecast SOC of 50 t/ha, the uncertainty may be ± 15 t/ha, while for soil with a forecast of 150 t/ha, it may reach ± 40 t/ha. This is logical and is an important characteristic of the model. Our uncertainty map now clearly shows not only where we are less certain, but also that the magnitude of this uncertainty depends on the predicted value itself.

By completing this final step, we have obtained a complete, scientifically based and ready-to-use set of cartographic materials that honestly reflects not only our knowledge of the spatial distribution of organic carbon, but also the limits of this confidence.

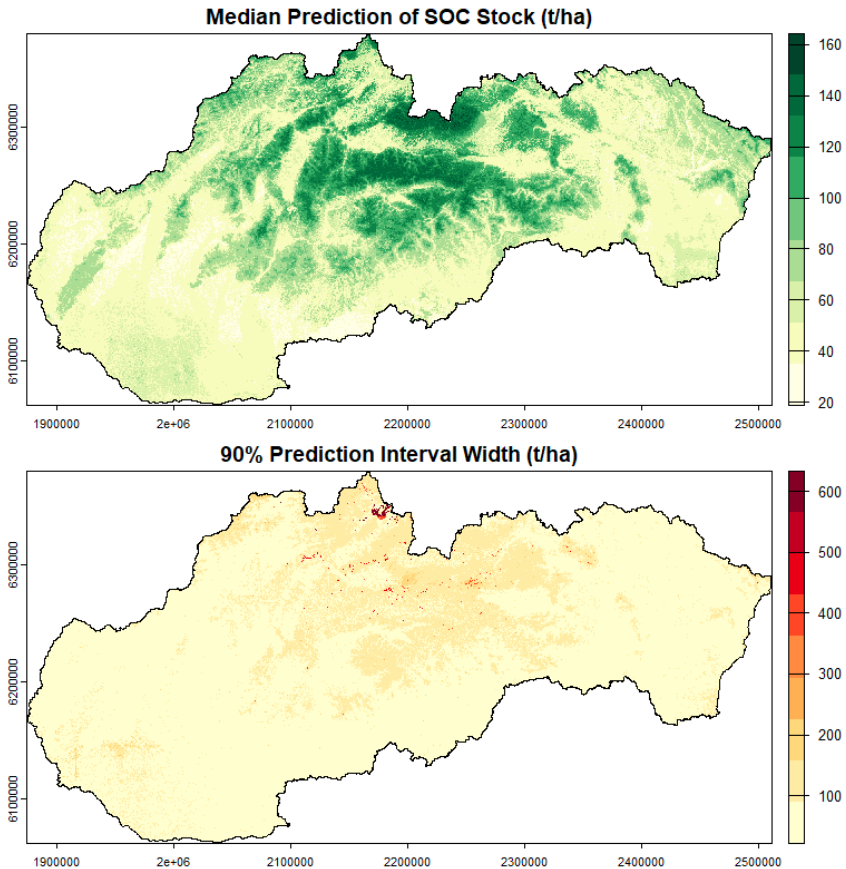


Fig. 3.12. Final cards in original units. On the left is a map of the median forecast of SOC content in t/ha for the 0-30 cm layer. On the right is an absolute uncertainty map showing the width of the predictive interval in t/ha

14.3. Practical application (estimation of carbon stocks, policy justification, inputs for models)

By creating detailed, quantified and spatially explicit maps of the organic carbon content and associated uncertainty, we have completed the technical part of our work. However, the true value of digital soil mapping is revealed when these maps become a tool for solving real-

world problems. Our maps are not just static images, but dynamic data sets that can serve as the basis for a wide range of application tasks. We consider three key areas of their practical application.

1. Organic Carbon Stocks Estimation (SOC Stocks)

Our maps show **SOC stocks** in tonnes per hectare Which is very good for many tasks, especially for national reporting under climate agreements or for assessing carbon sequestration potential.

2. Rationale for policy and sustainable governance

SOC maps and their uncertainties are a powerful tool for decision-making at the national and regional levels.

- **Identification of "hot spots":** Maps allow you to identify areas with a critically low content of organic matter that require priority restoration measures (e.g. introduction of green manure, non-dump cultivation).
- **Land-use planning:** Based on maps, recommendations for optimal land use can be developed, e.g. removing land from intensive cultivation on steep slopes with a low SOC content that are vulnerable to erosion and converting them to alkalization.
- **Monitoring and reporting:** Re-mapping at regular intervals allows you to track the dynamics of carbon content and evaluate the effectiveness of implemented agro-environmental policies. At the same time, the uncertainty map helps to properly plan the network of monitoring sites.

3. Inputs for other models

Very often, DSM products are not the end goal, but only an intermediate but critical **input layer for other, more complex models**:

- **Yield patterns:** The organic matter content is one of the key factors determining the potential yield of crops.
- **Hydrological models:** SOC affects the moisture-holding capacity of the soil, which is an important parameter for models that predict runoff, infiltration and flood risks.
- **Erosion models (e.g., USLE/RUSLE, SIMWE etc):** Organic matter improves soil structure, making it more resistant to water and wind erosion. The SOC map is an important component for

calculating the soil erosion resistance factor.

Thus, after completing this guide, we didn't just learn how to create maps. We have mastered the workflow that allows us to generate fundamental spatial information, which is the basis for many further scientific research and practical solutions in the field of natural resource management.

Conclusions

Digital Soil Mapping (DSM) has become a central tool for understanding and managing one of the Earth's most critical resources – soil. Throughout this textbook, we have presented both the conceptual foundations and the practical workflows necessary to apply predictive modelling in soil science. By combining classical soil survey principles with modern geospatial data, open-source software, and machine learning methods, DSM bridges the gap between traditional cartography and data-driven environmental science.

The practical exercises demonstrated how real-world soil data can be harmonized, integrated with covariates, and transformed into predictive models. Using R and its extensive ecosystem of packages, students and researchers are equipped to move from raw data to reproducible outputs: maps of soil properties, indicators of model uncertainty, and validation metrics that support objective decision-making. The case studies for Slovakia illustrate not only the potential of local models, but also the broader transferability of these methods to other regions, provided that environmental covariates and sampling strategies are carefully adapted.

A recurring theme across the chapters has been the importance of rigor and transparency. Reproducibility through scripting, careful harmonization of spatial data, and explicit communication of model uncertainty are not optional details, but essential practices that make DSM products credible and useful. At the same time, we emphasized the limitations inherent in any modelling approach: restricted sampling coverage, scale mismatches, and the challenges of non-stationary soil–landscape relationships. These boundaries do not diminish the value of DSM but remind us that soil maps are always generalisations, to be interpreted with caution and context.

Future Directions

Looking ahead, DSM is poised to benefit from several transformative technological shifts:

- **Artificial Intelligence (AI) and Deep Learning.** New AI architectures, including convolutional and transformer-based neural networks, have shown promise in extracting soil–landscape patterns directly from high-dimensional data such as hyperspectral imagery or time-series satellite observations. These methods can complement traditional regression and ensemble approaches by capturing more complex, non-linear dependencies.
- **Remote Sensing Data Explosion.** The availability of high-resolution global datasets from satellites (e.g., Sentinel, Landsat, PlanetScope) and UAV platforms will dramatically expand the pool of covariates available for soil modelling. Near-real-time monitoring of vegetation, moisture, and land-use change can improve the temporal alignment of covariates with soil sampling campaigns, reducing bias and enabling dynamic soil property mapping.
- **Cloud Computing and Big Data Frameworks.** Platforms such as Google Earth Engine, Google Cloud, TensorFlow, Amazon Web Services, OpenEO, and high-performance R/Python libraries allow the analysis of massive raster archives without the constraints of local hardware. This will make national and continental DSM products more feasible and reproducible.
- **Integration with Process-Based Models.** Hybrid approaches that combine machine learning with mechanistic soil models (e.g., water and carbon cycling simulations) are expected to yield more robust predictions, especially under changing climate conditions.

These advances will not eliminate the need for careful sampling design, uncertainty quantification, or context-specific interpretation, but they will broaden the scope and scalability of DSM applications.

It is our hope that this textbook will serve as both a **practical manual** and a **starting point** for deeper exploration. By engaging with the methods presented here, readers are encouraged not only to reproduce the examples, but also to adapt them creatively to their own landscapes, questions, and datasets. In doing so, they will contribute to a growing community of practice that uses data science to better understand and steward the soils on which all terrestrial life depends.

APPENDIX A: List of recommended R packages

Throughout this tutorial, we have used a number of R packages, each of which plays a key role in the digital soil mapping workflow. This app provides a generalized list of these packages with a brief description of their primary purpose. Installing these packages at the beginning of your project will provide you with all the tools you need.

Data Manipulation and Visualization

dplyr – Provides an intuitive grammar of data transformation through verbs such as `select`, `filter`, `mutate`, `group_by`, and `summarise`, enabling efficient manipulation of tabular data (Wickham et al., 2025a).

tidyverse – A meta-package that bundles key packages for modern data science, including `dplyr`, `ggplot2`, and `readr`, offering a coherent and consistent workflow (Wickham et al., 2025b).

ggplot2 – Implements the Grammar of Graphics, allowing for both exploratory data visualization and high-quality publication graphics (Wickham, 2016).

readr – Provides fast and efficient functions for importing and exporting delimited text files such as CSV (Wickham et al., 2023).

readxl / writexl – Support reading and writing of Microsoft Excel files (.xls, .xlsx), making integration with common spreadsheet formats straightforward (Wickham & Bryan, 2023; Ooms, 2023).

corrplot – Specialized in visualizing correlation matrices in the form of correlograms, offering a clear way to explore relationships among variables (Wei & Simko, 2021).

patchwork – Adds a simple grammar for combining multiple `ggplot2` plots into coherent multi-panel figures (Pedersen, 2023).

RColorBrewer – Supplies high-quality, perceptually robust colour palettes curated for thematic cartography and visualization (Neuwirth, 2022).

Working with spatial data

sf (Simple Features) – Provides a standardized and modern approach to working with vector spatial data (points, lines, polygons), fully compatible with tidyverse workflows (Pebesma, 2018).

terra – A fast and efficient package for working with raster data, supporting chunk-based processing and large-area analysis that may exceed available RAM (Hijmans, 2025).

Modeling and validation

rsample – Part of the tidymodels ecosystem, offering flexible resampling infrastructure for partitioning data into training and test sets, with stratification support (Kuhn & Wickham, 2020).

rpart – Implements classical recursive partitioning for building decision trees in classification and regression tasks (Therneau & Atkinson, 2019).

rpart.plot – Enhances the visualization of trees built with rpart, improving interpretability (Milborrow, 2022).

randomForest – A well-established implementation of the Random Forest algorithm for classification and regression (Liaw & Wiener, 2002).

ranger – A modern, fast C++ implementation of Random Forests, also supporting Quantile Regression Forests for uncertainty estimation (Wright & Ziegler, 2017).

Cubist – Builds rule-based regression models that combine accuracy with interpretability, extending beyond decision trees (Kuhn & Quinlan, 2025).

caret – Provides a unified framework for machine-learning workflows, here used primarily to build confusion matrices and compute associated accuracy metrics (Kuhn, 2008).

yardstick – Part of tidymodels, offering consistent functions for computing performance metrics in classification and regression (Kuhn & Vaughan, 2023).

APPENDIX B: Data sources for the example of Slovakia

During the practical part of this manual (Parts II and III), we used a geospatial data set for the territory of Slovakia. Although some of this data has been modified or simplified for educational purposes, it is based on real, publicly available sources. This appendix provides an overview and links to primary sources that can be used for similar research. **This set of materials (source files and results) can be downloaded from the following permanent links:**

Cherlinka, V., Gallay, M., & Dmytruk, Y. (2025). Predictive Modeling of Soil Types and Their Characteristics - supplementary data [Data set]. in Predictive Modeling of Soil Types and Their Characteristics (1st ed., 198 p). Zenodo. <https://doi.org/10.5281/zenodo.16926392>

Administrative boundaries

The vector layer with the borders of Slovakia was obtained from the **GADM (Database of Global Administrative Areas) database**. It is a high-quality, publicly available resource that provides administrative boundaries for all countries of the world at several levels of detail.

Source: GADM (<https://gadm.org/>)

Format: GeoPackage, Shapefile, R Spatial objects (.rds)

https://geodata.ucdavis.edu/gadm/gadm4.1/shp/gadm41_SVK_shp.zip

https://geodata.ucdavis.edu/gadm/gadm4.1/gpkg/gadm41_SVK.gpkg

Level of detail: Level 0 (national border) has been used for this guide.

Point soil data

A set of point data on soil types and organic carbon content used in the manual will be generated from two different sources, by randomized sampling from original vector or raster maps, in particular soil organic carbon reserves were obtained from data from the GSOCmap project. This dataset was created with the initial purpose of demonstrating the workflow and, we believe, has fully fulfilled its task. Despite this, the following sources can be used for more detailed research:

LUCAS Soil Database: A Eurostat project that provides harmonized data on topsoil properties for thousands of points across the European Union. It is one of the most important sources for large-

scale digital mapping.

Source: European Soil Data Centre (ESDAC)
(<https://esdac.jrc.ec.europa.eu/>)

National Soil Services: More detailed data can usually be obtained from national soil or geological institutes. For Slovakia, such a body is the National Agricultural and Food Center (NPPC) - Research Institute for Soil Science and Soil Protection.

Digital Elevation Model (DEM)

The DMR3.5 digital elevation model was used as the basis for all morphometric covariates (height, slope, exposure, etc.). The digital relief model DMR3.5 was created for the purpose of creating layers for the cartographic representation of the elevation in accordance with ZBGIS data. DMR3.5 is based on the original DMR3 model supplemented with recalculated areas of flat parts of lowlands, basins and valleys of large rivers. Before the creation of DMR3.5, 3D geodatabase ZBGIS collected by the photogrammetric method was selected as the input data source. The output format is ESRI GRID with a resolution of 10 m/pixel, 25 m/pixel, 50 m/pixel and 100 m/pixel..

Source: DMR3.5 <https://rpi.gov.sk/en/metadata/0e6e1625-5c59-4ea4-a483-dc459fbff20b>

Resolution: 10 m/pixel, 25 m/pixel, 50 m/pixel and **100 m/pixel**.

Geological map

The rasterized geological map used as a predictor of the parent rock is based on data normally provided by national geological surveys. For Slovakia, such a source is the **Dioniz Štúr State Geological Institute (Štátny geologický ústav Dionýza Štúra, ŠGÚDŠ)**.

Source: ŠGÚDŠ (<https://www.geology.sk/>)

Note: For use in DSM, vector geologic maps require pre-processing, including rasterization and bringing to a single classification system suitable for modeling.

APPENDIX C: Glossary of Terms

Bagging (Bootstrap Aggregating) – An ensemble machine learning method that creates multiple training subsamples from the original dataset by random selection with replacement. A separate model is trained on each subsample, and the final prediction is obtained by averaging (regression) or voting (classification).

Bootstrap – Resampling with replacement from a dataset, forming the basis for Bagging and other ensemble methods.

Classification – A type of supervised machine learning task where the goal is to predict a categorical outcome (class). For example, predicting soil type.

Confusion Matrix – A contingency table used to assess classification accuracy by comparing predicted and true classes. It forms the basis for most accuracy metrics (e.g., overall accuracy, Kappa, precision, recall).

Coordinate Reference System (CRS) – A coordinate system that defines how two-dimensional map coordinates correspond to real-world locations on the Earth’s surface.

Covariate (Predictor, Independent Variable) – In DSM, a spatial layer (typically raster) representing a factor of soil formation (e.g., elevation, slope, land cover) used as an explanatory variable in models.

Cross-validation (CV) – A resampling method where data are repeatedly split into training and validation folds to provide robust accuracy estimates.

Data Frame – A basic R structure for storing tabular data, where rows represent observations and columns may contain variables of different data types.

Data “tidy” (Tidy Data) – A concept of data organization where each row corresponds to an observation, each column to a variable, and each table to one type of observational unit. It underpins the tidyverse philosophy.

Decision Tree – A machine learning algorithm that models data using a hierarchical structure of decision rules (“if–then” statements) resembling a branching tree.

Ensemble Learning – A family of machine learning methods that combine multiple base models to improve predictive accuracy and stability (e.g., Bagging, Random Forests, Boosting).

Extraction – The process of retrieving values from raster layers at the locations of vector objects (usually points), a key step in linking covariates with soil observations in DSM.

Extent – The spatial coverage of a geospatial dataset, defined by its minimum and maximum X and Y coordinates.

Harmonization – The process of aligning all raster layers (covariates) to a common spatial grid with the same coordinate system, extent, and resolution. Essential before spatial modelling.

Kappa Coefficient (κ) – A statistical measure of classification accuracy that adjusts for agreement occurring by chance. It is particularly useful for imbalanced datasets.

Machine Learning (ML) – A branch of artificial intelligence that develops algorithms capable of learning patterns from data and making predictions without explicit programming.

Overfitting – A situation where a model becomes overly complex and fits noise or random patterns in the training data, reducing its ability to generalize to new data.

Package (in R) – A standardized collection of functions, datasets, and documentation designed to extend R with specialized functionality.

Pipeline Operator ($\%>\%$) – An operator from the magrittr package that passes the output of one function directly into the input of the next, enabling readable and concise workflows.

R^2 (Coefficient of Determination) – A regression metric that indicates the proportion of variance in the dependent variable explained by the model.

Raster – A spatial data model that represents geographic phenomena as a regular grid of cells (pixels), each holding a value (e.g., elevation, temperature).

Regression – A type of supervised machine learning task where the goal is to predict a continuous numerical variable, such as soil organic carbon content.

RMSE (Root Mean Squared Error) – A regression accuracy metric that measures the average squared difference between predicted and observed values, expressed in the units of the target variable.

Random Forest – A widely used ensemble learning algorithm for classification and regression. It builds multiple decision trees using random subsets of data and predictors, then combines their results by majority vote (classification) or averaging (regression).

SCORPAN – A mnemonic acronym representing soil formation factors extended for DSM: Soil, Climate, Organisms, Relief, Parent material, Age, and spatial Position.

sf (Simple Features) – An R package implementing the OGC Simple Features standard for vector spatial data (points, lines, polygons). It integrates well with tidyverse workflows.

terra – An R package for efficient and large-scale raster data analysis, supporting spatial modelling and operations that exceed available RAM by using chunk-based processing.

Tidyverse – A collection of R packages (including dplyr, ggplot2, readr) that share a consistent grammar, design philosophy, and data structures for modern data science.

Training / Test Split – The practice of dividing data into a training set (for building a model) and a test set (for evaluating model performance on unseen data).

Uncertainty – A measure of confidence in model predictions. In regression, often expressed as prediction intervals; in classification, as class probabilities.

Validation – The process of objectively evaluating a model's accuracy and reliability, usually performed on test data not used for training.

Vector – A basic R data structure consisting of an ordered sequence of elements of the same type (numeric, character, or logical).

REFERENCES

1. Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5–32. <https://doi.org/10.1023/A:1010933404324>
2. Cohen, J. (1960). A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1), 37–46. <https://doi.org/10.1177/001316446002000104>
3. Congalton, R. G. (1991). A review of assessing the accuracy of classifications of remotely sensed data. *Remote Sensing of Environment*, 37(1), 35–46. [https://doi.org/10.1016/0034-4257\(91\)90048-B](https://doi.org/10.1016/0034-4257(91)90048-B)
4. Global Soil Partnership. (2022). *Global soil organic carbon map – GSOCmap v. 1.6: Technical report*. FAO.
5. Hengl, T., & MacMillan, R. A. (Eds.). (2019). *Predictive soil mapping with R*. OpenGeoHub Foundation.
6. Heung, B., Saurette, D., & Bulmer, C. E. (2021). Digital soil mapping. In *Digging into Canadian soils* (pp. 313–327). Canadian Society of Soil Science.
7. Hijmans, R. J. (2025). *terra: Spatial data analysis* (Version 1.8-62) [R package]. <https://github.com/rspatial/terra>
8. James, G., Witten, D., Hastie, T., & Tibshirani, R. (2013). *An introduction to statistical learning: With applications in R*. Springer. <https://doi.org/10.1007/978-1-4614-7138-7>
9. Kabacoff, R. I. (2021). *R in action* (3rd ed.). Manning Publications.
10. Kuhn, M. (2008). Building predictive models in R using the caret package. *Journal of Statistical Software*, 28(5), 1–26. <https://doi.org/10.18637/jss.v028.i05>
11. Kuhn, M., & Quinlan, R. (2025). *Cubist: Rule-based regression modeling* (Version 0.4) [R package]. <https://topepo.github.io/Cubist/>
12. Kuhn, M., & Vaughan, D. (2023). *yardstick: Tidy characterizations of model performance* (Version 1.2.0) [R package]. <https://yardstick.tidymodels.org>
13. Kuhn, M., & Wickham, H. (2020). *rsample: General resampling infrastructure* (Version 1.1.1) [R package]. <https://rsample.tidymodels.org>
14. Landis, J. R., & Koch, G. G. (1977). The measurement of observer agreement for categorical data. *Biometrics*, 33(1), 159–174. <https://doi.org/10.2307/2529310>
15. Liaw, A., & Wiener, M. (2002). Classification and regression by randomForest. *R News*, 2(3), 18–22.
16. Lovelace, R., Nowosad, J., & Muenchow, J. (2019). *Geocomputation with R*. CRC Press. <https://doi.org/10.1201/9780203730058>
17. Malone, B. P., Minasny, B., & McBratney, A. B. (2017). *Using R for digital soil mapping*. Springer. <https://doi.org/10.1007/978-3-319-44327-0>
18. McBratney, A. B., Mendonça Santos, M. L., & Minasny, B. (2003). On digital soil mapping. *Geoderma*, 117(1–2), 3–52. [https://doi.org/10.1016/S0016-7061\(03\)00223-4](https://doi.org/10.1016/S0016-7061(03)00223-4)

19. Milborrow, S. (2022). *rpart.plot: Plot 'rpart' models: An enhanced version of plot.rpart* (Version 3.1.0) [R package]. <https://cran.r-project.org/package=rpart.plot>
20. Neuwirth, E. (2022). *RColorBrewer: ColorBrewer palettes* (Version 1.1-3) [R package]. <https://cran.r-project.org/package=RColorBrewer>
21. Ooms, J. (2023). *writexl: Export data frames to Excel 'xlsx' format* (Version 1.4.2) [R package]. <https://cran.r-project.org/package=writexl>
22. Pebesma, E. (2018). Simple features for R: Standardized support for spatial vector data. *The R Journal*, 10(1), 439–446. <https://doi.org/10.32614/RJ-2018-009>
23. Pebesma, E., & Bivand, R. (2023). *Spatial data science: With applications in R*. Chapman & Hall/CRC. <https://doi.org/10.1201/9780429459016>
24. Pedersen, T. L. (2023). *patchwork: The composer of plots* (Version 1.1.3) [R package]. <https://cran.r-project.org/package=patchwork>
25. R Core Team. (2023). *R: A language and environment for statistical computing*. R Foundation for Statistical Computing. <https://www.R-project.org>
26. Therneau, T., & Atkinson, E. (2019). *rpart: Recursive partitioning and regression trees* (Version 4.1-15) [R package]. <https://cran.r-project.org/package=rpart>
27. Wei, T., & Simko, V. (2021). *corrplot: Visualization of a correlation matrix* (Version 0.92) [R package]. <https://cran.r-project.org/package=corrplot>
28. Wickham, H. (2016). *ggplot2: Elegant graphics for data analysis*. Springer. <https://doi.org/10.1007/978-3-319-24277-4>
29. Wickham, H., & Bryan, J. (2023). *readxl: Read Excel files* (Version 1.4.3) [R package]. <https://cran.r-project.org/package=readxl>
30. Wickham, H., François, R., Henry, L., Müller, K., & Vaughan, D. (2025a). *dplyr: A grammar of data manipulation* (Version 1.1.4) [R package]. <https://dplyr.tidyverse.org>
31. Wickham, H., François, R., Henry, L., Müller, K., & Vaughan, D. (2025b). *tidyverse: Easily install and load the tidyverse* (Version 2.0.0) [R package]. <https://tidyverse.tidyverse.org>
32. Wickham, H., Grolemund, G. (2017). *R for data science: Import, tidy, transform, visualize, and model data*. O'Reilly Media.
33. Wickham, H., Hester, J., & François, R. (2023). *readr: Read rectangular text data* (Version 2.1.4) [R package]. <https://cran.r-project.org/package=readr>
34. Wilkinson, L. (2005). *The grammar of graphics* (2nd ed.). Springer. <https://doi.org/10.1007/0-387-28695-0>
35. Wright, M. N., & Ziegler, A. (2017). ranger: A fast implementation of random forests for high dimensional data in C++ and R. *Journal of Statistical Software*, 77(1), 1–17. <https://doi.org/10.18637/jss.v077.i01>
36. Yan, F., Shangguan, W., Zhang, J., & Hu, B. (2020). Depth-to-bedrock map of China at a spatial resolution of 100 meters. *Scientific Data*, 7(1), 2. <https://doi.org/10.1038/s41597-020-0372-7>
37. Zhong, S. H., Liu, Y., Li, S. Z., & Hu, B. (2023). A machine learning method for distinguishing detrital zircon provenance. *Contributions to Mineralogy and Petrology*, 178(35). <https://doi.org/10.1007/s00410-023-02017-9>

Predictive Modeling of Soil Types and Their Characteristics

University Textbook

Authors:

doc. Vasyl Cherlinka, DrSc.; doc. Mgr. Michal Gallay, PhD.; Prof. Yuriy Dmytruk, DrSc.

Publisher:

Pavol Jozef Šafárik University in Košice,
ŠafárikPress

Year of publication: 2025**Number of pages:** 204**Extent:** 7,6 author sheets**Edition:** First

This book was funded by the European Union's NextGenerationEU through the Recovery and Resilience Plan for Slovakia (project No. 09I03-03-V01-00049).



**Funded by the
European Union**
NextGenerationEU

**[RECOVERY
AND RESILIENCE]
PLAN**