

Maurizio Petrelli

Machine Learning for Earth Sciences

Using Python to Solve Geological Problems

December 26, 2022

Springer Nature

To my family and friends

Preface

“Machine Learning for the Earth Sciences” provides Earth Scientists with a progressive partway from zero to Machine learning, with examples in Python aimed at the solution of geological problems. The book is devoted to Earth Scientists, at any level, from students to academics and professionals who would like to be introduced to Machine Learning. Basic knowledge of Python programming is demanded to successfully benefit from this book. If you are a complete novice to Python, I suggest you to start with Python introductory reads like “Introduction to Python in Earth Science Data Analysis”¹ or similar lectures. “Machine Learning for the Earth Sciences” is divided into five parts and it attempts to be geologist friendly. Machine Learning mathematics is gently provided and technical parts limited to the essentials. Part I introduces the basics of machine learning with a geologist-friendly language. It starts by introducing definitions, terminology, and fundamental concepts (e.g., the types of learning paradigms). Then shows how to set up a Python environment for Machine Learning applications and it finally describes the typical Machine Learning workflow. Part II and III are about unsupervised and supervised learning, respectively. They start describing some widely-used algorithms. Then, they provide examples of applications to Earth Sciences like the clustering and dimensionality reduction in petro-volcanological applications, the clustering of multi-spectral data, well-log data facies classification, and machine learning regression in petrology. Part IV deals with the scaling of machine learning applications. When your PC starts suffering from the dimension of the data set or the complexity of the model, you need scaling! Finally, Part V introduces deep learning. It starts describing the PyTorch library and it provides an example application to Earth Sciences. If you are working in Earth Science and would like to start exploiting the power of Machine learning in your projects, this is the right place for you.

Assisi, 17-12, 2022

Maurizio Petrelli

¹ <https://bit.ly/python-mp>

Acknowledgments

I would like to acknowledge all the people who encouraged me again when I decided to start this new challenging adventure, arriving just after the end of a satisfying but extremely strenuous one, i.e., the book titled “Introduction to Python in Earth Science Data Analysis: From Descriptive Statistics to Machine Learning.” First, I would like to thank my colleagues at the Department of Physics and Geology, University of Perugia. Also, I thank the Erasmus Plus (E+) program that supported my new foreign teaching excursions in Hungary, Azores, and Germany. Professor Francois Holtz (Leibniz Universität Hannover), José Manuel Pacheco (Universidade dos Açores), and Professor Szabolcs Harangi (Eötvös University Budapest) are also kindly acknowledged for allowing me to run the “Introduction to Machine Learning” courses at their institutions. In addition, I thank J. ZhangZhou (Zhejiang University) and Kunfeng Qiu (China University of Geosciences) who invited me in making talks and short courses on topics related to the application of Machine learning to Earth Sciences. I also give my heartfelt thanks to my family, who, one more time, put up with me as I wrote this book.

Overview

Let me introduce myself

Hi and welcome. My name is Maurizio Petrelli and I currently work at the Department of Physics and Geology, University of Perugia (UniPg). My research focuses on the petrological characterization of volcanoes with an emphasis on the dynamics and timescales of pre-eruptive events. For this work, I combine classical and unconventional techniques. Since 2002, I've worked intensely in the laboratory, mainly focusing on the development UniPg's facility for Laser Ablation Inductively Coupled Plasma Mass Spectrometry (LA-ICP-MS). In February 2006, I obtained my Ph.D. degree with a thesis entitled "Nonlinear Dynamics in Magma Interaction Processes and their Implications on Magma Hybridization." Since December 2021, I am Associate Professor at Department of Physics and Geology at UniPg. Currently, I am developing a new line of research for applying Machine Learning techniques in Geology. Finally, I also manage the LA-ICP-MS laboratory at UniPg.

Styling conventions

I use conventions throughout this book to identify different types of information. For example, Python statements, commands, and variables used within the main body of the text are set in italics. A block of Python code is highlighted as follows:

```
1 import numpy as np
2
3 def sum(a,b):
4     return a + b
5
6 c = sum(3,4)
```

Shared codes

All code presented in this book is tested on the Anaconda Individual Edition ver. 2020.11 (Python 3.8.5) and is available at my GitHub repository (🔗 petrelli-m):

🔗 http://bit.ly/ml_earth_sciences

Involvement and collaborations

I am always open to new collaborations worldwide. Feel free to contact me by mail to discuss new ideas or propose a collaboration. You can also reach me through my personal website or by Twitter. I love sharing the content of this book in short courses everywhere. If you are interested, please contact me to organize a visit to your institution.

Personal contacts:

✉ maurizio.petrelli@unipg.it

🐦 [@mauripetre](https://twitter.com/mauripetre)

🌐 <https://www.mauriziopetrelli.info>

Contents

Overview	xi
Part I Basic Concepts of Machine Learning for Earth Scientists	
1 An Introduction to Machine Learning	3
1.1 Machine Learning: definitions and terminology	3
1.2 The Learning Process	4
1.3 Supervised Learning	5
1.4 Unsupervised Learning	7
1.5 Semi-Supervised Learning	9
2 Setting Up your Python Environments for Machine Learning	11
2.1 Python Modules for Machine Learning	11
2.2 A Local Python Environment for Machine Learning	11
2.3 ML Python Environments on Remote Linux Machines	13
2.4 Working with your Remote Instance	19
2.5 Preparing Isolated Deep Learning Environments	21
2.6 Cloud Based Machine Learning Environments	23
2.7 Speed Up your ML Python Environment	24
3 Machine Learning Workflow	29
3.1 Machine Learning Step-by-Step	29
3.2 Get your Data	30
3.3 Data Pre-Processing	33
3.4 Train a Model	44
3.5 Model Validation and Testing	48
3.6 Model Deploy and Persistence	55
Part II Unsupervised Learning	

4	Unsupervised Machine Learning Methods	61
4.1	Unsupervised Algorithms	61
4.2	Principal component Analysis	61
4.3	Manifold Learning	62
4.3.1	Isometric Feature Mapping	63
4.3.2	Locally Linear Embedding	63
4.3.3	Laplacian Eigenmaps	63
4.3.4	Hessian Eigenmaps	64
4.4	Hierarchical Clustering	64
4.5	DBSCAN	65
4.6	Mean Shift	65
4.7	K-Means	66
4.8	Spectral Clustering	66
4.9	Gaussian Mixture Models	67
5	Clustering and Dimensionality Reduction in Petrology	69
5.1	Unveil the Chemical Record of a Volcanic Eruption	69
5.2	Geological Setting	71
5.3	The investigated data set	72
5.4	Data Pre-Processing	72
5.5	Clustering analyses	77
5.6	Dimensionality Reduction	80
6	Clustering of Multi-Spectral Data	83
6.1	Spectral Data from Earth-Observing Satellites	83
6.2	Import Multi-spectral Data in Python	84
6.3	Descriptive Statistics	88
6.4	Pre-processing and Clustering	91
Part III Supervised Learning		
7	Supervised Machine Learning Methods	97
7.1	Supervised Algorithms	97
7.2	Naive Bayes	97
7.3	Quadratic and Linear Discriminant Analysis	99
7.4	Linear and Nonlinear Models	100
7.5	Loss Functions, Cost Functions, and Gradient Descent	102
7.6	Ridge Regression	106
7.7	Least Absolute Shrinkage and Selection Operator (LASSO)	107
7.8	Elastic-Net	107
7.9	Support Vector Machines	108
7.10	Supervised Nearest Neighbors	110
7.11	Trees Based Methods	111

8	Well Log Data Facies Classification by Machine Learning	113
8.1	Motivation	113
8.2	Inspection of the Data Sets and Pre-Processing	114
8.3	Model Selection and Training	126
8.4	Final evaluation	132
9	Machine Learning Regression in Petrology	139
9.1	Motivation	139
9.2	The LEPR data set and data pre processing	139
9.3	Compositional data analysis	147
9.4	Model training and error assessment	151
9.5	Results Evaluation	153
Part IV Scaling Your Machine Learning Models		
10	Parallel Computing and Scaling with Dask	159
10.1	Warming Up: Basic Definitions	159
10.2	Basics of Dask	160
10.3	‘Eager’ computation Vs. ‘Lazy’ evaluation	167
10.4	Diagnostic and feedback	173
11	Scale Your Models in the Cloud	175
11.1	How to Scale your environment in the Cloud	175
11.2	Scaling in the Cloud: the Hard Way	176
11.3	Scaling in the Cloud: the Easy Way	178
Part V Next Step: Deep Learning		
12	Introduction to Deep Learning	189
12.1	What does Deep Learning mean?	189
12.2	PyTorch	191
12.3	PyTorch Tensors	191
12.4	Structuring a feedforward network in PyTorch	194
12.5	How to train a feedforward network	195
12.5.1	The universal approximation theorem	195
12.5.2	Loss Functions in PyTorch	195
12.5.3	The Back-Propagation and its implementation in PyTorch ..	195
12.5.4	Optimization	197
12.5.5	Network Architectures	198
12.6	Example Application	201
References and Further Readings		206

Part I
Basic Concepts of Machine Learning for
Earth Scientists

Chapter 1

An Introduction to Machine Learning

1.1 Machine Learning: definitions and terminology

Shai and Shai (2014) define Machine Learning (ML) as “the automated detection of meaningful patterns in data.” It is a broad definition, so I am going to constrain it better by providing more definitions by different authors (e.g., Géron, 2017; Jordan and Mitchell, 2015; Murphy, 2012; Samuel, 1959).

As an example, Murphy (2012) defines ML as the application of algorithms and methods to detect patterns in large data sets and the use these patterns to predict future trends, to classify, or to make other types of strategic decisions.

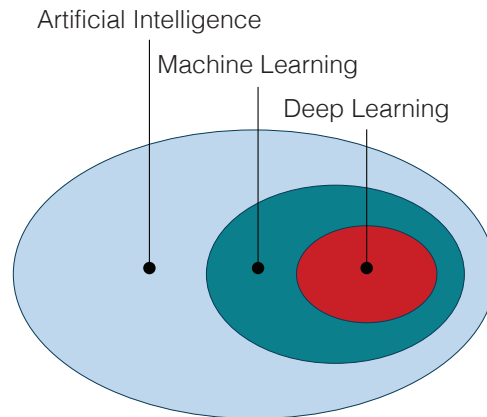


Fig. 1.1 Artificial Intelligence, Machine Learning, and Deep Learning.

In one of the earliest attempts, Samuel (1959) outlined one of the ML goals, i.e., a computer that can learn how to solve a specific task, without being explicitly

programmed. We can also take advantage of a more formal definition by Mitchell (1997): “A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E.” But what is the experience for a computer program? In Physical Sciences, the experience for a computer program almost always coincides with data. As a consequence, we can reword the definition by Mitchell (1997) to: A computer program is said to learn from data D with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with the analyses of D.

One shared feature of ML methods is that, in contrast to more traditional uses of computers, in these cases, due to the complexity of the patterns that need to be detected, a human programmer cannot provide an explicit, fine-detailed specification of how such tasks should be executed (Shai & Shai, 2014).

Using the set theory, we can define ML as a subset of Artificial Intelligence (AI), i.e., the effort to automate intellectual tasks normally performed by humans (Chollet, 2021). Please note that AI defines a broad domain retaining both machine learning and deep learning. However, the AI set also includes many other different approaches and techniques, some of those not involving the process of learning.

Summarizing, we can point to some key features of ML algorithms:

- ML is a subset of AI;
- ML methods try to extract meaningful patterns in your data set;
- ML algorithms are not explicitly programmed to solve a specific task;
- The learning process is a fundamental task in ML;
- ML methods learn from data;
- DL is a subset of ML.

When we start a new discipline, the first task consists in learning the basic concepts and terminology. Table 1.1 reports a basic glossary, useful for a Geo-scientist. It is minded to start familiarizing the “language” used by data scientists, which is often difficult and sometimes misleading for a novice.

1.2 The Learning Process

As stated above, ML algorithms are not programmed to process a conceptual model defined *a priori* but instead they attempt to uncover the complexities of large data sets through a so-called learning process (Bishop, 2007; Shai & Shai, 2014). In other words, the main goal of ML algorithms is to transform experience, i.e., data, into “knowledge” (Shai & Shai, 2014).

To better understand, we can compare the learning process of ML algorithms to that of humans.

For example, humans begin learning how to use the alphabet by observing the world around them where they find sounds, written letters, words, or phrases. Then, at school, they understand the significance of the alphabet and how to combine the

Table 1.1 ML, Basic terminology. For a detailed glossary, please refer to the online ML course by GoogleTM: <https://bit.ly/mlglossary>.

Term	Description
Tensor	A tensor is a multi-dimensional array.
Feature	It is an input variable used by machine learning algorithms.
Attribute	It is often used as synonym feature.
Label	It consists of the correct “answer” or “result” for a specific input tensor.
Observation	An observation is a synonym for instance and example. It is a row of your data set, characterized by one or more features. In labeled data sets, observations also contains a label. In a geo-chemical data set, an observation consists of one sample.
Class	A calss is a set of observations, characterized by the same label.
Prediction	It is the output of a machine learning algorithms for a specific input observation.
Model	A model in machine learning is what a machine learning algorithm has learned after the training.
Training a model	The process of determining the best model. Is is a synonymous of learning process.
Training data set	The subset of the investigated data set used to train a model during the learning process.
Validation data set	The subset of the investigated data set used to validate a model during the learning process.
Test data set	An independent data set used to test a model after the validation process.

different letters. Similarly, ML algorithms use the training data to learn significant patterns. Then, they use the learned expertise to provide an output (Shai & Shai, 2014). One of the ways to classify ML algorithms relies on the degree of “supervision”, i.e., supervised, unsupervised, and semi-supervised (i.e., Shai and Shai, 2014)

1.3 Supervised Learning

The training of ML methods, characterized by supervised learning, always provides both the input data and the desired solutions, i.e. the label, to the algorithm. Examples of applications of supervised learning are regression and classification.

In classification tasks (Fig. 1.2 A and B), ML algorithms try assigning a new observation to a specific class, i.e., a set of instances characterized by the same label (Lee, 2019). If you do not understand some terms, please refer to Table 1.1. In regression problems (Fig. 1.2 C and D), ML algorithms try guessing the value for one or more dependent variables, in response to an observation.

We will discuss extensively the application of regression and classification tasks in Earth Science problems later in the book. However, Fig. 1.2 provides an outline of two geological examples of supervised learning in the field of classification and regression. They are the identification of the volcanic source using glass shard compositions, i.e., a typical problem in tephrostratigraphy and tephrochronology (Lowe, 2011), and the retrieving of magma storage temperatures based on clinopyroxenes chemistry (Petrelli et al., 2020), respectively.

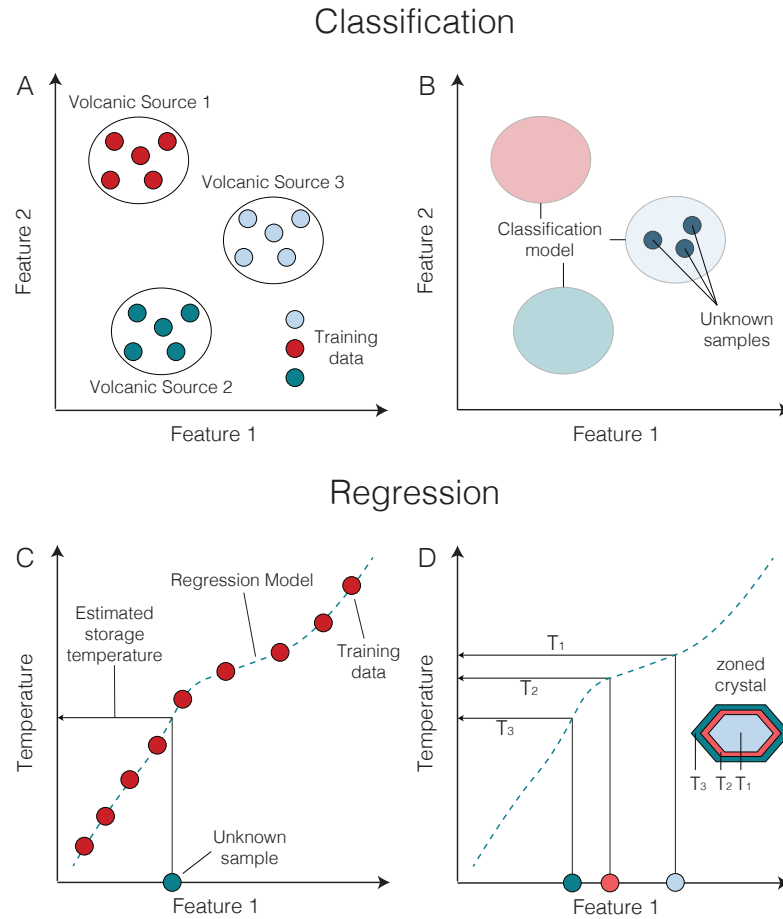


Fig. 1.2 Supervised learning: classification (A, B) and regression (C, D).

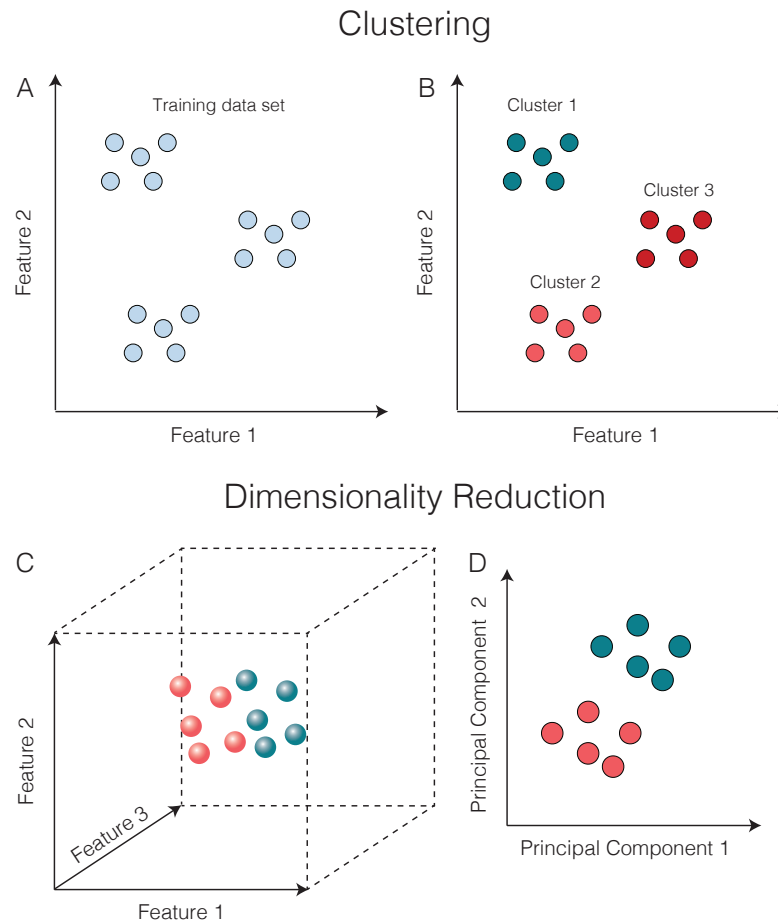


Fig. 1.3 Unsupervised learning: clustering (A, B) and Dimensionality Reduction (C, D).

1.4 Unsupervised Learning

The unsupervised learning process acts with unlabeled training data. Therefore, the ML algorithm tries to exert significant patterns in the investigated data set, without the external feeding of solutions, distinctive of supervised methods. Some application fields of unsupervised learning are clustering, dimensionality reduction, and the detection of outlier or novelty observations.

The clustering process consists of grouping “similar” observations into “homogeneous” groups (i.e., Fig. 1.3 A, B). It helps in discovering unknown patterns in unlabeled data sets. In Earth Sciences, clustering has widespread applications in seismology (e.g., Trugman and Shearer, 2017), remote sensing (e.g., Wang et al.,

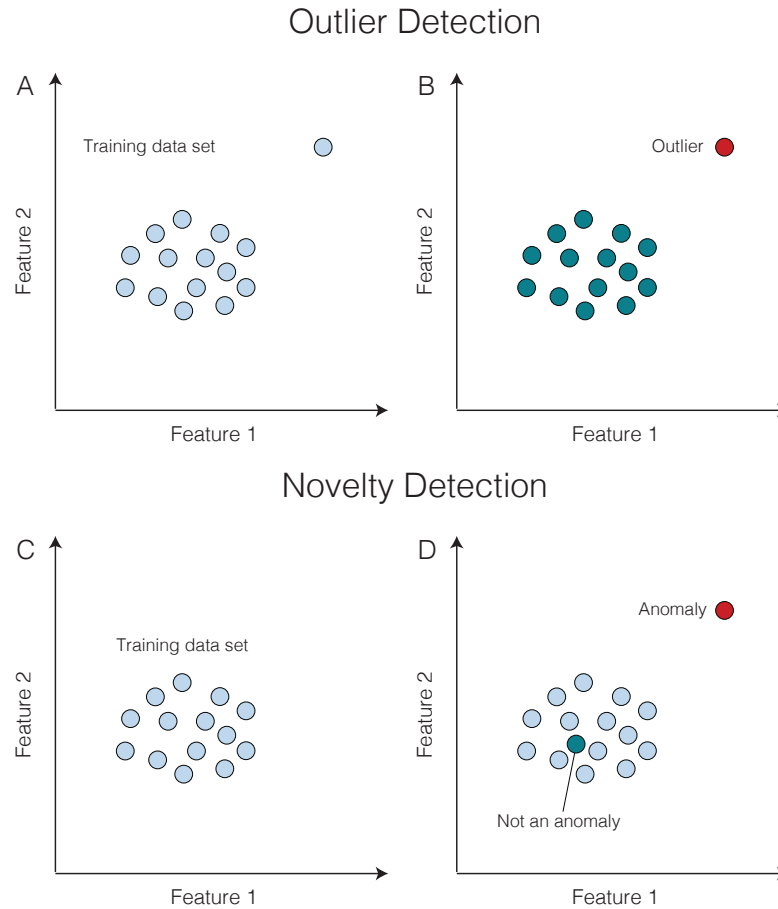


Fig. 1.4 Unsupervised learning: outlier (A, B) and novelty (C, D) detection.

2018), volcanology (e.g., Caricchi et al., 2020), and geochemistry (e.g., Boujibar et al., 2021) to cite a few.

The reduction of the dimensionality (Fig. 1.3 B and C) of a problem reduces the number of features to deal with, allowing the visualization of high-dimensional data sets (e.g., Morrison et al., 2017) or increasing the efficiency of a ML workflow by reducing the time elapsed for the learning process or the complexity of the investigated problem. Tenenbaum et al. (2000) provide a concise but effective definition of dimensionality reduction: “finding meaningful low-dimensionality structures hidden in their high-dimensionality observations.”

Finally, the detection of outlier or novelty observations (Fig. 1.4) deals with deciding whether a new observation belongs to a single set, i.e., an inlier, or should be considered as different, i.e., an outlier or a novelty. The main difference between

outlier or novelty detection relies in the learning process. In outlier detection (Fig. 1.4 A and B), training data contains both inliers and potential outliers. Therefore, the algorithm tries to define which are the observation deviating from the others. In novelty detection (Fig. 1.4 C and D), the training data set contains inliers only. Therefore, the algorithm aims to decide if a new observation is an outlier, i.e., a novelty, or not.

1.5 Semi-Supervised Learning

As you can argue, semi-supervised learning is somehow in between supervised and unsupervised training methods. Typically, semi-supervised algorithms learn by a small portion of labeled data with a large quantity of unlabeled data (Zhu & Goldberg, 2009). In detail, semi-supervised learning algorithms use unlabeled data to improve supervised learning tasks when the labeled data is scarce or expensive (Zhu & Goldberg, 2009). To better understand, please look at Figure 1.5. In detail, Figure 1.5A reports a supervised classification model using two labeled observations as training data set. Also, Figure 1.5 B displays a classification model resulting from a semi-supervised learning using the same two labeled data of Figure 1.5 A, plus several unlabeled observations.

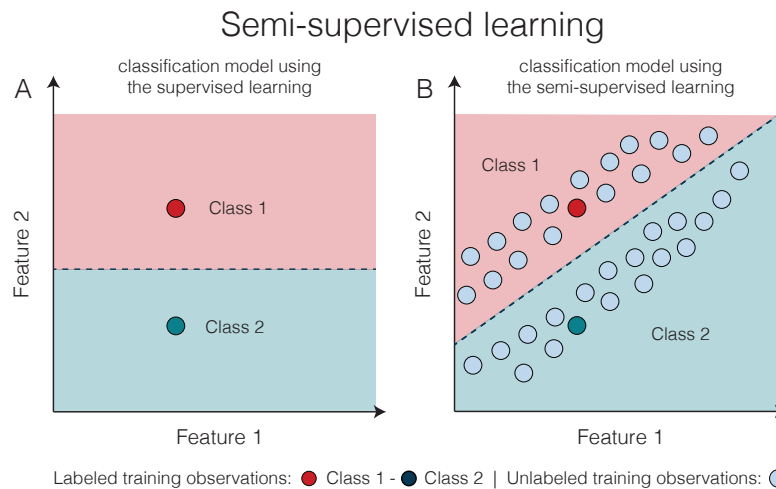


Fig. 1.5 (A) a supervised classification model using two labeled observation as training data set; (B) a semi-supervised classification model using the same two labeled observations in (A) plus many unlabeled instances.

Chapter 2

Setting Up your Python Environments for Machine Learning

2.1 Python Modules for Machine Learning

Python is a widely-used programming language for ML. The development of A ML model in Python bases on both general-purpose scientific libraries (e.g., NumPy, ScyPy, and pandas) and specialized modules (e.g., Scikit-learn¹, PyTorch², and TensorFlow³).

Scikit-learn: Scikit-learn is a Python module to solve ML problems of small- to medium-scale (Pedregosa et al., 2011). It implements a wide range of state-of-the-art machine learning algorithms, making it one of the best options to start learning ML (Pedregosa et al., 2011).

PyTorch: PyTorch is a Python package that combines high-level features for tensor management, neural network development, autograd computation and back-propagation (Paszke et al., 2019). The PyTorch library grows up within Meta's AI⁴ (formerly Facebook AI) research team. Also, it has a strong ecosystem and a large user community support its development (Papa, 2021).

TensorFlow: TensorFlow begins at Google it has been open sourced in 2015. It combines tools, libraries, and community resources to develop and deploy deep learning models in Python (Bharath & Reza Bosagh, 2018).

2.2 A Local Python Environment for Machine Learning

The Individual Edition of the Anaconda Python Distribution⁵ provides an example of a “ready-to-use” scientific Python environment to perform basic ML tasks by the

¹ <https://scikit-learn.org>

² <https://pytorch.org>

³ <https://www.tensorflow.org>

⁴ <https://ai.facebook.com>

⁵ <https://www.anaconda.com>

scikit-learn module. Also, it allows advanced tasks like installing libraries that are specifically developed for Deep Learning, e.g., PyTorch and TensorFlow. To install the Individual Edition of the Anaconda Python distribution, I suggest following the directives given in the official documentation⁶.

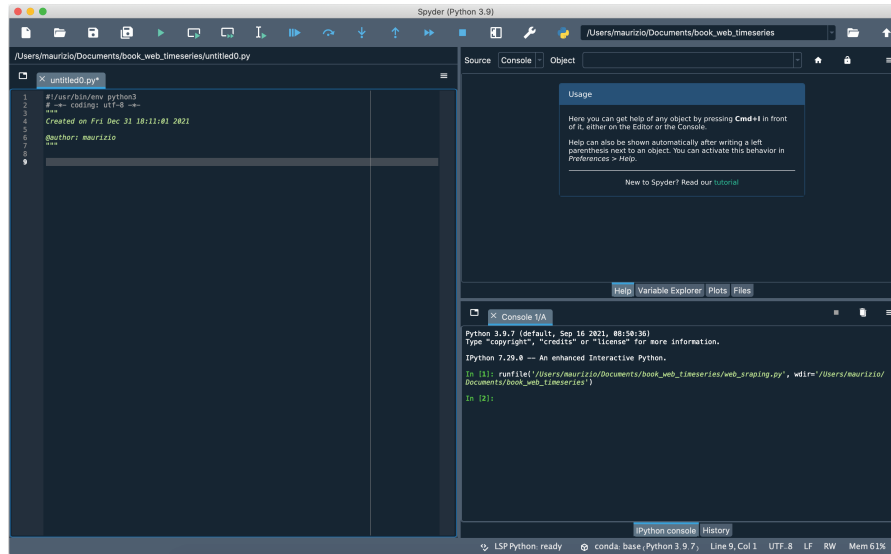


Fig. 2.1 Screenshot of Spyder IDE. The text editor for writing code is on the left. The bottom-right panel is the IPython interactive console, and the top-right panel is the Variable Explorer.

First, download and run the most recent stable installer for your Operating System (i.e., Windows OS, Mac OS, or Linux). For Windows and Mac OS, a graphical installer is also available. The installation procedure using the graphical installer is the same as for any other software application. The Anaconda installer automatically installs the Python core and Anaconda Navigator, plus about 250 packages defining a complete environment for scientific visualization, analysis, and modeling. Over 7500 additional packages, including PyTorch and TensorFlow, can be installed individually, as the need arises, from the Anaconda repository with the “conda”⁷ package management system. The basic tools to start learning and developing small-to medium-scale ML projects are the same as those used for any scientific Python project. As a consequence, I suggest using Spyder and JupyterLab.

Spyder⁸ is an Integrated Development Environment (IDE), combining a text editor to write code, inspection tools for debugging, and interactive Python consoles for code execution (Figure 2.1).

⁶ <https://www.anaconda.com/products/individual/>

⁷ <https://docs.conda.io/>

⁸ <https://www.spyder-ide.org>

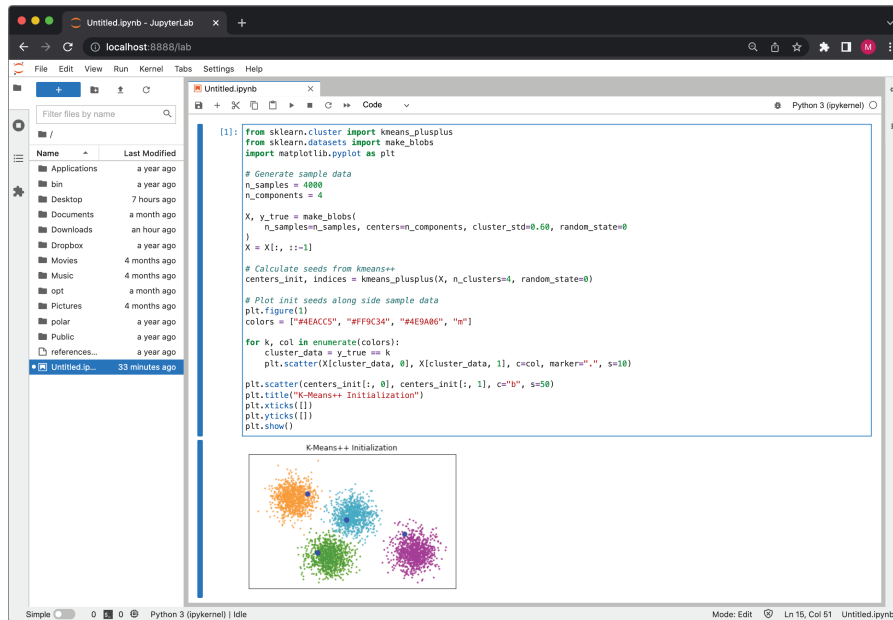


Fig. 2.2 Screenshot of Jupyter Notebook combining narrative text, code, and visualizations.

JupyterLab⁹ is a web-based development environment to manage Jupyter Notebooks, web applications for creating and sharing computational documents (Figure 2.2)

2.3 ML Python Environments on Remote Linux Machines

Being able to access and work on remote computational infrastructures is mandatory for large-scale and data intensive ML workflows. It is far beyond the scope of the present book to provide a detailed description on how to develop High Performance Computational (HPC) infrastructures. However, they are often constituted of a cluster of Linux instances, i.e., virtual computing environments based on the Linux operating system.

Therefore, the instructions on how to connect to and work with a remote Linux instance deserve a description. In the present section, I am going to show you how to setup a Debian instance on the Amazon Web ServicesTM (AWS) facilities. Then, I will show how to setup the Anaconda Individual Edition python environment on your AWS Debian instance.

⁹ <https://jupyter.org>

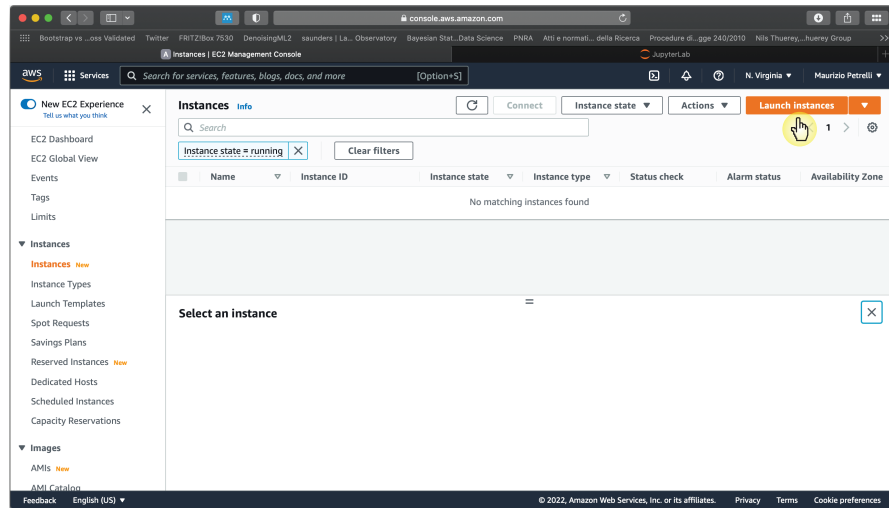


Fig. 2.3 Screenshot of the EC2 management console. the “Launch instance” button allows to start a new instance.

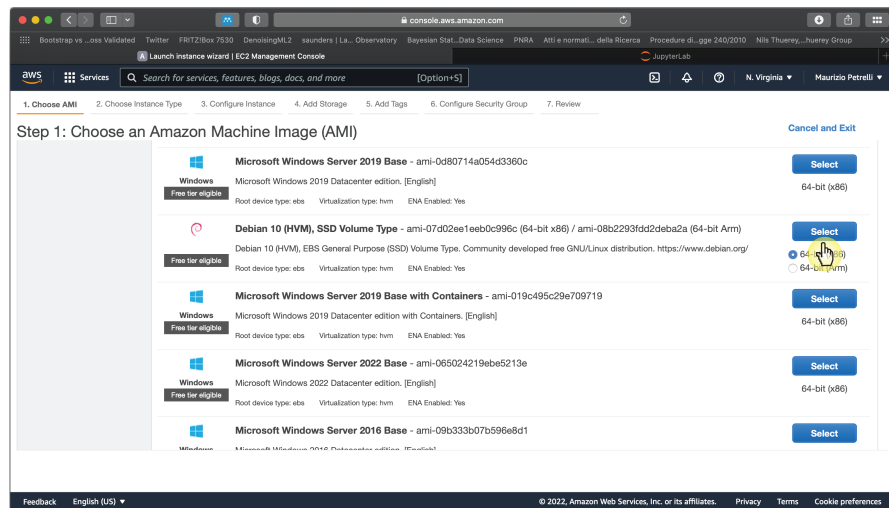


Fig. 2.4 The first step consists of selecting the Amazon Machine Image (AMI).

Figure 2.3 shows the Amazon management console of the “Elastic Compute Cloud” (EC2)¹⁰. From the EC2 management console, a new computational instance can be launched by clicking the “Launch new instance” button. A guided step-by-step procedure will follow, allowing you the definition of each detail (i.e., 1 chose the Amazon Machine Image , 2 choose instance type, 3 configure instance, 4

¹⁰ <https://aws.amazon.com/ec2/>

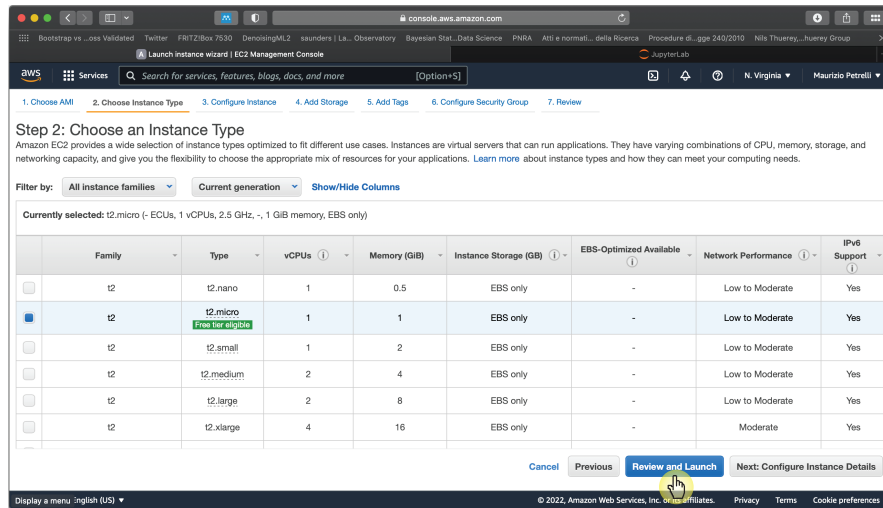


Fig. 2.5 The second step consists of selecting the “Instance Type”.

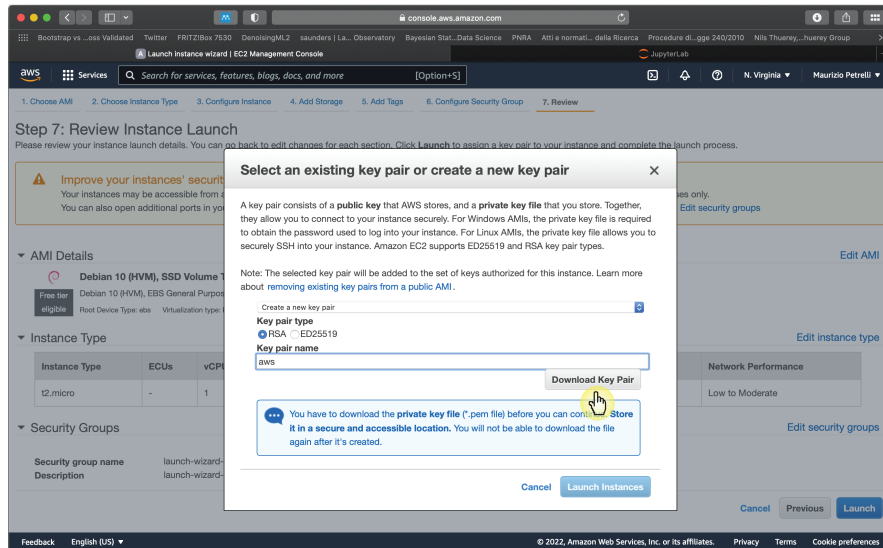


Fig. 2.6 Before launching a new instance you must select a “key pairs”.

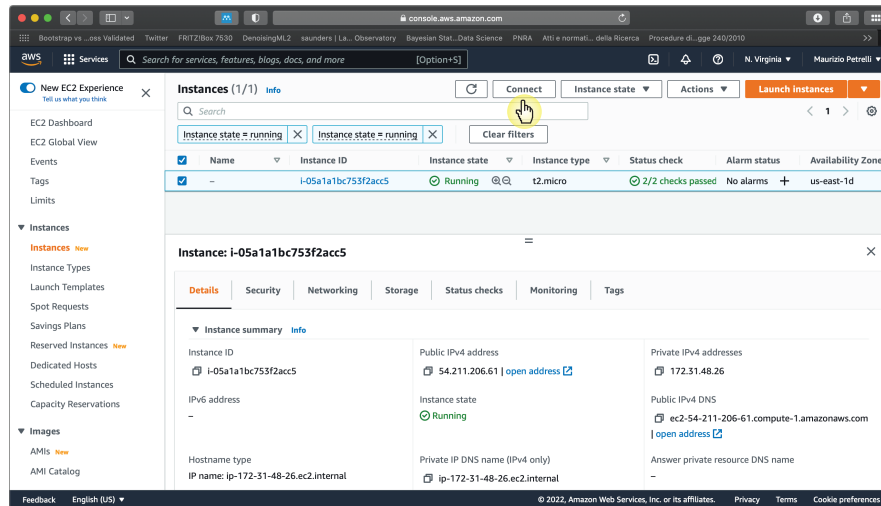


Fig. 2.7 Connecting to an instance.

add storage, 5 add Tags, 6 configure security group, 7 review and launch). At the first step, i.e., choose an Amazon Machine Image (AMI; Figure 2.4), we select the Debian 10 64-bit (x86) AMI. As instance type, we select the t2.micro that is eligible as “Free Tier”. To note, very large instance types could be selected. As an example, the g5.48xlarge instance type shares 192 virtual CPU, 768 GiB of memory, and a network performance of 100 Gigabit. The amount of allowed power is only a matter of the budget at your disposal. We can safely set all the other instance parameters, i.e., steps from 3 to 6, to the default values and click on the “Review and Launch button”. The last task consists of selecting an existing key pair or creating a new one. A “key pair” defines the security credentials to prove your identity when connecting to a remote instance. It consists of a “public key”, stored in the remote instance, and a “private key”, hosted in your machine. Anyone who possesses the “private key” of a specific “key pair” can connect to the instance that stores the associated public key. From your Linux and Unix OS (including Mac OS), you can create a “key pair” using the `ssh-keygen` command. However, the EC2 management console allows you to create and manage “key pairs with” a single click (Figure 2.6).

The last step consists of launching the instance that, after the initialization, will appear in the EC2 management console (Figure 2.7). To access an instance, select it in the EC2 management console and click on the “Connect” button (Figure 2.7). It will open the “Connect to instance” window, showing all the available options to access the instance (Figure 2.8). Our choice is to access by using the Secure Shell (SSH) protocol (Figure 2.8). The SSH Protocol is a cryptographic communication system for secure remote login and network services over an insecure network. It allows you to “safely” connect and work on a remote instance from your desk or sofa.

To connect to the remote instance, we need a ssh client (e.g., the Mac OS Terminal or PuTTY¹¹) and digit the following command:

```
ssh -i local_path/aws.pem user@user_name@host
```

where *ssh* command initializes the ssh connection with the *user* account to the *host* (i.e., an IP or a domain name) remote instance. the *-i* option selects a specific private key, i.e., *aws.pem* to pair the public key in the *host* instance.

For the specific case reported in Figure 2.8, I digit:

```
ssh -i /Users/maurizio/.ssh/aws.pem admin@ec2-52-91-26-146.compute-1.amazonaws.com
```

I am now connected to the remote instance in one AWS computing facility (Figure 2.9) and I am ready to install the Anaconda Python Individual Edition from the command line.

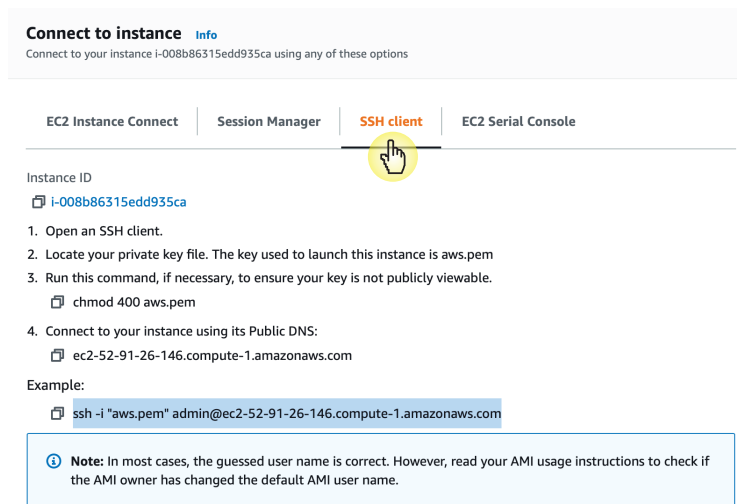


Fig. 2.8 Accessing by a SSH client.

Before starting the install procedure for the Anaconda Python Individual edition, I suggest upgrading Debian packages:

```
$ sudo apt-get update
$ sudo apt-get dist-upgrade
```

The *sudo apt-get update* command gets you an updated list of packages. Then the *sudo apt-get dist-upgrade* will ‘intelligently’ upgrade these packages, without upgrading

¹¹ <https://www.putty.org>

```

maurizio — admin@ip-172-31-52-75: ~ — ssh -i ~/.ssh/aws.pem admin@ec2-52-91-26-146.compute-1.amazonaws.com — 127x33
Last login: Tue Jan  4 11:34:27 on ttys000
(base) maurizio@Maurizios-MacBook-Pro ~ % ssh -i ~/.ssh/aws.pem admin@ec2-52-91-26-146.compute-1.amazonaws.com
Linux ip-172-31-52-75 4.19.0-14-cloud-amd64 #1 SMP Debian 4.19.171-2 (2021-01-30) x86_64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Jan  4 10:34:44 2022 from 79.19.187.94
admin@ip-172-31-52-75:~$

```

Fig. 2.9 Well done! You are connected to your remote instance.

the current Debian release. Now download the last Anaconda Python distribution¹² for Linux-x86_64 using *curl*:

```
$ curl -O https://repo.anaconda.com/archive/Anaconda3-2021.11-Linux-x86_64.sh
```

if *curl* does not work, install it:

```
$ sudo apt-get install curl
```

At this point, we need verifying the data integrity of the installer with cryptographic hash verification through the SHA-256 checksum. We'll use the *sha256sum* command along with the filename of the script:

```
$ sha256sum Anaconda3-2021.11-Linux-x86_64.sh
```

The result, i.e.,

```
fedf9e340039557f7b5e8a8a86affa9d299f5e9820144bd7b92ae9f7ee08ac60
Anaconda3-2021.11-Linux-x86_64.sh
```

must match the cryptographic hash verification code reported in the Anaconda repository¹³ As final step, we can run the installation script:

```
$ bash Anaconda3-2021.11-Linux-x86_64.sh
```

¹² <https://repo.anaconda.com/archive/>

¹³ <https://docs.anaconda.com/anaconda/install/hashes/lin-3-64/>

It will start a step-by-step guided procedure starting from:

```
Welcome to Anaconda3 2021.11
In order to continue the installation process, please review the
  license
agreement.
Please, press ENTER to continue
```

Press 'ENTER' to proceed to access the license information and go ahead clicking 'ENTER' until you get the following question:

```
Do you approve the license terms? [yes|no]
```

Type 'yes' to get the next step, i.e., the selection of the location for the installation:

```
Anaconda3 will now be installed into this location:
/home/admin/anaconda3
- Press ENTER to confirm the location
- Press CTRL-C to abort the installation
- Or specify a different location below
```

I suggest pressing 'ENTER' to keep the default location. At the end of the installation, you will receive the following output:

```
...
installation finished.
Do you wish the installer to initialize Anaconda3
by running conda init? [yes|no]
[no] >>>
```

Type 'yes', and digit:

```
$ source ~/anaconda3/bin/activate
```

or restart the shell to activate the base conda environment, highlighted by (base) at the beginning of the prompt command, e.g.:

```
(base) admin@ip-172-31-59-186:~$
```

Now, the base environment for ML in Python is ready in your remote instance.

2.4 Working with your Remote Instance

Once connected with your remote instance, i.e., by:

```
$ ssh -i local_path/aws.pem user@user_name@host
```

the knowledge of the most basic command of a Linux OS is mandatory. However, a detailed explanation of the architecture, commands, and operations of the Linux OS

is far away from the scope of the present book. As a consequence, I suggest reading specialized books (i.e., Negus, 2015; Ward, 2021) to acquire specific skills. Table 2.1 reports the most common command allowing you the file transfer between your local machine and the remote instance. Also, it shows basic tools for file management in a Linux environment.

Table 2.1 Basic Linux commands

Command	Description
ls	view the contents of a directory.
cd..	move one directory up.
cd folder_name	go to the folder named folder_name
cp myfile.jpg /new_folder	copy myfile.jpg to the new_folder path:
mv	use mv to move files, the syntax is similar to cp
mkdir my_folder	create a new folder named my_folder
rm	delete directories and the contents within them. Take care with rm!
tar	archive multiple files into a compressed file
chmod	allow you to change the read, write, and execute permissions of files and directories.
top	it will display a list of running processes, cpu and memory usage.
pwd	print the current working directory, i.e., the one where you are in
sudo	It is the abbreviation of “SuperUser Do”. It enables you to run tasks requiring administrative permissions. Take great care with sudo!

To copy a file from your local machine to the remote instance and *vice versa* I suggest using the scp command, based on the SSH protocol. In detail:

```
$ scp -i local_path/aws.pem filename user@host:/home/user/
filename
```

will copy the file named ‘filename’ from the local machine to the folder */home/user/* of remote instance *host*. As we know (i.e., section 2.3) the *aws.pem* private key stores the credentials to securely login to the *host instance*. To copy a file from your remote instance to the local machine use:

```
$ scp -i local_path/aws.pem user@host:/home/user/filename /
localfolder/filename
```

finally, to launch a python script use the *python* command:

```
$ python myfile.py
```

to run multiple python files you could use a bash script, i.e., a text file named *my_bash_script.sh*, then run it:

```
$ bash my_bash_script.sh
```

Here are two examples:

```
#!/bin/bash
/home/path_to_script/script1.py
/home/path_to_script/script2.py
/home/path_to_script/script3.py
/home/path_to_script/script4.py
```

and

```
#!/bin/bash
/home/path_to_script/script1.py &
/home/path_to_script/script2.py &
/home/path_to_script/script3.py &
/home/path_to_script/script4.py &
```

to run them sequentially and in parallel, respectively.

Note that, the Anaconda Individual Edition comes with scikit-learn as default package. Deep learning packages like Tensorflow and PyTorch must be installed separately. To avoid conflicts, I suggest creating isolated Python environments to work with PyTorch and TensorFlow, respectively.

2.5 Preparing Isolated Deep Learning Environments

Conda is an open-source package management system and environment management developed by Anaconda¹⁴. It allows to install and update Python packages and dependencies. Also, it allows managing isolated Python environments to avoid conflicts. As an example, the following statement:

```
conda create --name env_ml python=3.9 spyder scikit-learn
```

will create a new Python 3.9 environment named `env_ml` with `spyder`, `scikit-learn`, and related dependencies installed. to activate the environment:

```
conda activate env_ml
```

to deactivate the current environment:

```
conda deactivate
```

to list the available environments:

```
conda info --envs
```

¹⁴ <https://www.anaconda.com/>

in the resulting list, the active environment is highlighted by *. Also, the active environment is typically reported at the beginning of the terminal prompt, e.g., (base):

```
(base) admin@ip-172-31-59-186:~$
```

to remove an environment:

```
conda remove --name env_ml --all
```

the statement:

```
conda env export > env_ml.yml
```

will export all the information about the active environment to a file named env_ml.yml. Using env_ml.yml, the environment can be easily shared with and installed by others using the following command:

```
conda env create -f env_ml.yml
```

you will find more details on environment management within conda official documentation¹⁵. The following listing resume all the steps to create a ML environment with deep learning functionalities based on PyTorch:

```
$ conda create --name env_pt python=3.9 spyder scikit-learn
$ conda activate env_pt
(env_pt)$ conda install pytorch torchvision torchaudio -c pytorch
```

the last command will install PyTorch, working on the cpu only, on my mac. To find the right command for your hardware and operating system, please refer to the PyTorch website¹⁶. Similarly, to create a ML environment based on scikit-learn with Tensorflow deep learning functionalities, type the following command:

```
$ conda create --name env_tf --channel=conda-forge tensorflow
```

As you can see, I utilized a specific channel, i.e., conda-forge¹⁷, to download tensorflow and spyder. Now listing my conda environments I get:

```
$ conda info --envs
Output:
# conda environments:
#
base                * /opt/anaconda3
env_ml              /opt/anaconda3/envs/env_ml
env_pt              /opt/anaconda3/envs/env_pt
env_tf              /opt/anaconda3/envs/env_tf
```

¹⁵ <https://docs.conda.io/>

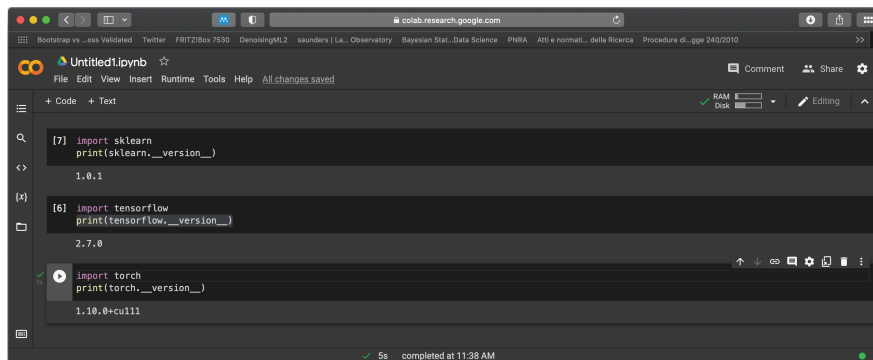
¹⁶ <https://pytorch.org/get-started/locally/>

¹⁷ <https://conda-forge.org>

2.6 Cloud Based Machine Learning Environments

With Cloud based ML environments, I refer to Jupyter Notebooks based services that are hosted in the Cloud. Examples are GoogleTM Colaboratory, Kaggle, AWS Sagemaker, and Saturn Cloud. The first two services, i.e., GoogleTM Colaboratory and Kaggle are both managed by GoogleTM and offer a free plan with limited computational resources. The AWSTM Sagemaker, only offers a limited free trial. Finally, Saturn Cloud offers a free plan with 30 hours of computations. They all allow the online use of Jupyter Notebooks. Figures 2.10, 2.11, and 2.12 report a quick look at the entry level notebooks GoogleTM Colaboratory, Kaggle, and AWSTM Sagemaker, respectively. Also, Figures 2.10 and 2.11 highlight that both GoogleTM Colaboratory and Kaggle come with all scikit-learn, Tensorflow, and PyTorch already installed and ready to use. However, GoogleTM Colaboratory looks updated to more recent versions. Finally, the most basic Sagemaker image, only comes with a non recent, i.e., 0.22.1, scikit-learn version. Using Saturn CloudTM, a new Jupyter instance can be launched by clicking the “New Jupyter” Server button (Fig. 2.13). It will open a new window ((Fig. 2.14) where you can personalize the instance. To note, the default configuration does not include neither PyTorch nor Tensorflow. However, they can be added quickly in the Extra Packages section (Fig. 2.14). As an example, Fig. 2.14 shows how to add PyTorch. Finally Fig. 2.15 demonstrated that the resulting environments come with the latest released of both Scikit-learn and PyTorch.

Although all the reported cloud based ML Jupyter environments are robust and flexible solutions, I suggest the use of GoogleTM Colaboratory or Saturn CloudTM as first choices for a novice.



The screenshot shows the Google Colaboratory web interface. The browser address bar displays 'colab.research.google.com'. The notebook title is 'Untitled1.ipynb'. The code editor shows three cells of code:

```
[7] import sklearn
print(sklearn.__version__)
1.0.1

[6] import tensorflow
print(tensorflow.__version__)
2.7.0

import torch
print(torch.__version__)
1.10.0+cu111
```

The status bar at the bottom indicates '5s completed at 11:38 AM'.

Fig. 2.10 GoogleTM Colaboratory.

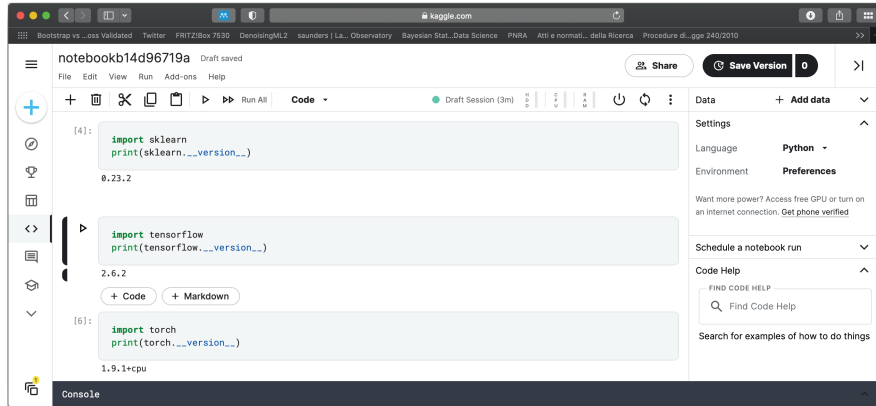
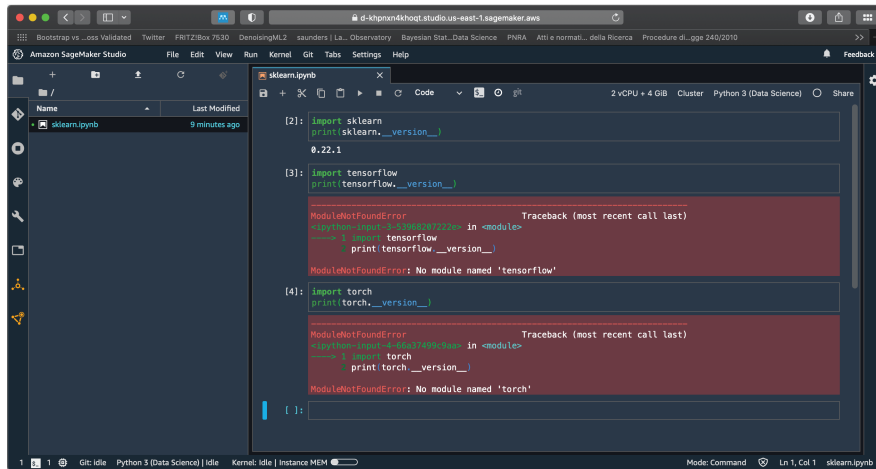


Fig. 2.11 Kaggle.

Fig. 2.12 AWSTM Sagemaker.

2.7 Speed Up your ML Python Environment

It is a common argument by Python detractors that Python is slow when compared with other established programming languages, e.g., C or FORTRAN. We all agree with this statement but, in my opinion, this is not the point. In scientific computations, Python relies on libraries developed in more performing languages, mainly in C/C++, and parallel computing platforms, e.g. CUDA¹⁸. As an example NumPy, the core Python library for scientific computing bases on “a well-optimized C code¹⁹. For ML purposes, all scikit-learn, PyTorch and Tensorflow delivers a base version of

¹⁸ <https://developer.nvidia.com/cuda-zone>

¹⁹ <https://numpy.org>

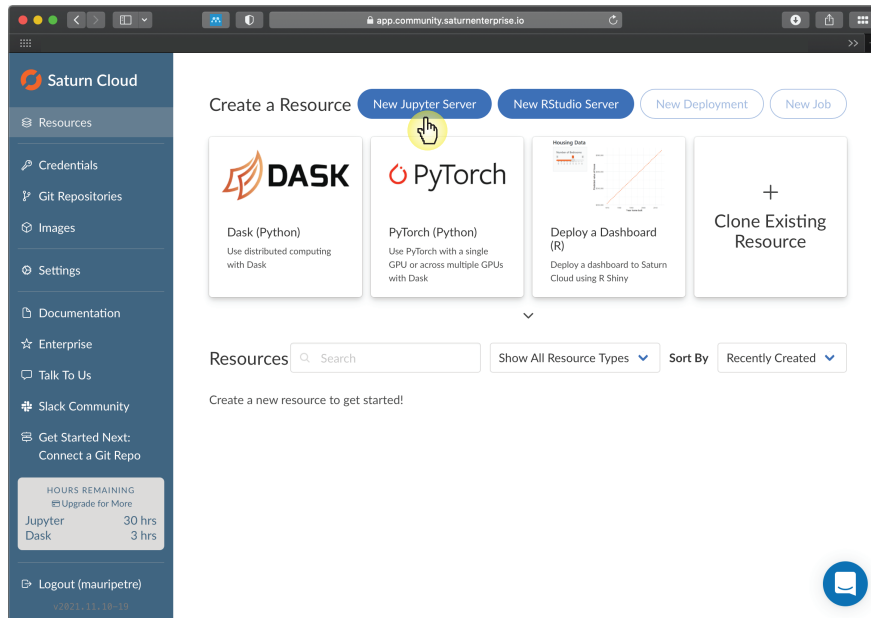


Fig. 2.13 Saturn CloudTM.

the library that can be safely installed in any local machine for rapid prototyping and little- to mid scale-problems. Also, optimized versions for computing intensive applications are also available. As an example, the IntelTM Extension for Scikit-learn accelerates ML applications in Python for Intel-based hardware of a factor 10-100X²⁰. To install the IntelTM Extension for Scikit-learn can be easily installed using `conda`. To prevent conflicts, I strongly recommend to create a new conda environment, e.g., `env_ml_intel`:

```
$ conda create -n env_ml_intel -c conda-forge python=3.9 scikit-learn-intelex scikit-learn rasterio matplotlib pandas spyder scikit-image seaborn
```

Now, listing my local environments, I get:

```
$ conda info --envs
Output:
# conda environments:
#
base                * /opt/anaconda3
env_ml              /opt/anaconda3/envs/env_ml
env_pt              /opt/anaconda3/envs/env_pt
env_tf              /opt/anaconda3/envs/env_tf
env_ml_intel        /opt/anaconda3/envs/env_ml_intel
```

²⁰ <https://github.com/intel/scikit-learn-intelex>

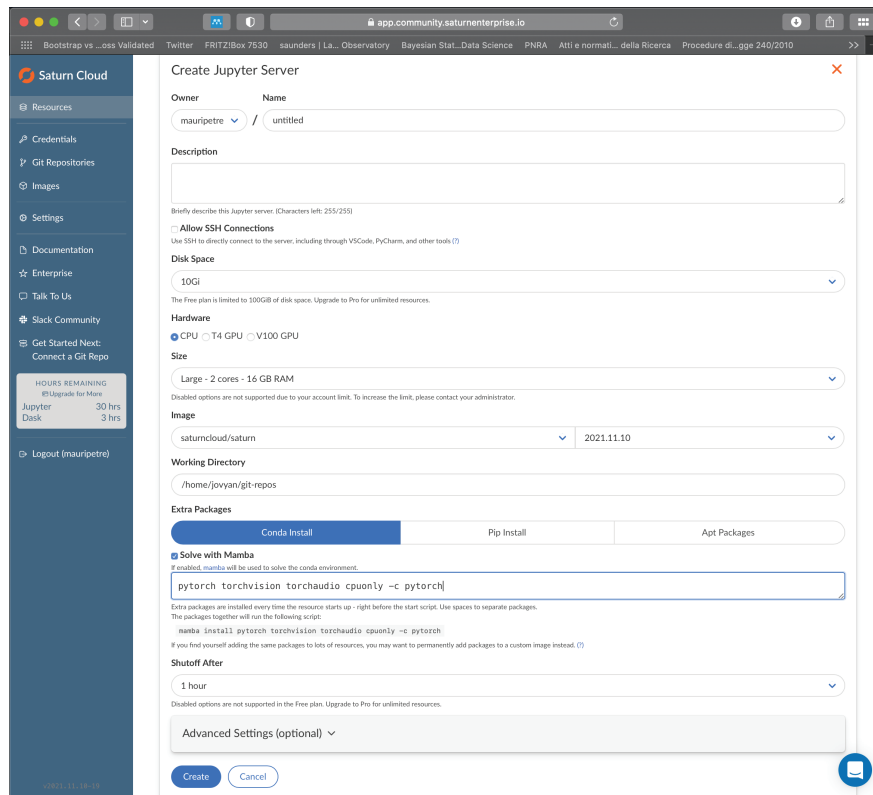


Fig. 2.14 Starting a Jupyter Server, i.e., a machine to run Jupyter Notebooks, in Saturn CloudTM.

In detail, I left the *base* environment untouched. Then I create two general purpose ML environments, i.e., *env_ml* and *env_ml_intel*, with the *larrt* optimized by Intel. Finally, I created two deep learning environments, i.e., *env_pt* and *env_tf*, based on PyTorch and Tensorflow, respectively.

To note, deep learning libraries, e.g., PyTorch and Tensorflow, are highly optimized to support GPU computing, e.g., CUDA²¹ and ROCm²². As an example a Pytorch, CUDA optimized version for Linux OS can be easily installed by *conda*:

```
$ conda install pytorch torchvision torchaudio cudatoolkit=11.3 -c pytorch
```

To provide a complete description on how to perform high performance computing (HPC) ML application in Python is far away from the scope of the present book. Therefore, please refer to the official documentation of each tool to get further details.

²¹ <https://developer.nvidia.com/cuda-zone>

²² <https://rocmdocs.amd.com/en/latest/>

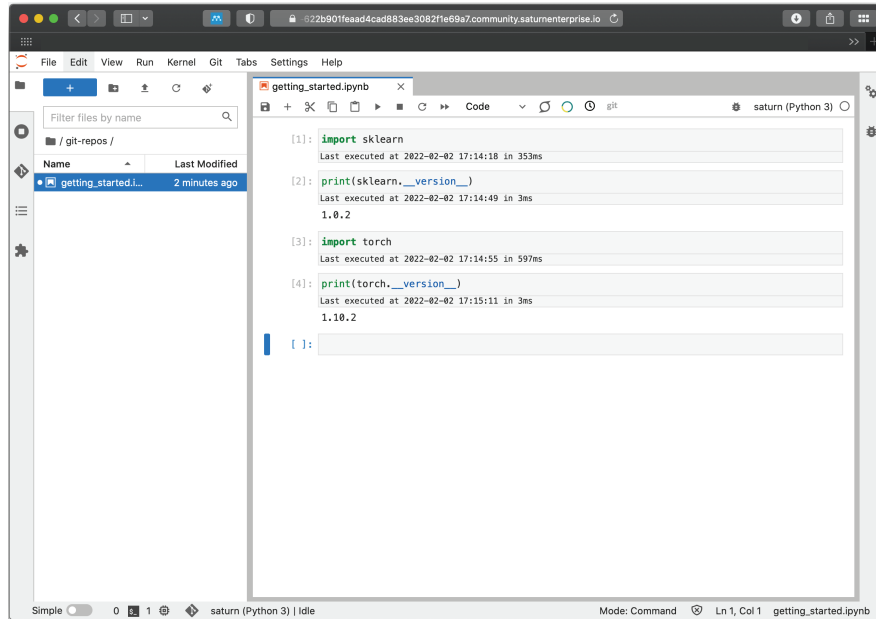


Fig. 2.15 Running a Jupyter Notebook in Saturn CloudTM.

Chapter 3

Machine Learning Workflow

3.1 Machine Learning Step-by-Step

Figure 3.1 reports a generalized workflow that is common to most ML projects. It starts with getting the data. In Earth Sciences, data could come from large scale samplings of geological or geochemical features, remote sensing, well logs data, and petrological experiments, to cite a few. The successive step is pre-processing. It consists of all the operations required to prepare your data set for successive steps, i.e., the training and validation. Training the model is about running ML algorithms, i.e., the core business of a ML model. Then, the validation step aims at checking the goodness of the training and ensuring the generalization capability of your model. Steps 3 and 4 are often closely connected and iterated many times to improve the quality of the results. The last step consists of deploying and securing your model.

Now we are going to evaluate each step providing insights to successfully run a ML model in the field of the Earth Sciences.



Fig. 3.1 Workflow of a ML model

3.2 Get your Data

Your data set repository could assume many different aspects. The easiest ones are tabular data stored in text (e.g., .csv) or ExcelTM files. Sometimes, a Structured Query Language (SQL) database stores your data. Larger data sets could be stored in the Hierarchical Data Format (HDF5)¹, Optimized Row Columnar (ORC)², Feather (i.e., Arrow IPC columnar format)³ or Parquet⁴ Formats, to cite a few.

For data that fits into your Random Access Memory (RAM), pandas is probably the best choice for data import and manipulation (e.g., slicing, filtering, etc) through *DataFrames*. Table 3.1 provide us with a description of the potentials of pandas methods for input and output (I/O).

If the data set starts filling your RAM entirely, probably Dask⁵ is the library of choice to manage your data and scale the Python libraries to parallel environments. In detail, Dask is a library minded to deal with “Big Data” through parallel computing in Python. Dask extend the concept of *DataFrame* to Dask *DataFrames*, i.e., large parallel *DataFrames* composed of many smaller pandas *DataFrames*. We will introduce Dask and parallel computing later in the book (i.e., chapters XX and XX). In the meantime, we start importing our data sets for ML applications in the field of Earth Sciences using pandas (code listing 3.1):

```

1 import pandas as pd
2
3 my_data = pd.read_excel("PGD_SiC_2021-01-10.xlsx", sheet_name='
4     PGD-SIC')
5 print(my_data.info(memory_usage="deep"))
6
7 Output:
8 <class 'pandas.core.frame.DataFrame'>
9 RangeIndex: 19978 entries, 0 to 19977
10 Columns: 123 entries, PGD ID to err[d(138Ba/136Ba)]
11 dtypes: float64(112), object(11)
12 memory usage: 29.4 MB
13 '''

```

Listing 3.1 Importing an Excel data set in Python.

I assume that you are already familiar with the *read_excel* statement in pandas. If it is not the case, I strongly suggest you to start with an introductory book like “Introduction to Python in Earth Science Data Analysis” (Petrelli, 2021). The statement at line 4 of code listing 3.1 tells you how much memory your data set

¹ <https://www.hdfgroup.org/solutions/hdf5/>

² <https://orc.apache.org>

³ <https://arrow.apache.org/docs/python/feather.html>

⁴ <https://parquet.apache.org>

⁵ <https://dask.org>

uses. In our case, the imported data set, consisting of ~ 20000 rows and 123 columns, consumes 24.4 MB, still far below the 32GB of my MacBookTM pro.

Table 3.1 Pandas methods to import standard and *state-of-the-art* file formats for Machine Learning applications

Method	Description	comment
<code>read_table()</code>	Read general delimited file	slow, not for large data sets
<code>read_csv()</code>	Read comma-separated values (csv) files	slow, not for large data sets
<code>read_excel()</code>	Read Excel files	slow, not for large data sets
<code>read_sql()</code>	Read sql files	slow, not for large data sets
<code>read_pickle()</code>	Read pickled objects	fast, not for large data sets
<code>read_hdf()</code>	Read Hierarchical Data Format (HDF) files	fast, good for large data sets
<code>read_feather()</code>	Read feather files	fast, good for large data sets
<code>read_parquet()</code>	Read parquet files	fast, good for large data sets
<code>read_orc()</code>	Read Optimized Row Columnar files	fast, good for large data sets

Large data sets, i.e., approaching or exceeding tera (10^{12}) or peta (10^{15}) bytes cannot be efficiently stored in text files (e.g., csv files) or in Excel. Standard relational databases (e.g., PostgreSQL, MySQL, and MS-SQL) can store information at large scale, but they are not efficient (i.e., fast enough) if compared with the *state-of-the-art* high performance data software libraries and file formats to manage, process, and store huge amounts of data. To note, the formal definition proposed by De Mauro et al. (2016) involves all the concepts of volume, velocity, and variety: “Big Data is the Information asset characterised by such a High Volume, Velocity and Variety to require specific Technology and Analytical Methods for its transformation into Value”. A detailed description of data storage and analysis frameworks for Big-Data is far away from the aims of the present book and I refer to specific texts for those interested (Panda et al., 2022; Pietsch, 2021). Here I limit to compare the performances of pandas in writing and reading .csv and hdf files at GB scale on my MacBook pro (2.3 GHz Quad-Core Intel Core i7, 32GB RAM). As an example, code listing 3.2 generate a pandas *DataFrame* of ~ 10 GB named **my_data**, composed of random numbers hosted in 26 columns and $5 \cdot 10^7$ rows.

I utilized the `my_data.info(memory_usage = “deep”)` (code listing 3.3 to check the real memory usage of **my_data**, i.e., 9.7 GB.

Code listing 3.4 shows the execution time to write (In [1], In [2], and In [3]) and read (In [4], In [5], and In [6]) a text file (.csv), a parquet, and hdf5, respectively. Results show that saving a .csv file require about 25 minues, i.e., a quite long time! On the contrary, saving parquet and hdf5 files need from 7 to 12 seconds, respectively. Reading times are of the same order of magnitude, i.e., about 5 minutes for the .csv and 30 seconds for both parquet and hdf5 files.

```

1 import pandas as pd
2 import numpy as np
3 import string
4
5 my_data = pd.DataFrame(np.random.normal(size=(50000000, 26)),
6                       columns=list(string.ascii_lowercase))

```

Listing 3.2 Generating a mid-size, data set of about 10 GB

```

1 In [1]: my_data.info(memory_usage="deep")
2 <class 'pandas.core.frame.DataFrame'>
3 RangeIndex: 50000000 entries, 0 to 49999999
4 Data columns (total 26 columns):
5 #   Column  Dtype
6 ---  -
7 0    a      float64
8 1    b      float64
9 2    c      float64
10 3    d      float64
11 4    e      float64
12 5    f      float64
13 6    g      float64
14 7    h      float64
15 8    i      float64
16 9    j      float64
17 10   k      float64
18 11   l      float64
19 12   m      float64
20 13   n      float64
21 14   o      float64
22 15   p      float64
23 16   q      float64
24 17   r      float64
25 18   s      float64
26 19   t      float64
27 20   u      float64
28 21   v      float64
29 22   w      float64
30 23   x      float64
31 24   y      float64
32 25   z      float64
33 dtypes: float64(26)
34 memory usage: 9.7 GB

```

Listing 3.3 Checking the memory usage of our *DataFrame*

In light of the evidence reported in Code listing 3.4, I strongly suggest to discontinue the use of text files to store and retrieve your data in favor of binary files like hdf5 or parquet. This is particularly true when the dimension of the data set starts increasing.

```
1 In [1]: %time my_data.to_csv('out.csv')
2 CPU times: user 22min 48s, sys: 55.8 s, total: 23min 44s
3 Wall time: 24min 16s
4
5 In [2]: %time my_data.to_parquet('out.parquet')
6 CPU times: user 13.1 s, sys: 2.71 s, total: 15.8 s
7 Wall time: 11.8 s
8
9 In [3]: %time my_data.to_hdf('out.h5', key="my_data", mode="w")
10 %time my_data.to_hdf('out.h5', key="my_data1", mode="w")
11 CPU times: user 39.2 ms, sys: 4.33 s, total: 4.37 s
12 Wall time: 6.59 s
13
14 In [4]: %time my_data_1 = pd.read_csv('out.csv')
15 CPU times: user 3min 28s, sys: 37.7 s, total: 4min 5s
16 Wall time: 4min 45s
17
18 In [5]: %time my_data1 = pd.read_parquet('out.parquet')
19 CPU times: user 12.7 s, sys: 26.3 s, total: 39 s
20 Wall time: 31 s
21
22 In [6]: %time my_data1 = pd.read_hdf('out.h5', key='my_data')
23 CPU times: user 10.2 s, sys: 12.7 s, total: 23 s
24 Wall time: 28.8 s
```

Listing 3.4 Generating a mid-size, data set of about

3.3 Data Pre-Processing

Pre-processing consists of all the operations required to prepare your data set for successive steps, i.e., the training and validation (Maharana et al., 2022). It is a crucial step since it makes raw data suitable to build your ML model. Probably, during the development of a ML project, you will spend most of your time making your data ready for the training. In detail, pre-processing refers to preparing (e.g., cleaning, organizing, normalizing) the raw data before moving to the training. Also, it includes the preliminary steps to allow validation (e.g., train-test splitting).

Data Inspection

Data inspection allows familiarizing with your data set. It consists of a qualitative investigation of your data. A fundamental task of data inspection is descriptive statistics. It provides you with a clear idea about the “shape” and the structure of your data set. How does it could help you. As an example, looking at the histogram distributions, you could start evaluating if the methods that assume a Gaussian distribution will fit your data.

Code listing 3.5 shows how to perform a preliminary determination of the main descriptive indexes of location (e.g., the mean and the median, i.e., p_{50} or 50% percentile) and dispersion [e.g., the standard deviation, the range, i.e., (range = max - min), or the interquartile range, i.e., ($iqr = p_{75} - p_{25}$)]

```

1 In [1]: sub_data = my_data[['12C/13C', '14N/15N']]
2
3 In [2]: sub_data.describe().applymap("{0:.0f}".format)
4
5 Out[2]:
6      12C/13C  14N/15N
7 count    19581    2544
8 mean         66    1496
9 std         207    1901
10 min          1         4
11 25%          44    336
12 50%          55    833
13 75%          69   2006
14 max       21400   19023

```

Listing 3.5 Performing descriptive statistics in Python

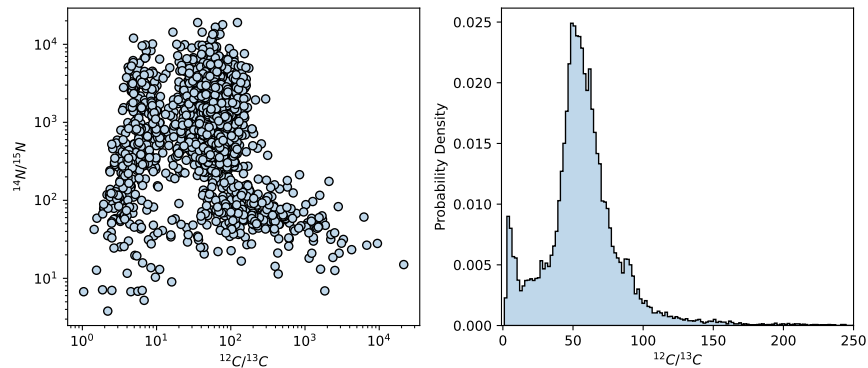


Fig. 3.2 Descriptive Statistics

Figure 3.2 and code listing 3.6 show how to perform a basic statistical visualization in Python. As an example, Fig. 3.2 shows the distribution of data in the $^{14}\text{N}/^{15}\text{N}$ Vs. $^{12}\text{C}/^{13}\text{C}$ projection and the histogram distribution of $^{12}\text{C}/^{13}\text{C}$ on the left and right panels, respectively.

```

1 import matplotlib.pyplot as plt
2
3 fig = plt.figure(figsize=(9,4))

```



```

4 ax1 = fig.add_subplot(1,2,1)
5 ax1.plot(my_data['12C/13C'], my_data['14N/15N'],
6         marker='o', markeredgcolor='k',
7         markerfacecolor='#BFD7EA', linestyle='',
8         color='#7d7d7d',
9         markersize=6)
10 ax1.set_yscale('log')
11 ax1.set_xscale('log')
12 ax1.set_xlabel(r'$^{12}C/^{13}C$')
13 ax1.set_ylabel(r'$^{14}N/^{15}N$')
14
15 ax2 = fig.add_subplot(1,2,2)
16 ax2.hist(my_data['12C/13C'], density=True, bins='auto',
17         histtype='stepfilled', color='#BFD7EA', edgecolor='
18         black',)
19 ax2.set_xlim(-1,250)
20 ax2.set_xlabel(r'$^{12}C/^{13}C$')
21 ax2.set_ylabel('Probability Density')
22 fig.set_tight_layout(True)

```

Listing 3.6 Performing descriptive statistics in Python

Data Cleaning and Imputation

In real-world data sets, like geological ones, the occurrence of “unwanted” entries is ubiquitous (Zhang, 2016). Examples are void (i.e., missing data), ‘Not a Number’ (NaN) entries or large outliers. The cleaning of your data set mainly consists of removing “unwanted” entries. As an example the methods `.dropna()` and `.fillna()` will help you when working with missing data, imported by pandas as NaN (code listing 3.7):

```

1 import pandas as pd
2
3 cleaned_data = my_data.dropna(
4     subset=['d(135Ba/136Ba)', 'd(138Ba/136Ba)'])
5
6 print("Before cleaning: {} cols".format(my_data.shape[0]))
7 print("After cleaning: {} cols".format(cleaned_data.shape[0]))
8
9 '''
10 Output:
11 Before cleaning: 19978 cols
12 After cleaning: 206 cols
13 '''

```

Listing 3.7 Removing NaN values

In detail, the `.dropna()` at line 3 removes all the rows where the isotopic value of $\delta^{135}\text{Ba}_{136}$ [‰] or $\delta^{138}\text{Ba}_{136}$ [‰] are missing.

Although appealing for its simplicity, the procedure of removing entries containing missing values has some drawbacks and the most significant is the information loss (Zhang, 2016). In particular, when dealing with a large number of features, a substantial number of observations can be removed due to the missing of a single feature, potentially introducing large biases (Zhang, 2016). A possible solution is data imputation, i.e., replacing missing values with imputed values. Several methods and techniques have been developed for data imputation. The easiest approach consists of replacing missing values with the mean, median, or mode of the investigated feature (Zhang, 2016). In pandas, `.fillna()` allows replacing NaN entries with a text or a specific value. Also, the `SimpleImputer()` in scikit-learn lets the imputation of missing values with mean, median, or mode.

A more evolved strategy consists of data imputation with regression (Zhang, 2016). In this case you firstly need to fit a regression model (e.g. linear or polynomial) and then use it for the imputation of missing values (Zhang, 2016). In scikit-learn, the `IterativeImputer()` develops an imputation strategy based on the regression by modelling each feature characterized by missing values as a function of other features.

Encoding categorical features

Most of the available machine learning algorithms do not support the use of categorical (i.e., nominal) features. Therefore, categorical data must be encoded, i.e., converted to a sequence of numbers. In scikit-learn, `OrdinalEncoder()` encodes categorical features as integers (i.e., 0 to `n_categories - 1`).

Data Augmentation

Data augmentation aims at increasing the generalization capability of ML models by increasing the amount of information in our data sets (Maharana et al., 2022). It consists either adding modified copies (e.g., flipper or rotated images in the case of image classification) of the available data or combining the existing features to generate new ones. As an example, Maharana et al. (2022) reports six techniques for data augmentation for image analysis by deep learning. They are (1) symbolic augmentation, (2) rule-based augmentation, (3) graph-structured augmentation, (4) mixup Augmentation, (5) feature space augmentation, and (6) neural augmentation (Maharana et al., 2022). Going in deep details of feature augmentation is far beyond the scope of the present book. However, we will take advantage of data augmentation in Chapter 8, where I followed the strategy proposed by Bestagini et al. (2017). In detail Bestagini et al. (2017) generated new features starting from the available ones, to improve the generalization capability of the investigated data sets.

Data Scaling and transformation

The scaling and transformation of a data set is often a crucial step in ML workflows. Many ML algorithms strongly benefit from a preliminary “standardization” of the investigated data set. As an example, all the algorithms that use the euclidean distance (there are many of them!) as fundamental metrics, may be significantly biased by the introduction of features that are strongly different in magnitude.

Definition: in a standardized data set, all features are centered on zero and their variance is of the same order.

If a feature has a variance that is orders of magnitude greater than the others, it might play a dominant role and prevent the algorithm from correctly learning other features.

The easiest way to standardize a data set is to subtract the mean and scale to unit variance [Eq. (3.1)]:

$$\tilde{x}_e^i = \frac{x_e^i - \mu^e}{\sigma_p^e}, \quad (3.1)$$

where \tilde{x}_e^i and x_e^i are the transformed and original components, respectively, belonging to the sample distribution of the chemical analysis of element e (i.e., SiO₂, TiO₂, etc.), which is characterized by a mean μ^e and a standard deviation σ_p^e .

Scikit-learn implements Eq. (3.1) in the `sklearn.preprocessing.StandardScaler()` class, which is a set of methods (i.e., functions) to scale both the training data set and unknown samples.

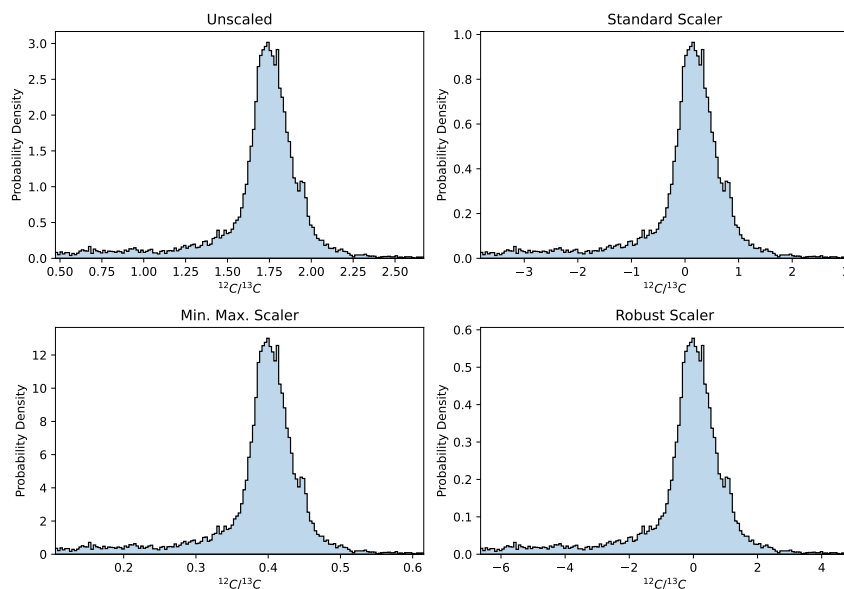


Fig. 3.3 Scaler and transformers, resulting from the code listing 3.3

In addition, scikit-learn implements additional scalers and transformers. In scikit-learn, scaler and transformers perform linear and nonlinear transformations, respectively. For example, *MinMaxScaler()* scales all features belonging to the data set between 0 and 1. Table 3.2 summarizes the main scalers and the transformers available in scikit-learn.

QuantileTransformer() provides nonlinear transformations that shrinks distances between marginal outliers and inliers. Finally, *PowerTransformer()* provides nonlinear transformations in which data are mapped to a normal distribution to stabilize variance and minimize skewness.

Also, the presence of outliers could affect the outputs of your model. If outliers are present in your data set, robust scalers or transformers are more appropriate. By default *RobustScaler()* removes the median and scales the data according to the interquartile range (IQR). Note that the *RobustScaler()* does not perform any removal of the outliers.

If you are lucky and the uncertainties of your estimations are quantified, e.g., by one sigma or a standard error, a preliminary cleaning of your data set could be applied to remove all data where the error exceeds a threshold of your preference.

Finally, making the logarithm of data sometimes helps in reducing the skewness of your sample, under the assumption of a data set deriving by a log-normal distribution (Corlett et al., 1957; Limpert et al., 2001).

Code listing 3.8 shows how to apply different scalers and transformers on the log-transformed $^{12}\text{C}/^{13}\text{C}$ SiC data. Fig. 3.3 shows the results.

```

1 import matplotlib.pyplot as plt
2 import numpy as np
3 from sklearn.preprocessing import MinMaxScaler
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.preprocessing import RobustScaler
6
7 X = np.log10(my_data[[' $^{12}\text{C}/^{13}\text{C}$ ']].dropna().to_numpy())
8
9 scalers = [ ("Unscaled", X),
10            ("Standard Scaler", StandardScaler().fit_transform(X)
11            ),
12            ("Min. Max. Scaler", MinMaxScaler().fit_transform(X))
13            ,
14            ("Robust Scaler", RobustScaler().fit_transform(X))
15            ]
16
17 fig = plt.figure(figsize=(10,7))
18
19 for ix, my_scaler in enumerate(scalers):
20     ax = fig.add_subplot(2,2,ix+1)
21     scaled_X = my_scaler[1]
22     ax.set_title(my_scaler[0])
23     ax.hist(scaled_X, density=True, bins='auto',
24            histtype='stepfilled', color='#BFD7EA', edgecolor='
black')
25     ax.set_xlabel(r' $^{12}\text{C}/^{13}\text{C}$ ')

```

```

24 ax.set_ylabel('Probability Density')
25 ax.set_xlim(np.percentile(scaled_X, 0.5),
26             np.percentile(scaled_X, 99.5))
27
28 fig.set_tight_layout(True)

```

Listing 3.8 Scalers and Trasformers**Table 3.2** Scalers and trasformers in Scikit-learn. Descriptions are taken from the official documentation of Scikit-learn.

Scaler	Description
sklearn.preprocessing.StandardScaler()	Standardize features by removing the mean and scaling to unit variance [Eq. (3.1)].
sklearn.preprocessing.MinMaxScaler()	Transform features by scaling each feature to a given range. The default range is [0,1].
sklearn.preprocessing.RobustScaler()	Scale features using statistics that are robust against outliers. This scaler removes the median and scales the data according to the quantile range. The default quantile range is the inter-quartile range.
Tranformer	Description
sklearn.preprocessing.PowerTransformer()	Apply a power transform feature-wise to make data more Gaussian-like.
sklearn.preprocessing.QuantileTransformer()	Transform features using quantile information. This method transforms features to follow a uniform or normal distribution. Therefore, for a given feature, this transformation tends to spread out the most frequent values.

Compositional Data Analysis (CoDA)

Before applying any statistical method, i.e., including ML algorithms, the underlying assumptions must be always properly addressed. An example is the assumption of normality behind many methods. Other assumptions may regard the topology of the sample space. Geochemical determinations are an example of the, so-called, compositional data (Aitchison, 1982; Aitchison & Egozcue, 2005; Razum et al., 2023), i.e., samples of non-negative multivariate data that have been expressed

relative to a fixed total (typically 1 or percentages summing to 100%). The analysis of compositional data is named compositional data analysis (CoDA; Aitchison, 1984).

In compositional data, the sample space is represented by the Aitchison simplex s^D :

$$s^D = \left\{ x = [x_1, x_2, \dots, x_D] \mid x_i > 0, i = 1, 2, \dots, D; \sum_{i=1}^D x_i = C \right\}. \quad (3.2)$$

where C is a constant, typically 1 or 100. Compositional data typically share two characteristics: a) data are always positive and b) they are characterized by a constant sum (i.e. they are not independent). These characteristics represent an obstacle in the application of most common statistical methods since they often assume that independent input samples ranging in the interval $[-\infty, \infty]$. From the topological point of view, the simplex, i.e., the sample space for compositional vectors, is radically different from the Euclidean space associated with unconstrained data (Aitchison, 1982; Aitchison & Egozcue, 2005; Razum et al., 2023). Therefore, any method relying on the Euclidean distance cannot be used with compositional data, directly.

There are four established transformations that attempt to map the Aitchison simplex to the euclidean space.

Pairwise log ratio transformation (*pwlr*) (Aitchison, 1982; Aitchison & Egozcue, 2005; Razum et al., 2023): The *pwlr* transformation isometrically maps a composition in the D -dimensional Aitchison-simplex to a $D(D - 1)/2$ dimensional space. In detail, it computes each possible log ratio, but accounting for the fact that $\log(A/B) = -\log(B/A)$, and therefore we need only one of them. On pairwise log ratio transformed data, we can apply multivariate methods not relying on the invertibility of the covariance function. The interpretation of *pwlr* transformed data quite simple, since a each component results from a simple operation of division, then transformed by a logarithm to reduce the skew of the resulting features.

The pairwise log ratio (*pwlr*) transformation is given by:

$$pwlr(\mathbf{x}) = [\xi_{ij} \mid i < j = 1, 2, \dots, D], \quad (3.3)$$

where $\xi_{ij} = \ln(c_i/c_j)$. It is noteworthy that the redundancy of the *pwlr* generated $D(D - 1)/2$ features, i.e., and extremely high-dimensional space. Therefore and any of additive log ratio transformation (*alr*), centered log ratio transformation (*clr*) or isometric log ratio transformation (*ilr*) transformations should be preferred in most applications.

Additive log ratio transformation (*alr*) (Aitchison, 1982; Aitchison & Egozcue, 2005; Razum et al., 2023):

$$alr(x) = \left[\ln \frac{x_1}{x_D}, \ln \frac{x_2}{x_D}, \dots, \ln \frac{x_{D-1}}{x_D} \right]. \quad (3.4)$$

The *alr* transformation non-isometrically maps the vectors on the D -dimensional Aitchison-simplex to a $D-1$ dimensional space. The *alr* transformed data can be analysed by multivariate statistical tools not relying on the euclidean distance. As in the case of *pwlr*, the interpretation of *alr* data is quite simple, since they also derive from a simple operation of division followed by a logarithm, to reduce the skew of the resulting features.

Centred log ratio transformation (*clr*):

$$clr(x) = \left[\ln \frac{x_1}{g_m(x)}, \ln \frac{x_2}{g_m(x)}, \dots, \ln \frac{x_D}{g_m(x)} \right]. \quad (3.5)$$

where $g_m(x)$ is the geometric mean of the components in x . The *clr* transformation isometrically maps the vectors in the D -dimensional Aitchison-simplex to a D -dimensional euclidean space. The *clr* transformed data can be analysed by all multivariate tools not relying on a full rank of the covariance (Aitchison, 1982; Aitchison & Egozcue, 2005; Razum et al., 2023).

Orthonormal log ratio transformation (*olr*), also known as isometric log ratio transformation (*ilr*). The most practical way to define an *olr* transformation is by using the concept of balances (Egozcue & Pawłowsky-Glahn, 2005) . A general balance is defined as:

$$b = \sqrt{\frac{rs}{r+s}} \ln \frac{(x_{i1} \cdot x_{i2} \cdot \dots \cdot x_{ir})^{1/r}}{(x_{j1} \cdot x_{j2} \cdot \dots \cdot x_{js})^{1/s}}. \quad (3.6)$$

where x_i and x_j are components in the numerator and denominator, respectively, while r is the number of components in the numerator and s in the denominator. In this way, it is possible to model interpretable variables (e.g., **Glahn2015ModelingData**). Each of the above-mentioned transformations has its unique properties which can be utilized in the compositional data analysis. *Clr* is often used for the construction of compositional biplots and for cluster analysis (van den Boogaart & Tolosana-Delgado, 2013). *Alr* transformation can be safely used in multivariate statistics whenever Mahalanobis distances are involved otherwise due to the oblique coordinates its usage is somewhat limited. *Olr* transformed data can be used safely for any multivariate technique since it is related to the orthonormal basis of the simplex.

In Python both `scikit-bio`⁶ and `pytolite`⁷ provide us with methods in the framework of CoDA.

A working example of data pre-processing

The code listings 3.9 and 3.10 show a step-by-step reproduction of data pre-processing by Boujibar et al. (2021) for a study on the clustering of pre-solar silicon carbide (SiC) grains. Does not matter if you cannot follow the specific astrophys-

⁶ <http://scikit-bio.org>

⁷ <https://pyrolite.readthedocs.io>

ical problem investigated by (Boujibar et al., 2021). The aim of the example is to highlight how to prepare a data set for ML investigations.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import numpy as np
4 from sklearn.preprocessing import StandardScaler
5 from sklearn.preprocessing import RobustScaler
6
7 # Import Data
8 my_data = pd.read_excel("PGD_SiC_2021-01-10.xlsx",
9                         sheet_name='PGD-SIC')
10
11 # limit to features of interest
12 my_data = my_data[['PGD ID', 'PGD Type', 'Meteorite', '12C/13C',
13                  'err+[12C/13C]', 'err-[12C/13C]', '14N/15N',
14                  'err+[14N/15N]', 'err-[14N/15N]',
15                  'd(29Si/28Si)', 'err[d(29Si/28Si)]',
16                  'd(30Si/28Si)', 'err[d(30Si/28Si)]']]
17
18 # Drop NaN
19 my_data = my_data.dropna()
20
21 # Removing M grains with large Si errors
22 my_data = my_data[~((my_data['err[d(30Si/28Si)]']>10) &
23                    (my_data['err[d(29Si/28Si)]']>10) &
24                    (my_data['PGD Type']=='M'))]
25
26 # Excluding C and U grains
27 my_data = my_data[(my_data['PGD Type']=='X') |
28                  (my_data['PGD Type']=='N') |
29                  (my_data['PGD Type']=='AB') |
30                  (my_data['PGD Type']=='M') |
31                  (my_data['PGD Type']=='Y') |
32                  (my_data['PGD Type']=='Z')]
33
34 # Excluding contaminated grains
35 my_data = my_data[~(((my_data['12C/13C']<93.56) &
36                      (my_data['12C/13C']>88.87)) &
37                    ((my_data['14N/15N']<339.94) &
38                     (my_data['14N/15N']>248)) &
39                    ((my_data['d(30Si/28Si)']<50)&
40                     (my_data['d(30Si/28Si)']>-50)) &
41                    ((my_data['d(29Si/28Si)']<50)&
42                     (my_data['d(29Si/28Si)']>-50))
43                    )]

```

Listing 3.9 Scalers and Trasformers

```

1 # Trasform silica isotopic delta to isotopic ratios

```



```

2 Si29_28_0 = 0.0506331
3 Si30_28_0 = 0.0334744
4 my_data['30Si/28Si'] = ((my_data['d(30Si/28Si)']/1000)+1) *
    Si30_28_0
5 my_data['29Si/28Si'] = ((my_data['d(29Si/28Si)']/1000)+1) *
    Si29_28_0
6
7 my_data['log_12C/13C'] = np.log10(my_data['12C/13C'])
8 my_data['log_14N/15N'] = np.log10(my_data['14N/15N'])
9 my_data['log_30Si/28Si'] = np.log10(my_data['30Si/28Si'])
10 my_data['log_29Si/28Si'] = np.log10(my_data['29Si/28Si'])
11
12 # Save to Excel
13 my_data.to_excel("sic_filtered_data.xlsx")
14
15 # Scvaling using StandardScaler() and RobustScaler()
16 X = my_data[['log_12C/13C', 'log_14N/15N', 'log_30Si/28Si',
    'log_29Si/28Si']].values
17
18 scalers = [("Unscaled", X),
19            ("Standard Scaler", StandardScaler().fit_transform(X)),
20            ("Robust Scaler", RobustScaler().fit_transform(X))
21            ]
22
23 # Make pictures
24 fig = plt.figure(figsize=(15,8))
25
26 for ix, my_scaler in enumerate(scalers):
27     scaled_X = my_scaler[1]
28     ax = fig.add_subplot(2,3,ix+1)
29     ax.set_title(my_scaler[0])
30     ax.scatter(scaled_X[:,0], scaled_X[:,1],
31              marker='o', edgecolor='k', color='#db0f00',
32              alpha=0.6, s=40)
33     ax.set_xlabel(r'$\log_{10}[\text{C}^{12}/\text{C}^{13}]$')
34     ax.set_ylabel(r'$\log_{10}[\text{N}^{14}/\text{N}^{15}]$')
35
36     ax1 = fig.add_subplot(2,3,ix+4)
37     ax1.set_title(my_scaler[0])
38     ax1.scatter(scaled_X[:,2], scaled_X[:,3],
39              marker='o', edgecolor='k', color='#db0f00',
40              alpha=0.6, s=40)
41     ax1.set_xlabel(r'$\log_{10}[\text{Si}^{30}/\text{Si}^{28}]$')
42     ax1.set_ylabel(r'$\log_{10}[\text{Si}^{29}/\text{Si}^{28}]$')
43
44 fig.set_tight_layout(True)

```

Listing 3.10 Scalers and Trasformers

In detail, code listing 3.9 starts with the importing of all the libraries and methods needed to achieve your goal, i.e., pandas, matplotlib, numpy plus the StandardScaler and RobustScaler from scikit-learn. Then the workflow starts. At line 8, we create a pandas DataFrame, named my_data, importing the data set of SiC analyses from

ExcelTM. All the successive steps prepare *my_data* for the processing by a ML algorithm.

To note, in code listing 3.9:

- Line 12 Limits the features to the ones of interest.
- Line 19 Removes non numerical data, i.e. Not a Number (NaN).
- Line 22 Removes all the rows lanel by 'M' in the 'PGD Type' column and characterized by large errors.
- Line 27 Limits the data set to specific labels in the PGD Type column, i.e., specific SiC clases, i.e., X, N, AB, M, Y, and Z, in agreement with the current classification (Stephan et al., 2021).
- Line 34 Removes contaminated grains, i.e., characterized by an isotopic signature too much similar to the one of the Earth.

Then, in code listing 3.10:

- Lines 2-5 Convert Silica values from δ notation to isotopic ratios.
- Lines 7-10 Apply a log-normal transformation, in agreement with *alr* CoDA transformation.
- Line 13 Save *my_data* to ExcelTM to have a record the results of pre-processing before the scaling.
- Line 16 Defines *X*, a 4 features numpy array in the shape accepted by most scikit-learn ML algorithms.
- Line 18 Defines three scenarios: a) unscaled data; b) scaling with StandardScaler(); c) scaling with RobustScaler().
- Lines 25-42 Perform the scaling at line 27 and make the diagrams reported in Fig. 3.4.

Fig. 3.4 shows the results of code listings 3.9 and 3.10. As expected, the application of different scalers and transformers does not change the structure of data. However, they strongly affect the position and the spread of the investigated features. As an example, the logarithm of $^{12}\text{C}/^{13}\text{C}$ range from 0 and 4 when unscaled, with a mean at about 1.7 (See Fig. 3.3 also). Both the Standard and the Robust Scalers centres the data set to 0, using the mean and the median, respectively, and they produces different spreads, since the Robust Scaler also account for the presence of outliers. For symmetric distributions in the absence of outliers, we expect similar results for the Standard and Robust Scalers.

3.4 Train a Model

Fig. 3.5 reports a cheat-sheet guiding us in model selection for the scikit-learn library. Scikit-learn allows working in the fields of both unsupervised (i.e., clustering and dimensionality reduction) and supervised (i.e., regression and classification)

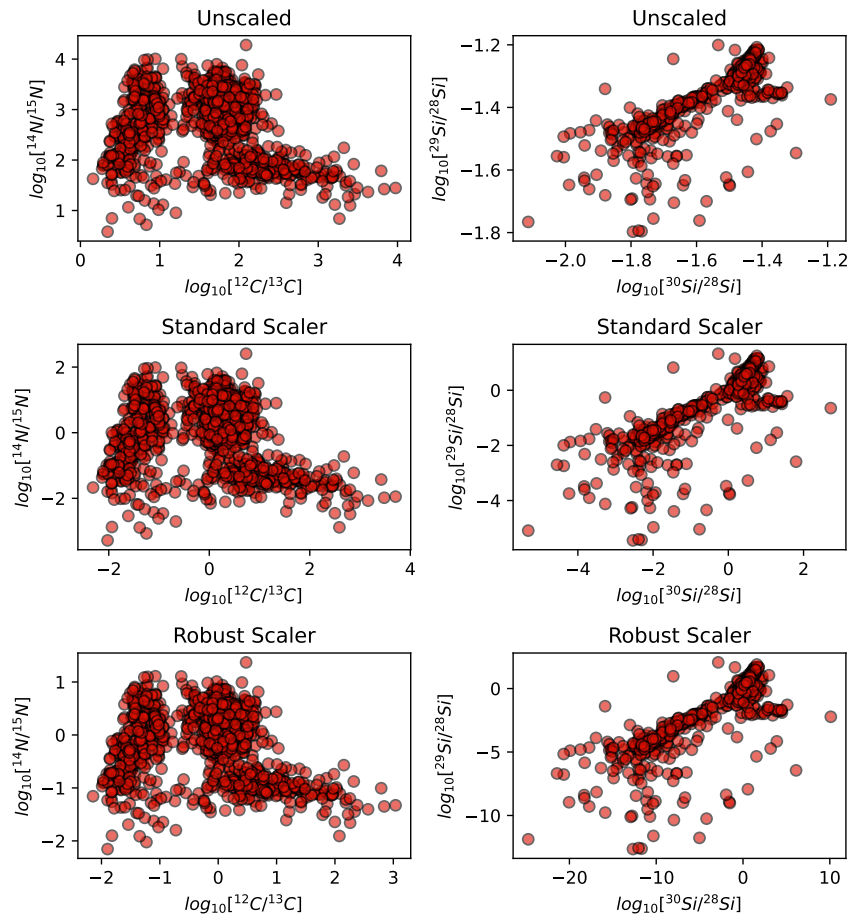


Fig. 3.4 Scaling SiC data with Scikit-learn

learning. About supervised learning, examples of classification algorithms are the Support Vector Classifier (SVC, cfr. par. 7.9) and the K-neighbors (cfr. par.7.10). In the field of regression, examples are the Stochastic Gradient Descent (SGD), the support vector (SVR) and the ensemble regressors. Examples of unsupervised learning are the Locally linear embedding (LLE, cfr. par. 4.3) and the Principal Component Analysis (PCA, cfr. par. 4.2) if we point to the dimensionality reduction. For the clustering, examples are the K-means, Gaussian Mixture Models (GMM, cfr. par. 4.9) and the spectral clustering. We will discuss in detail the details of the most popular ML algorithms in chapters 7 and 4 dealing with supervised and unsupervised learning, respectively.

In the following, I report a quick example about the training of an unsupervised algorithm on the SiC analyses that we are using as a proxy for a scientific data set in

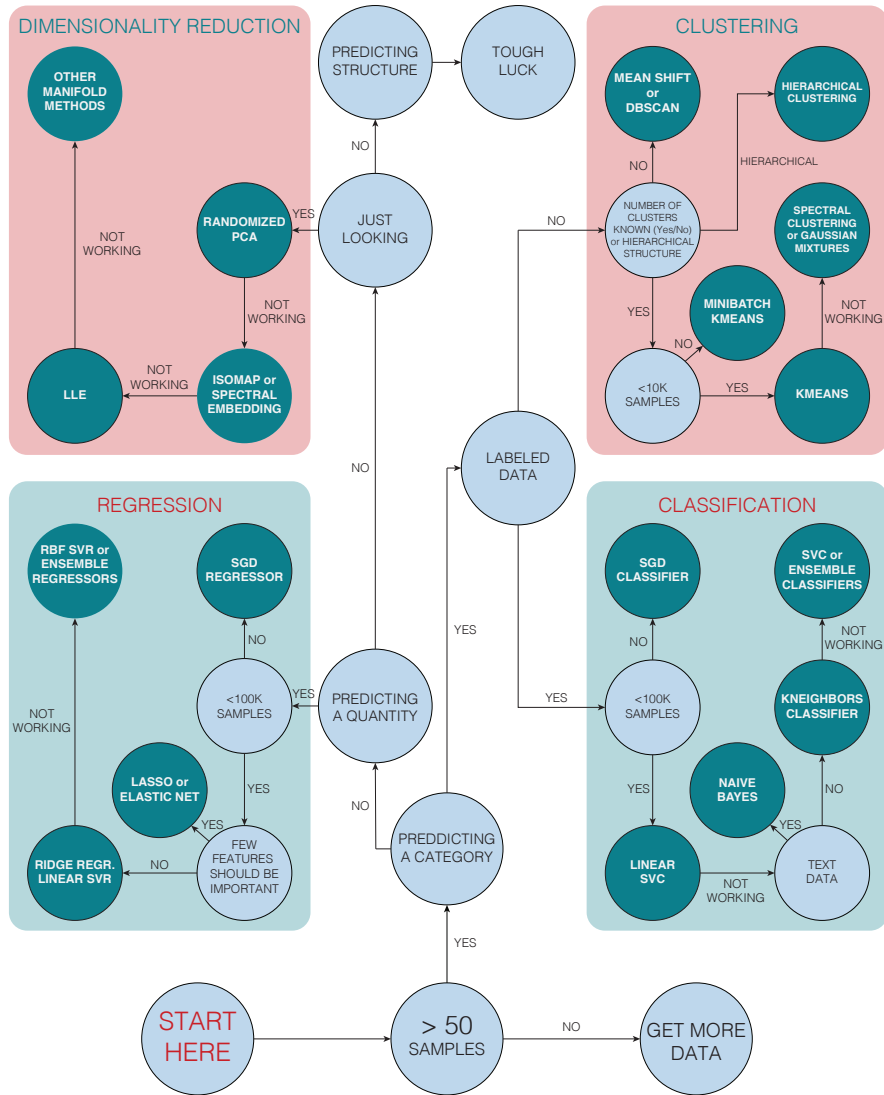


Fig. 3.5 Scikit-learn algorithm cheat-sheet. Modified from the official documentation of scikit-learn

the field of planetary sciences. Code listing 3.11 shows how to perform the clustering by Gaussian Mixtures (crf. section. 4.9) algorithm on the SiC data pre-processed by code listings 3.9 and 3.10. As you can note, the core of the training is at line 12, where I parameterized the *GaussianMixture()* algorithm, i.e., defining 9 clusters and fixing the random state of the pseudo random number generator to allow the reader reproducing my results exactly.

Generally speaking, the *.fit()* method in scikit-learn launch the training of ML algorithms. Then, using the *.predict()* method, we get the results or we transfer the obtained knowledge to unknown data. Fig. 3.6 displays the result of the clustering by *GaussianMixture()*, i.e., lines 16-29 of code listing 3.11.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.mixture import GaussianMixture as GMM
5
6 my_colors = ['#AF41A5', '#0A3A54', '#0F7F8B', '#BFD7EA', '#F15C61',
7             '#C82127', '#ADADAD', '#FFFFFF', '#EABD00']
8
9 scaler = StandardScaler().fit(X)
10 scaled_X = scaler.transform(X)
11
12 my_model = GMM(n_components = 9, random_state=(42)).fit(scaled_X)
13
14 Y = my_model.predict(scaled_X)
15
16 fig, ax = plt.subplots()
17
18 for my_group in np.unique(Y):
19     i = np.where(Y == my_group)
20     ax.scatter(scaled_X[i,0], scaled_X[i,1],
21             color=my_colors[my_group],
22             label=my_group + 1, edgecolor='k', alpha=0.8)
23
24 ax.legend(title='Cluster')
25
26 ax.set_xlabel(r'$\log_{10}[\text{C}^{12}/\text{C}^{13}]$')
27 ax.set_ylabel(r'$\log_{10}[\text{N}^{14}/\text{N}^{15}]$')
28
29 fig.tight_layout()

```

Listing 3.11 Application of the *GaussianMixture()* algorithm to SiC data.

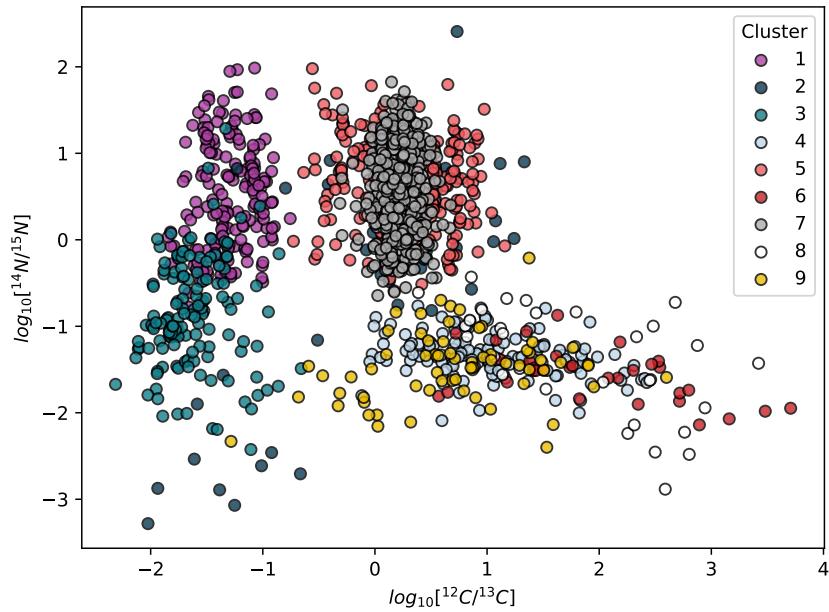


Fig. 3.6 The clustering resulting by the application of the *GaussianMixture()* algorithm to SiC data, code listing 3.11

3.5 Model Validation and Testing

The validation and testing of a model is the third crucial step, i.e., after the pre-processing and the training, in ML. They allow you to evaluate the “goodness” of a model.

Splitting the investigated data set in three portions

The approach of model validation and model testing by splitting the investigated data set into three portions is clearly described by Hastie et al. (2017): the best approach for model assessment in ML “is to randomly divide the data set into three parts: a training set, a validation set, and a test set. The training set is used to fit the models; the validation set is used to estimate prediction error for model selection; the test set is used for assessment of the generalization error of the final chosen model.”

Typically, we use the training data set to train a selection of candidate models. Candidate models could be different algorithms, a single algorithm tuned with different hyper-parameters (i.e., one or more variables that affect the behaviour on an algorithm), or a combination of both. Then, we use the validation data set to evaluate candidate models and chose the most performing one. Finally, we test the

selected model using the test data set. As an example, the `train_test_split()` method in scikit-learn allows the random splitting of a data set into two portions, e.g., train plus validation and test sets, respectively. Repeating the `train_test_split()` method on the training plus validation set, will allow us to divide it further in the training and validation set.

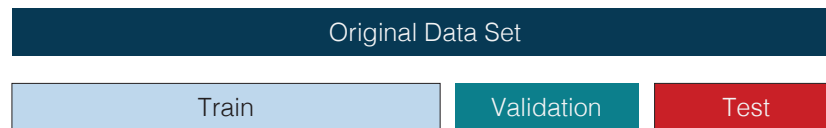


Fig. 3.7 Splitting the investigated data set in three portions

```

1 from sklearn import preprocessing
2 from sklearn.model_selection import train_test_split
3
4 le = preprocessing.LabelEncoder()
5 le.fit(my_data['PGD Type'])
6 y = le.transform(my_data['PGD Type'])
7
8 X_train_valid, X_test, y_train_valid, y_test = train_test_split(
9     X, y, test_size=0.20)
10
11 X_train, X_valid, y_train, y_valid = train_test_split(
12     X, y, test_size=0.25)

```

Listing 3.12 Scalers and Trasformers

To note, the statements of lines 4 to 6 in Code listing 3.12 simply convert the labels referring to a specific SiC Class, i.e., PGD Types still present in `my_data` after the pre-processing, like M, Y, Z, X, AB, N to numbers from 0 to 5. It will allow an easier management of labels during the execution of supervised methods in the fields of regressions and classification.

Cross-Validation

The partitioning of the investigated data into three sets (i.e., train, validation, and test) drastically reduces the number of samples exposed during the training. As a consequence, the training process can be biased by the non-reliability of the training set due, as an example, to the presence of borderline cases in the test data.

The cross-validation (CV) procedure can overcome this limitation and, therefore, can be minded as an evolution of the static division of the investigated set of data in three parts.

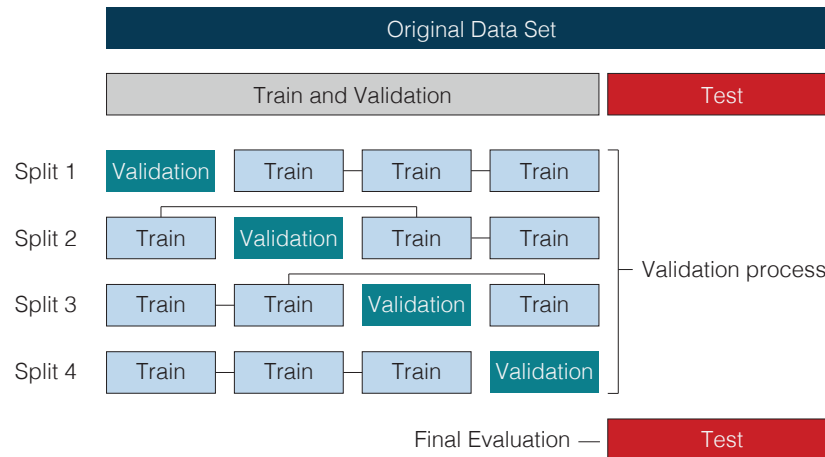


Fig. 3.8 Example of k-fold cross-validation

In the cross-validation procedure, the initial set of data is initially split into two portions, i.e., the test plus a joint training and validation sets.

```

1 from sklearn import svm
2 from sklearn import preprocessing
3 from sklearn.model_selection import cross_validate
4
5 le = preprocessing.LabelEncoder()
6 le.fit(my_data['PGD Type'])
7 y = le.transform(my_data['PGD Type'])
8
9 my_model = svm.SVC(kernel='linear', C=1, random_state=42)
10
11 cv_results = cross_validate(my_model, scaled_X, y, cv=5,
12                             scoring='accuracy')
13
14 print(cv_results['test_score'])
15
16 '''
17 Output:
18 [0.98529412 0.97785978 0.9704797 0.98154982 0.95940959]
19 '''

```

Listing 3.13 Application of a linear Support Vector Classifier to SiC data

Then, in the most basic strategy of cross-validation, named k-fold CV, the joint training and validation set is split into k smaller batches.

The following steps consist of repeating the training and the validation for the candidate model as follows: a) we use k-1 folds as the training set; b) the result of the training is validated against the remaining k set of the data; c) we repeat the procedure for the next split.

The performance of the candidate model can be estimated using the selected metrics and averaging the obtained k results. As an example code listing 3.13 shows how to perform the k-fold CV in scikit-learn using the `cross_validate()` method. After converting the 5 labels in the 'PGD Type' columns, i.e., M, Y, Z, X, AB, N to a numeric index ranging from 0 to 5 (lines from 5 to 7), we define a linear Support Vector Classifier (crf. sec. ??) characterized by a C hyper-parameter equal to 1 (line 9). Finally, we perform the k-fold CV dividing the data set in 5 folds and using the accuracy as metrics. As expected, we obtain 5 estimations for the accuracy, one for each split (Fig. 3.8).

Using the k-fold cross-validation, n-different candidate models can be evaluated by repeating n-times the k-fold CV. As an example, the `GridSearchCV()` method in scikit-learn performs an exhaustive search (i.e., evaluate all possible combinations of the proposed parameters) over a range of parameter values for a specific estimator (i.e., ML algorithm).

```

1 from sklearn import svm
2 from sklearn import preprocessing
3 from sklearn.model_selection import GridSearchCV
4
5 le = preprocessing.LabelEncoder()
6 le.fit(my_data['PGD Type'])
7 y = le.transform(my_data['PGD Type'])
8
9 parameters = {'kernel':('linear', 'rbf'), 'C':[0.1, 1, 10]}
10 my_model = svm.SVC()
11
12 my_grid_search = GridSearchCV(my_model, parameters,
13                               cv = 4, scoring='accuracy')
14
15 my_grid_search.fit(scaled_X, y)

```

Listing 3.14 Scalers and Transformers

As an example, the `GridSearchCV()` can be used to evaluate the best choice for the hyperparameters of an ML algorithm, such as the C parameter and the 'kernel function' of a Support Vector Machine (crf. par. 7.9). In detail, the code listing 3.14 shows how to define the grid for the selected hyperparameters at line 9. At line 10, we define the model, i.e., a Support Vector Machine. At line 12, we define the grid search for our SVC model, using the parameters defined at line 9, a 4-fold cross-validation, and using the accuracy as metrics. Finally, at line 15 we physically perform the grid search for all the combinations among the defined parameters. In detail, at line 9, we

defined two Kernel functions and three values for C. Therefore, the grid search will perform 6 cross-validations splitting, each time, the scaled_X data set in 4 folds.

Code listing 3.15 shows how to get the results of a *GridSearchCV()*. In detail the *best_estimator_*, *best_score_*, and *cv_results_* attributes provide us with the best combination of hyperparameters, the best score, and a dictionary containing all the results, respectively.

```

1 In [01]: my_grid_search.best_estimator_
2 Out[01]: SVC(C=10, kernel='linear')
3
4 In [02]: my_grid_search.best_score_
5 Out[02]: 0.9778761061946903
6
7 In [03]: my_grid_search.cv_results_
8 Out[03]:
9 {'mean_fit_time': array([0.00605977, 0.02105349, 0.00482285,
10      0.01113951, 0.00554657, 0.00662667]),
11  'std_fit_time': array([3.7539e-04, 6.0314e-04, 2.1346e-04,
12      7.0395e-04, 5.5384e-04, 3.1989e-05]),
13  'mean_score_time': array([0.00242817, 0.01987976, 0.00181627,
14      0.00979179, 0.00133586, 0.00618142]),
15  'std_score_time': array([7.4277e-05, 1.6316e-03, 1.6929e-04,
16      2.7074e-04, 2.2063e-04, 6.4881e-04]),
17  'param_C': masked_array(data=[0.1, 0.1, 1, 1, 10, 10],
18      mask=[False, False, False, False, False, False],
19      fill_value='?', dtype=object),
20  'param_kernel': masked_array(data=['linear', 'rbf', 'linear',
21      'rbf', 'linear', 'rbf'],
22      mask=[False, False, False, False, False, False],
23      fill_value='?', dtype=object),
24  'params': [{'C': 0.1, 'kernel': 'linear'},
25      {'C': 0.1, 'kernel': 'rbf'},
26      {'C': 1, 'kernel': 'linear'},
27      {'C': 1, 'kernel': 'rbf'},
28      {'C': 10, 'kernel': 'linear'},
29      {'C': 10, 'kernel': 'rbf'}],
30  'split0_test_score': array([0.92330383, 0.8879056, 0.98230088,
31      0.91150442, 0.97935103, 0.97050147]),
32  'split1_test_score': array([0.9380531, 0.88495575, 0.97935103,
33      0.92625369, 0.98525074, 0.97935103]),
34  'split2_test_score': array([0.92330383, 0.89380531, 0.97345133,
35      0.91740413, 0.97640118, 0.96460177]),
36  'split3_test_score': array([0.91740413, 0.88495575, 0.96755162,
37      0.90560472, 0.97050147, 0.96460177]),
38  'mean_test_score': array([0.92551622, 0.8879056, 0.97566372,
39      0.91519174, 0.97787611, 0.96976401]),
40  'std_test_score': array([0.00762838, 0.00361282, 0.00566456,
41      0.00762838, 0.00531792, 0.0060364]),
42  'rank_test_score': array([4, 6, 2, 5, 1, 3], dtype=int32)}

```

Listing 3.15 getting the results of a *GridSearchCV()*

To achieve the final validation, i.e., as reported in Fig. 3.8, we required a preliminary splitting of `scaled_X` in two portions, i.e., train plus validation and test sets using the `train_test_split()` method.

Leave One Out Cross-Validation

The Leave One Out (or LOO) cross validation is a limit case of the k-fold CV. Using the LOO approach, each train set is created by taking all the samples except one. The test set is then created using the sample left out. In the LOO approach, the cross validation typically performs over all the potential training sets, i.e., each sample of the investigated data set. Code listing 3.16 highlights how to perform a LOO Cross-Validation on the same study case reported in code listing 3.13. Fig. 3.9 reports the results of the LOO Cross-Validation of code listing 3.16. In the specific case study, code listing 3.13 cross-validates 1356 models, each considering one sample of the investigated samples as test data set, with all the others utilized for the training.

```

1 import numpy as np
2 from sklearn import svm
3 from sklearn.model_selection import LeaveOneOut
4 from sklearn.model_selection import cross_validate
5 import matplotlib.pyplot as plt
6
7 loo = LeaveOneOut()
8
9 my_model = svm.SVC(kernel='linear', C=1, random_state=42)
10
11 cv_results = cross_validate(my_model, scaled_X, y, cv=loo,
12                             scoring='accuracy')
13
14 fig, ax = plt.subplots()
15 my_x = [0,1]
16 my_height = [np.count_nonzero(cv_results['test_score'] == 0),
17              np.count_nonzero(cv_results['test_score'] == 1)]
18 my_bar = ax.bar(x = my_x, height=my_height, width=1,
19                 color=['#F15C61', '#BFD7EA'],
20                    tick_label=['wrongly classified', 'correctly
21                               classified'],
22                    edgecolor='k')
23 ax.set_ylabel('occurrences')
24 ax.set_title('LOO cross validation n = {}'.format(len(scaled_X)))
25 ax.bar_label(my_bar)

```

Listing 3.16 Leave One Out Cross-Validation

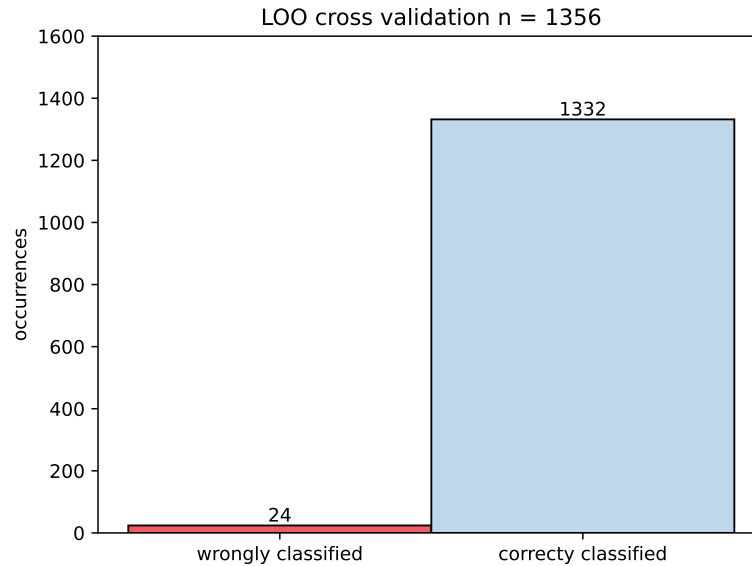


Fig. 3.9 Scikit-learn algorithm cheat-sheet. Modified from the official documentation of scikit-learn

Metrics

As you have probably noticed, the validation process is based on a metric. As an example, all the code listings 3.13, 3.13, 3.16 specify `scoring='accuracy'`. It means that all the reported examples use accuracy as a metric to quantify the “goodness” of a model. To note there are a plethora of metrics that you can potentially use to validate a model. As an example, Tables 3.3, 3.4, and 3.5 report the metrics that are available in scikit-learn for classification, regression and clustering, respectively⁸. To note, all the metrics reported in Tables 3.3, 3.4, and 3.5 follow the same convention: the goodness of the model increases with increasing the value returned by the selected metric. In other words, higher values for a specific metric are better than lower ones.

⁸ https://scikit-learn.org/stable/modules/model_evaluation.html

Table 3.3 Metrics and scoring for Classification

Method in metrics	keywords	Description	Eq.
.accuracy_score	'accuracy'	Accuracy classification score	[12]
.balanced_accuracy_score	'balanced_accuracy'	Compute the balanced accuracy	[12]
.top_k_accuracy_score	'top_k_accuracy'	Top-k Accuracy classification	[12]
.average_precision_score	'average_precision'	Compute the average precision	[12]
.brier_score_loss	'neg_brier_score'	Compute the Brier score loss	[12]
.precision_score	'precision' 'precision_micro' 'precision_macro' 'precision_weighted' 'precision_samples'	Compute the precision	[12]
.f1_score	'f1' 'f1_micro' 'f1_macro' 'f1_weighted' 'f1_samples'	Compute the F1 score	[12]
.recall_score	'recall' 'recall_micro' 'recall_macro' 'recall_weighted' 'recall_samples'	Compute the recall	[12]
.jaccard_score	'jaccard' 'jaccard_micro' 'jaccard_macro' 'jaccard_weighted' 'jaccard_samples'	Jaccard similarity coefficient	[12]
.roc_auc_score	'roc_auc' 'roc_auc_ovr' 'roc_auc_ovo' 'roc_auc_ovr_weighted' 'roc_auc_ovo_weighted'	Area Under the Receiver Operating Characteristic Curve (ROC AUC)	[12]

Over-fitting and Under-fitting

Avoiding Data leakage

3.6 Model Deploy and Persistence

The deployment and persistence of a machine learning model is the last step of our workflow. There are many options to secure the persistence of a model. Examples

Table 3.4 Metrics and scoring for the Regression

Method in metrics	keywords	Description	Eq.
.explained_variance_score	'explained_variance'	Explained variance regression score.	[12]
.max_error	'max_error'	Calculates the maximum residual error.	[12]
.mean_absolute_error	'neg_mean_absolute_error'	Mean absolute error regression loss.	[12]
.mean_squared_error	'neg_mean_squared_error'	Mean squared error regression loss.	[12]
	'neg_root_mean_squared_error'	Root mean squared error regression loss.	
.mean_squared_log_error	'neg_mean_squared_log_error'	Mean squared logarithmic error regression loss.	[12]
.median_absolute_error	'neg_median_absolute_error'	Median absolute error regression loss.	[12]
.r2_score	'r2'	R^2 - coefficient of determination score.	[12]
.mean_poisson_deviance	'neg_mean_poisson_deviance'	Mean Poisson deviance regression loss.	[12]
.mean_gamma_deviance	'neg_mean_gamma_deviance'	Mean Gamma deviance regression loss.	[12]
.mean_absolute_percentage_error	'neg_mean_absolute_percentage_error'	Mean absolute percentage error regression loss.	[12]

are the use of pickles, joblib's pipelines, the Open Neural Network Exchange Format (ONNX), and the Predictive Model Markup Language (PMML) format.

In detail, the pickle module allows the serializing and de-serializing of Python object structures, like your ML models. Saving a model using pickles is straightforward. As an example, code listing XX show how to guarantee a model persistence using pickles. When working with large sets of data, joblib's pipelines are more efficient than pickles (code listing xx).

To note, these two first approaches, i.e., pickle and joblib, share some maintainability and security issues. As an example, pickle and joblib assume the deployment of models in the same environment, i.e., the same version of libraries and Python core, where they have been saved.

Due to the above-reported issues, I suggest using the ONNX and PMML formats. They aim to improve model portability on different computing architectures and long term archiving. As an example, code listing XX report how to secure your ML model using ONNX.

Table 3.5 Metrics and scoring for the Clustering

Method in metrics	keywords	Description	Eq.
.adjusted_mutual_info_score	'adjusted_mutual_info_score'	Adjusted Mutual Information	[12]
		between two clusterings	[12]
.adjusted_rand_score	'adjusted_rand_score'	Rand index adjusted for chance	[12]
.completeness_score	'completeness_score'	Completeness metric of a cluster labeling given a ground truth	[12] [12]
.fowlkes_mallows_score	'fowlkes_mallows_score'	Measure the similarity of two clusterings of a set of points	[12] [12]
.homogeneity_score	'homogeneity_score'	Homogeneity metric of a cluster labeling given a ground truth	[12] [12]
.mutual_info_score	'mutual_info_score'	Mutual Information	[12]
		between two clusterings	[12]
.normalized_mutual_info_score	'normalized_mutual_info_score'	Normalized Mutual Information	[12]
		between two clusterings	[12]
.rand_score	'rand_score'	Rand index	
.v_measure_score	'v_measure_score'	V-measure cluster labeling given a ground truth	

Part II
Unsupervised Learning

Chapter 4

Unsupervised Machine Learning Methods

4.1 Unsupervised Algorithms

As introduced in chapter 1, the unsupervised learning process acts with unlabeled training data in the attempt of exerting significant patterns in the investigated data set. In the present chapter, I am going to gently introduce the unsupervised algorithms for dimensionality reduction and clustering reported in Fig.3.5. Finally, I will provide some specific references to allow the readers in going deeper into the mathematics governing these ML methods. In detail, I will start describing the algorithms for dimensionality reduction. They are the Principal Component Analysis and methods based on the Manifold Learning. Then, I will describe clustering methods, e.g., Hierarchical Clustering, DBSCAN, Mean Shift, K-Means, Spectral Clustering, and Gaussian Mixtures Models.

4.2 Principal component Analysis

Principal component analysis (PCA) is a multivariate statistical method that attempts to extract relevant information from a data set and represents it in a lower-dimensional space (Jolliffe & Cadima, 2016). It aims at increasing the interpretability of a data set by reducing the dimensionality of the problem but, at the same time, minimizing information loss (Jolliffe & Cadima, 2016). In detail, it derives new uncorrelated variables, named principal components, that maximize variance (Jolliffe & Cadima, 2016). Mathematically speaking, the PCA is an eigenvalue/eigenvector problem (Jolliffe & Cadima, 2016). Now consider a d -dimensional sample set $X = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$. The sample set X is equivalent to a $n \times d$ data matrix \mathbf{X} , whose j th column is the vector \mathbf{x}_j of observations on the j th variable (Jolliffe & Cadima, 2016). We look for a linear combination of the columns of matrix \mathbf{X} with maximum variance (Jolliffe & Cadima, 2016). Such linear combinations are given by:

$$\sum_{j=1}^d a_j \mathbf{x}_j = \mathbf{X}\mathbf{a}, \quad (4.1)$$

where $\mathbf{a} = \{a_1, a_2, \dots, a_d\}$ is a vector of constants (Jolliffe & Cadima, 2016). The variance of any linear combination defined by the Eq. 4.1 is given by (Jolliffe & Cadima, 2016):

$$\text{var}(\mathbf{X}\mathbf{a}) = \mathbf{a}^T \mathbf{S} \mathbf{a}, \quad (4.2)$$

where \mathbf{S} is the sample covariance matrix associated with the data set (Jolliffe & Cadima, 2016).

The solution of the problem, i.e., identifying the linear combination with maximum variance, consists of finding a d -dimensional vector \mathbf{a} which maximizes the quadratic form $\mathbf{a}^T \mathbf{S} \mathbf{a}$ (Jolliffe & Cadima, 2016). To achieve a defined solution, the most common restriction assumes working with unit-norm vectors, i.e., requiring $\mathbf{a}^T \mathbf{a} = 1$. Now the problem is equivalent to maximizing the following relation (Jolliffe & Cadima, 2016):

$$\mathbf{a}^T \mathbf{S} \mathbf{a} - \lambda (\mathbf{a}^T \mathbf{a} - 1), \quad (4.3)$$

After a differentiation with respect to the vector \mathbf{a} , and equating to the null vector, we have (Jolliffe & Cadima, 2016):

$$\mathbf{S} \mathbf{a} = \lambda \mathbf{a}. \quad (4.4)$$

In the Eq. 4.4, \mathbf{a} is a unit-norm eigenvector and λ is the corresponding eigenvalue of \mathbf{S} (Jolliffe & Cadima, 2016). The full set of eigenvectors of \mathbf{S} are the solutions to the problem of obtaining up to d new linear combinations $\mathbf{X}\mathbf{a}_k = \sum_{j=1}^d a_{jk} \mathbf{x}_j$, which successively maximize variance, subject to uncorrelatedness with previous linear combinations (Jolliffe & Cadima, 2016; Jolliffe, 2002).

4.3 Manifold Learning

The main idea behind Manifold Learning methods is that although natural data sets are often depicted in very high-dimensional spaces, they can be successfully described in lower dimensions since the processes generating the data are often characterized by few degrees of freedom (Zheng & Xue, 2009). From the mathematical point of view, Manifold Learning methods try modeling the data as “lying on or near a low-dimensional manifold embedded in a higher-dimensional space (Zheng & Xue, 2009)”. Describing in detail the mathematics behind Manifold Learning is behind the scope of the present book. In the following, I will introduce the basic concepts of Manifold Learning and I strongly encourage you in going into deeper details if you intend use these techniques in your researches (Zheng & Xue, 2009).

Manifold: A d -dimensional manifold \mathbb{M} is a topological space that is locally homeomorphic with respect to \mathbb{R}^d .

Homomorphism: a map from one algebraic structure to another of the same type that preserves all the relevant structures.

Embedding: an embedding of a manifold \mathbb{M} into \mathbb{R}^d is a smooth homeomorphism from \mathbb{M} to a subset of \mathbb{R}^d .

4.3.1 Isometric Feature Mapping

The isometric feature mapping (Isomap) is an ML algorithm that is “capable of discovering the nonlinear degrees of freedom that underlie complex natural observations Tenenbaum et al. (2000).” It consists of three main steps: 1) Construct a neighborhood graph; 2) compute the shortest paths; 3) Construct d -dimensional embedding Tenenbaum et al. (2000). In the practice, Isomap search for a lower-dimensional embedding while maintaining geodesic distances between all points. In scikit-learn the method *Isomap()* performs the Isometric Feature Mapping.

4.3.2 Locally Linear Embedding

Locally Linear Embedding (LLE) (Roweis & Saul, 2000), a ML algorithm that “computes low-dimensional, neighborhood-preserving embeddings of high-dimensional inputs (Roweis & Saul, 2000)”. In the practice, LLE maps the inputs into a single global coordinate system of lower dimensionality (Roweis & Saul, 2000). Also, its optimizations do not involve local minima (Roweis & Saul, 2000). In the practice, LLE search for a lower-dimensional projection of the data while preserving the distances within local neighborhoods. In scikit-learn the method *LocallyLinearEmbedding()* performs the LLE.

4.3.3 Laplacian Eigenmaps

A Laplacian Eigenmap (Belkin & Niyogi, 2003) first develops a graph incorporating neighborhood information starting from a data set in RD. Then it utilizes the Laplacian to compute a low-dimensional representation. Practically, Laplacian Eigenmaps consists of three main steps: 1) Constructing the adjacency graph; 2) Choosing the weights; 3) Computing Eigenmaps.

4.3.4 Hessian Eigenmaps

Hessian eigenmaps (Donoho & Grimes, 2003) are similar to Laplacian eigenmaps but they replace the Laplacian operator with the Hessian. The main difference between Laplacian and Hessian eigenmaps relies on the capability of Hessian eigenmaps to overcome the convexity limitation of Laplacian eigenmaps (Zheng & Xue, 2009). In scikit-learn Hessian eigenmaps can be performed with the *LocallyLinearEmbedding()*, i.e., the same that we use for the LLE, but specifying *method = 'hessian'*.

4.4 Hierarchical Clustering

Hierarchical clustering algorithms (Johnson, 1967) build a hierarchical representation of the data set structure, where clusters at each level of the hierarchy are assembled by merging or splitting clusters at the next lower or upper level, respectively (Hastie et al., 2017; Johnson, 1967). There are two main paradigms for hierarchical clustering: agglomerative (i.e., bottom-up) and divisive (i.e., top-down). Agglomerative strategies start from the bottom where every single observation forms a cluster (Hastie et al., 2017; Johnson, 1967). Then, at each successive level, the algorithm recursively merges a selected pair of clusters into a single cluster. The criterion for merging, i.e., linkage, is based on a specific metric of dissimilarity (Hastie et al., 2017; Johnson, 1967). On the contrary, the divisive approach starts from a single cluster containing all the observations and, at each subsequent level, it recursively splits one of the existent clusters into two new clusters using a metric of dissimilarity (Hastie et al., 2017; Johnson, 1967). In scikit-learn, the method *AgglomerativeClustering()* performs the agglomerative hierarchical clustering, i.e., using a bottom up approach. To define the linkage criterion, that is based on the concept of dissimilarity, please consider two set of observations, i.e. clusters, G and H . The hierarchical clustering estimates the dissimilarity $d(G, H)$ between G and H on the set of pairwise observations dissimilarities d_{ij} where one member of the pair i is in G and the other j is in H (Hastie et al., 2017). Using *AgglomerativeClustering()*, the linkage criterion could be: single, complete, group average or Ward (Table 4.1)

Table 4.1 linkage options in *AgglomerativeClustering()*

parameter	Equation	Note
linkage='single'	$d_{sl}(G, H) = \min_{\substack{i \in G \\ j \in H}} d_{ij}$	uses the minimum of the distances between all observations of the two sets
linkage='complete'	$d_{cl}(G, H) = \max_{\substack{i \in G \\ j \in H}} d_{ij}$	uses the maximum distances between all observations of the two sets
linkage='average'	$d_{ga}(G, H) = \frac{1}{n_g n_h} \sum_{i \in G} \sum_{j \in H} d_{ij}$	uses the average of the distances of each observation of the two sets

Finally, the Ward's linkage criterion (set as default in scikit-learn) states that the distance between two clusters, G and H , is how much the sum of squares will increase when we merge them:

$$\Delta(G, H) = \frac{|G||H|}{|G| + |H|} \|\mathbf{m}_G + \mathbf{m}_H\|^2, \quad (4.5)$$

where Δ is the “merging cost” of combining the clusters G and H . Also, \mathbf{m} , $|G|$ and $|H|$ are the center of cluster and the cardinal of the set G and H , respectively.

The dissimilarities d_{ij} , can be estimated using different metrics. Using the method *AgglomerativeClustering()*, they can be, among others, “euclidean” or “manhattan”. In the case of “ward” linkage, the only accepted metric is “euclidean” (see Eq. 4.5).

4.5 DBSCAN

The Density-Based Spatial Clustering of Applications with Noise, i.e., DBSCAN, algorithm relies on a “density-based notion of clusters which is designed to discover clusters of arbitrary shape (Ester et al., 1996)”. Topologically, DBSCAN identifies a core sample if there exist a pre-defined minimum number of other samples (i.e., neighbors of the core sample) within a distance of ϵ (Ester et al., 1996). A cluster is a set of core samples plus their neighbors. Any sample that is neither a core sample nor a neighbor, i.e., it is at least ϵ far from any core sample, is marked as an outlier (Ester et al., 1996). The DBSCAN does not require the number of clusters to be specified.

4.6 Mean Shift

The Mean Shift is a nonparametric technique for clustering analysis (Comaniciu & Meer, 2002).

The Mean Shift algorithm starts performing a kernel density estimation in the investigated d -dimensional feature space (Derpanis, 2005). As a result, the kernel density estimation defines an empirical probability density function where “dense regions” defines local maxima (i.e., modes) of the underlying distribution (Derpanis, 2005). Finally, the Mean Shift algorithm performs a gradient ascent procedure, i.e., it searches for these maxima in the empirical probability density function, until convergence (Derpanis, 2005). In detail, The Mean Shift procedure for a given observation \mathbf{x}_i is as follows (Comaniciu & Meer, 2002; Derpanis, 2005):

1. Compute the mean shift vector $\mathbf{m}(\mathbf{x}_i^t)$;
2. Translate density estimation window: $\mathbf{x}_i^{t+1} = \mathbf{x}_i^t + \mathbf{m}(\mathbf{x}_i^t)$;
3. Iterate steps 1. and 2. until convergence.

The mean shift vector is defined as follow [Eq. 3 in Comaniciu and Meer (2002)]:

$$\mathbf{m}(\mathbf{x}_i^t) = \left[\frac{\sum_{i=1}^n \mathbf{x}_i g\left(\left\|\frac{\mathbf{x}-\mathbf{x}_i}{h}\right\|^2\right)}{\sum_{i=1}^n g\left(\left\|\frac{\mathbf{x}-\mathbf{x}_i}{h}\right\|^2\right)} - \mathbf{x} \right] \quad (4.6)$$

where the function $g(x)$ denotes the derivative of the selected kernel estimator and h (i.e., the bandwidth parameter) defines the radius of kernel (Comaniciu & Meer, 2002).

In scikit-learn the *MeanShift()* method perform the mean shift clustering using a flat kernel. Please note that the default scikit-learn parametrization of the mean shift algorithm automatically sets the number of clusters and the optimal h , i.e. the bandwidth. However, h can be manually adjusted using the *bandwidth* parameter.

4.7 K-Means

The clustering by K-Means consists of separating samples into different groups of equal variance. Please note the K-Means algorithm requires the number of clusters to be specified. Mathematically, the K-Means algorithm can be expressed as follow: given an integer k and a set of n data points in \mathbb{R}^d , the goal is to choose k centers to minimize the total squared distance between each point and its closest center, i.e., the inertia (ϕ) (Arthur & Vassilvitskii, 2007):

$$\phi = \sum_{\mathbf{x} \in X} \min_{\mathbf{c} \in C} \|\mathbf{x} - \mathbf{c}\|^2 \quad (4.7)$$

Usually, the K-Means implementation, e.g., in scikit-learn, refers to the solution of the problem proposed by Lloyd (1982). In detail, the solution proposed by Arthur and Vassilvitskii (2007) consists of four steps:

1. Arbitrarily choose an initial k centers $C = \{\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_k, \}$;
2. For each $i \in \{1, \dots, k\}$, set the cluster Y_i to be the set of points in X that are closer to \mathbf{c}_i ;
3. defines new centroids \mathbf{c}_i by averaging all the samples assigned to each previous centroid;
4. Repeat Steps 2 and 3 until C no longer changes.

In scikit-learn, the method *KMeans()* executes the K-Means clustering. Also, The *MiniBatchKMeans()* implements a modification of the KMeans algorithm by utilizing mini-batches to save computation time.

4.8 Spectral Clustering

Spectral Clustering (Von Luxburg, 2007) is a ML technique that combines clustering with dimensionality reduction (Sugiyama, 2015). In detail, Spectral Clustering

utilizes a kernel function to transform samples into a feature space. Then, it applies locality preserving projection to reduce the dimensionality (Fig. 4.1), which has the property that cluster structure of data tends to be preserved (Sugiyama, 2015). Please note that a locality preserving projection in the feature space is equivalent to the Laplacian eigenmap manifold method described in Section 4.3.3 (Sugiyama, 2015). In the practice the Spectral Clustering a low-dimension embedding of the similarity (or affinity) matrix between samples (Von Luxburg, 2007). Finally, Spectral Clustering utilizes a clustering method, e.g., K-Means, to obtain cluster labels (Sugiyama, 2015; Von Luxburg, 2007).

In shikiti-learn the *SpectralClustering()* method applies the Spectral Clustering. To note, *SpectralClustering()* requires the number of clusters to be specified in advance.

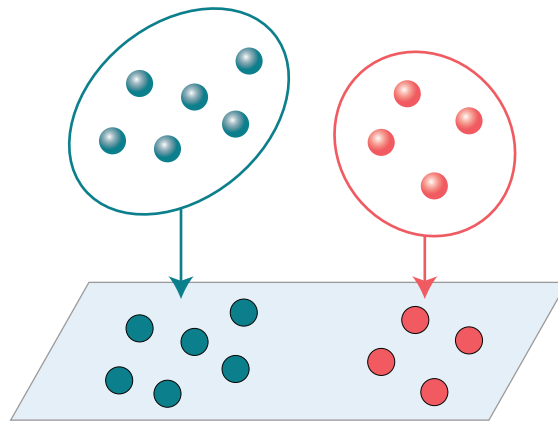


Fig. 4.1 Locality preserving projection. It tries to maintain the cluster structure when reducing the dimensionality of the problem. Modified from Sugiyama (2015)

4.9 Gaussian Mixture Models

Gaussian Mixture Models (GMMs) try reconstructing the probability density function that underlies the investigated data set as generated by a mixture of a finite number of Gaussian distributions with unknown parameters.

Mathematically, consider a d -dimensional, i.e., characterized by d variables or features, sample set $X = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n\}$ of independent and identically distributed, observations (McLachlan & Peel, 2000). Generally speaking, Finite Mixtures Models (FMMs) assume that the observations $\mathbf{x} \in X$ derive by a probability density function described by a mixture of g components (McLachlan & Peel, 2000; Scrucca et al., 2016):

$$f(\mathbf{x}, \psi) = \sum_{i=1}^g \pi_i f_i(\mathbf{x}, \theta_i) \quad (4.8)$$

where g and $\psi = \{\pi_1, \dots, \pi_{g-1}, \theta_1, \dots, \theta_g\}$ are the number of mixture components and the parameters of the model, respectively (Scrucca et al., 2016). Also, $f_i(\mathbf{x}, \theta_i)$ is the i^{th} component density for the sample observation \mathbf{x} and parametrized by the vector θ_i . Finally, $\{\pi_1, \dots, \pi_{g-1}\}$ are the mixing weights (Scrucca et al., 2016).

In many applications, the component densities $f_i(\mathbf{x}, \theta_i)$ are assumed to belong to the same parametric family (McLachlan & Peel, 2000). In some applications, the component densities are taken to be different. Gaussian mixtures models assume $f_i(\mathbf{x}, \theta_i)$ as multivariate normal (McLachlan & Peel, 2000).

The implementation of a finite gaussian mixtures model assumes $f_i(\mathbf{x}, \theta_i)$ as multivariate normal, a fixed G , and consists of estimating the model parameters ψ (McLachlan & Peel, 2000).

In scikit-learn the methods `GaussianMixture()` and `BayesianGaussianMixture` implement the finite Gaussian Mixture model based on the expectation-maximization [EM; Dempster et al. (1977)] and Variational Bayesian Inference (Blei & Jordan, 2006; Hastie et al., 2017), respectively. The Variational Bayesian Inference is similar to the Expectation Maximization. However, it adds a regularization step by integrating information from prior distributions (Blei & Jordan, 2006; Hastie et al., 2017). The aim is to avoid pathological special cases, often found in expectation-maximization solutions (Blei & Jordan, 2006).

Chapter 5

Clustering and Dimensionality Reduction in Petrology

5.1 Unveil the Chemical Record of a Volcanic Eruption

Unsupervised machine learning methods can help us in decoding the chemical record stored in the crystal cargo of a single eruption or multiple volcanic events (Boschetti et al., 2022; Caricchi et al., 2020; Musu et al., 2022). This record often includes the major element's chemical composition, i.e., multivariate compositional data (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005; Boschetti et al., 2022), of different crystal phases, e.g., olivine, clinopyroxene, orthopyroxene, amphibole, plagioclase, garnet, and quartz. Each of these phases provides clues to unravel the complex dynamics of a volcanic plumbing system (Ubide et al., 2021), and its evolution (Costa et al., 2020; Petrelli & Zellmer, 2020).

During the crystallization process (Fig. 5.1), minerals grow and adapt the textural aspect and chemistry to the melt compositions and the thermodynamic conditions of the magmatic system (Ubide et al., 2021). As an example, concentric chemical zones from the core to the rim of a crystal may reflect sequential changes in the magmatic system through time (Fig. 5.1). Moderate to rapid growths at intermediate to high degrees of undercooling ($\Delta T = T_{liquidus} - T_{crystallisation}$) may result in sector zoning in euhedral crystals or skeletal to dendritic textures Fig. 5.1. In addition, diffusive re-equilibration of compositional gradients can further modify the chemical patterns within crystals (Costa et al., 2020; Petrelli & Zellmer, 2020).

At shallow crustal levels (Figure 5.1), pre- and syn-eruptive dynamics cover a complex range of processes, including magma fractionation, recharge, mixing, assimilation, and degassing (Ubide et al., 2021). Interrogating the crystal cargo of an eruption provides us the passport to unravel the complex dynamics occurring in a volcanic plumbing system before and during eruptions (Ubide et al., 2021).

In the present chapter, I will focus on the data set reported by Musu et al. (2022). It consists of clinopyroxene analyses (cpx) erupted by the South-East Crater of Mt. Etna during the sequence of lava fountains in February-March 2021 (Musu et al., 2022).

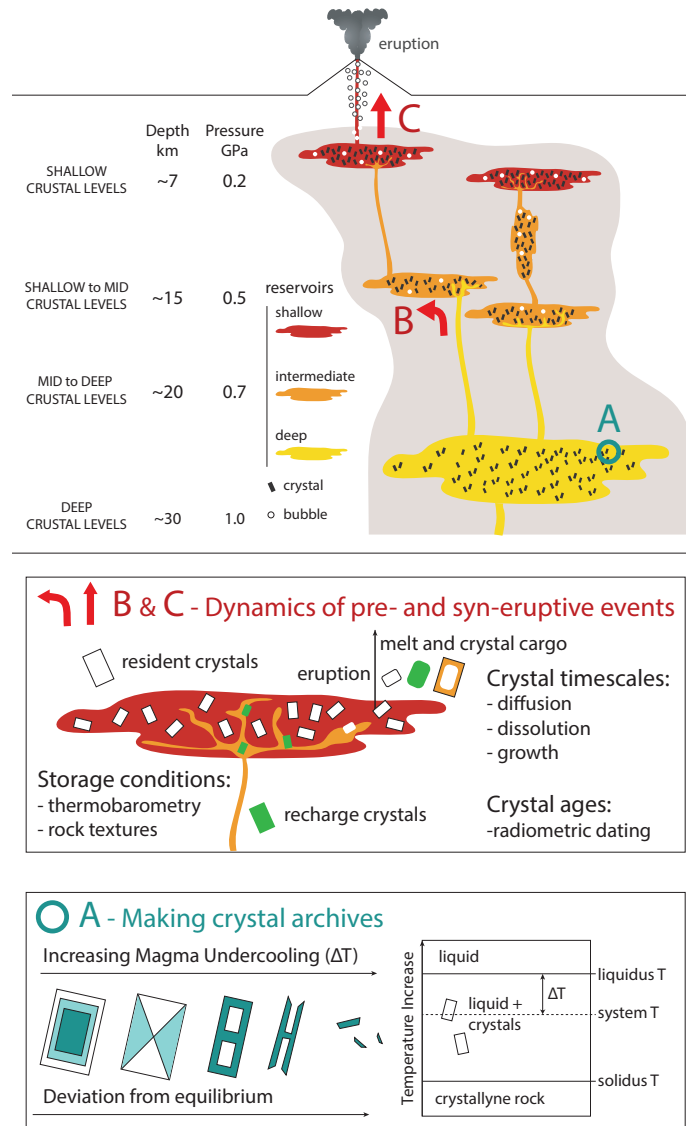


Fig. 5.1 The architecture of a volcanic plumbing system and related pre- and syneruptive dynamics. Modified from Petrelli and Zellmer (2020) and Ubide et al. (2021)

Musu et al. (2022) focused on cpx's analyses since 1) cpx is typically found in mafic to intermediate magmas, 2) it crystallizes over a wide range of temperatures (T) and pressures (P), 3) cpx chemistry is susceptible to changes in response to magma composition, water content, pressure, and temperature variations (Musu et al., 2022). All these properties make cpx a robust thermobarometer (Higgins et al., 2021; Jorgenson et al., 2022; Petrelli et al., 2020; Putirka, 2008) and a fine recorder of the chemical evolution of magmatic systems (Boschetti et al., 2022; Caricchi et al., 2020; Ubide & Kamber, 2018).

5.2 Geological Setting

Mt. Etna is located in southern Italy, eastern Sicily (Italy; Fig. 5.2) and It is the largest active volcano in Europe (Branca & Del Carlo, 2004). Also, Mt. Etna is one of the most active volcanoes in the world (Cappello et al., 2013; Corsaro & Miraglia, 2022).

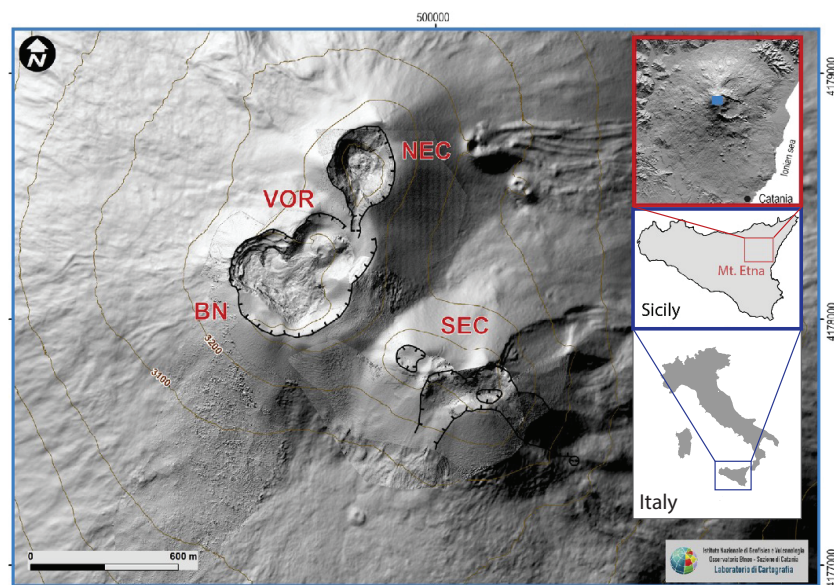


Fig. 5.2 The Etna volcano. Modified from Musu et al. (2022)

Mt. Etna volcano exhibits different eruptive behaviors, from effusive to explosive, including strombolian and violent lava-fountaining occurrences (Branca & Del Carlo, 2004; Corsaro & Miraglia, 2022; Ferlito et al., 2014). Eruptions come from summit craters and fissure vents placed along its flanks (Branca & Del Carlo, 2004; Di Renzo et al., 2019; Musu et al., 2022). The summit area consists of four active vents, i.e.,

Voragine (VOR), Bocca Nuova (BN), North-East Crater (NEC), and South-East Crater (SEC). Among these, SEC is the youngest and the most active (Andronico & Corsaro, 2011; Corsaro & Miraglia, 2022; Di Renzo et al., 2019).

A cyclical eruptive sequence started at the SEC on December 13, 2020. It generated 66 paroxysms up to February 21, 2022 (Andronico & Corsaro, 2011; Bonaccorso et al., 2021; Marchese et al., 2021).

5.3 The investigated data set

The dataset consists of Major Element chemical analyses collected along rim-to-core transects on clinopyroxenes with a point spacing of $2 \mu\text{m}$ (Musu et al., 2022). The total number of analyses is 1250 (Musu et al., 2022). The analyses were collected using a JEOL 8200 Superprobe at the University of Geneva and a JEOL JXA-8530F at the University of Lausanne (Musu et al., 2022). Clinopyroxene samples belong to lapilli that were collected from the deposits of the 16th, 19th, and 28th February and 2nd and 10th March 2021 lava fountains. Fig. 5.3 highlights the tabular structure of the data set.

5.4 Data Pre-Processing

Code listings 5.1 and 5.2 report our data pre-processing strategy, including a final step of data visualization. It consists of a preliminary step of data cleaning, followed by a data transformation in agreement with CoDA, and a ‘robust’ normalization. The resulting CoDA transformed and scaled data have been finally visualized.

Data Cleaning

Code listings 5.1 mainly defines a preliminary data cleaning procedure. In detail, the function named *calc_cations_on_oxygen_basis()* (lines from 4 to 29) calculates the number of cations deriving from a specific chemical analysis based on a fixed number of oxygens on the chemical formula of a specific crystal phase. We are dealing with clinopyroxene analyses, so the base chemical formula contains 6 oxygens and 4 cations (line 36). Also, we define a tolerance of 0.06. This means that we will discard all the analyses that return less than 3.94 and more than 4.06 cations in the formula, respectively. We are mainly discarding bad chemical analyses (e.g., affected by melt or other contaminations, or other issues). If you do not understand this step, please refer to an introductory text on mineralogy for further details (REF). Another test for anhydrous crystal phases is to check for the closure, i.e., verifying that the sum of the oxides is close enough to 100 wt. % (lines 32 and 33).

A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
1 n	point	distance	dist.norm	year	mount	cpv	Na2O	Al2O3	SiO2	MgO	TiO2	FeO	CaO	C2O3	MnO	NiO	Total	comment	sample	eruption	line	sector
2	1	0	0	2021	AMI	cpv1	1.1	10.3	47.9	10.5	2.0	9.0	18.4	0.0	0.1	0.0	99.4	AMI_cpv1_Line001	ET16-TARD	16/02/2021	ln1	no
3	2	3.11	0.11	2021	AMI	cpv1	0.4	6.5	46.7	12.6	2.0	8.7	22.3	0.0	0.1	0.0	99.2	AMI_cpv1_Line001	ET16-TARD	16/02/2021	ln1	no
4	3	6.21	0.22	2021	AMI	cpv1	0.4	6.5	46.7	12.6	2.1	8.9	22.3	0.0	0.1	0.0	99.7	AMI_cpv1_Line004	ET16-TARD	16/02/2021	ln1	no
5	4	9.32	0.33	2021	AMI	cpv1	0.4	6.7	46.6	12.5	2.1	9.0	22.2	0.0	0.1	0.0	99.7	AMI_cpv1_Line004	ET16-TARD	16/02/2021	ln1	no
6	5	12.43	0.44	2021	AMI	cpv1	0.4	7.3	45.9	12.1	2.3	9.2	22.1	0.0	0.1	0.0	99.6	AMI_cpv1_Line005	ET16-TARD	16/02/2021	ln1	no
7	6	15.53	0.56	2021	AMI	cpv1	0.4	6.8	46.6	12.4	2.3	9.0	22.1	0.0	0.1	0.0	99.4	AMI_cpv1_Line006	ET16-TARD	16/02/2021	ln1	no
8	7	18.64	0.67	2021	AMI	cpv1	0.4	6.2	47.2	12.8	1.9	9.0	22.1	0.0	0.1	0.0	99.8	AMI_cpv1_Line007	ET16-TARD	16/02/2021	ln1	no
9	8	21.74	0.78	2021	AMI	cpv1	0.4	5.9	46.7	12.5	1.8	9.3	21.6	0.0	0.1	0.0	99.6	AMI_cpv1_Line008	ET16-TARD	16/02/2021	ln1	no
10	9	24.85	0.89	2021	AMI	cpv1	0.4	3.8	49.8	14.0	1.5	9.2	21.3	0.0	0.1	0.0	100.1	AMI_cpv1_Line010	ET16-TARD	16/02/2021	ln1	no
11	10	27.96	1.00	2021	AMI	cpv1	0.4	3.8	49.8	14.0	1.5	9.2	21.3	0.0	0.1	0.0	99.3	AMI_cpv1_Line010	ET16-TARD	16/02/2021	ln1	no
12	11	31.07	1.11	2021	AMI	cpv1.ln2	0.9	7.2	48.5	12.1	1.7	9.1	19.6	0.0	0.2	0.0	99.3	AMI_cpv1.ln2_Line001	ET16-TARD	16/02/2021	ln2	sector
13	12	34.18	1.22	2021	AMI	cpv1.ln2	0.4	3.7	49.6	13.9	1.3	8.2	21.8	0.0	0.1	0.0	99.1	AMI_cpv1.ln2_Line002	ET16-TARD	16/02/2021	ln2	sector
14	13	37.29	1.33	2021	AMI	cpv1.ln2	0.3	4.4	49.0	13.8	1.5	8.4	21.8	0.0	0.1	0.0	99.4	AMI_cpv1.ln2_Line003	ET16-TARD	16/02/2021	ln2	sector
15	14	40.40	1.44	2021	AMI	cpv1.ln2	0.4	4.3	49.0	13.7	1.4	8.5	21.8	0.0	0.1	0.0	99.2	AMI_cpv1.ln2_Line004	ET16-TARD	16/02/2021	ln2	sector
16	15	43.51	1.55	2021	AMI	cpv1.ln2	0.3	3.8	49.5	14.1	1.3	8.3	21.8	0.0	0.1	0.0	99.3	AMI_cpv1.ln2_Line005	ET16-TARD	16/02/2021	ln2	sector
17	16	46.62	1.66	2021	AMI	cpv1.ln2	0.3	3.8	49.6	14.1	1.3	8.3	21.7	0.0	0.1	0.0	99.2	AMI_cpv1.ln2_Line006	ET16-TARD	16/02/2021	ln2	sector
18	17	49.73	1.77	2021	AMI	cpv1.ln2	0.3	3.9	49.4	14.0	1.3	8.5	21.6	0.0	0.1	0.0	99.1	AMI_cpv1.ln2_Line007	ET16-TARD	16/02/2021	ln2	sector
19	18	52.84	1.88	2021	AMI	cpv1.ln2	0.3	3.7	49.8	14.3	1.3	8.6	21.6	0.0	0.1	0.0	99.5	AMI_cpv1.ln2_Line008	ET16-TARD	16/02/2021	ln2	sector
20	19	55.95	1.99	2021	AMI	cpv1.ln2	0.3	4.3	48.9	13.7	1.5	8.7	21.7	0.0	0.1	0.0	99.3	AMI_cpv1.ln2_Line009	ET16-TARD	16/02/2021	ln2	sector
21	20	59.06	2.10	2021	AMI	cpv1.ln2	0.4	6.5	46.3	12.4	2.2	9.0	22.0	0.0	0.1	0.0	99.3	AMI_cpv1.ln2_Line010	ET16-TARD	16/02/2021	ln2	sector
22	21	62.17	2.21	2021	AMI	cpv2	0.4	6.5	46.3	12.3	2.3	9.1	21.9	0.0	0.1	0.0	98.9	AMI_cpv2_Line001	ET16-TARD	16/02/2021	ln3	no
23	22	65.28	2.32	2021	AMI	cpv2	0.4	6.6	46.2	12.3	2.3	9.0	22.0	0.0	0.1	0.0	98.8	AMI_cpv2_Line003	ET16-TARD	16/02/2021	ln3	no
24	23	68.39	2.43	2021	AMI	cpv2	0.4	6.6	46.2	12.3	2.3	9.0	22.0	0.0	0.1	0.0	98.8	AMI_cpv2_Line003	ET16-TARD	16/02/2021	ln3	no
25	24	71.50	2.54	2021	AMI	cpv2	0.4	6.7	46.2	12.3	2.3	9.0	22.0	0.0	0.1	0.0	99.0	AMI_cpv2_Line004	ET16-TARD	16/02/2021	ln3	no
26	25	74.61	2.65	2021	AMI	cpv2	0.4	6.8	46.2	12.4	2.1	8.8	22.1	0.0	0.1	0.0	98.9	AMI_cpv2_Line005	ET16-TARD	16/02/2021	ln3	no
27	26	77.72	2.76	2021	AMI	cpv2	0.3	6.7	46.6	12.7	2.0	8.5	22.2	0.0	0.1	0.0	99.2	AMI_cpv2_Line006	ET16-TARD	16/02/2021	ln3	no
28	27	80.83	2.87	2021	AMI	cpv2	0.4	7.0	46.3	12.5	2.2	8.8	22.1	0.0	0.1	0.0	99.4	AMI_cpv2_Line007	ET16-TARD	16/02/2021	ln3	no
29	28	83.94	2.98	2021	AMI	cpv2	0.4	6.5	46.7	12.5	2.0	8.6	22.2	0.0	0.1	0.0	99.0	AMI_cpv2_Line008	ET16-TARD	16/02/2021	ln3	no
30	29	87.05	3.09	2021	AMI	cpv2	0.4	6.8	46.1	12.1	2.3	9.2	22.0	0.0	0.1	0.0	99.0	AMI_cpv2_Line009	ET16-TARD	16/02/2021	ln3	no
31	30	90.16	3.20	2021	AMI	cpv2	0.3	6.9	46.2	12.3	2.2	8.9	22.1	0.0	0.1	0.0	99.1	AMI_cpv2_Line010	ET16-TARD	16/02/2021	ln3	no
32	31	93.27	3.31	2021	AMI	cpv2	0.4	6.7	46.5	12.5	2.0	8.7	22.3	0.0	0.1	0.0	99.3	AMI_cpv2_Line011	ET16-TARD	16/02/2021	ln3	no
33	32	96.38	3.42	2021	AMI	cpv2	0.4	6.6	46.5	12.3	2.3	9.1	22.0	0.0	0.1	0.1	99.3	AMI_cpv2_Line012	ET16-TARD	16/02/2021	ln3	no
34	33	99.49	3.53	2021	AMI	cpv2	0.4	6.3	46.8	12.5	2.1	8.9	22.0	0.0	0.1	0.0	99.2	AMI_cpv2_Line013	ET16-TARD	16/02/2021	ln3	no
35	34	102.60	3.64	2021	AMI	cpv2	0.4	6.4	46.7	12.5	2.2	9.0	22.0	0.0	0.1	0.0	99.3	AMI_cpv2_Line014	ET16-TARD	16/02/2021	ln3	no
36	35	105.71	3.75	2021	AMI	cpv2	0.4	6.8	46.5	12.5	2.1	8.7	22.3	0.0	0.1	0.0	99.3	AMI_cpv2_Line015	ET16-TARD	16/02/2021	ln3	no
37	36	108.82	3.86	2021	AMI	cpv2	0.4	6.7	46.6	12.7	1.9	8.4	22.3	0.0	0.1	0.0	99.2	AMI_cpv2_Line016	ET16-TARD	16/02/2021	ln3	no

Fig. 5.3 The structure of the investigated data set.

Now moving to the code listing 5.1 it starts with isolating from the data set, the only chemical elements of interest for the successive statistical analyses (i.e., SiO_2 , TiO_2 , Al_2O_3 , FeO , MgO , CaO , and Na_2O ; lines 7 to 10). The last stem of data cleaning consist of removing all the rows containing data that are below or exceed the 0.1 and 99.9 percentile, respectively (lines 12-14).

```

1 import numpy as np
2 import pandas as pd
3
4 def calc_cations_on_oxygen_basis(myData0, my_ph, my_el, n_ox):
5     Weights = {
6         'SiO2': [60.0843,1.0,2.0], 'TiO2': [79.8788,1.0,2.0],
7         'Al2O3': [101.961,2.0,3.0], 'FeO': [71.8464,1.0,1.0],
8         'MgO': [40.3044,1.0,1.0], 'MnO': [70.9375,1.0,1.0],
9         'CaO': [56.0774,1.0,1.0], 'Na2O': [61.9789,2.0,1.0],
10        'K2O': [94.196,2.0,1.0], 'Cr2O3': [151.9982,2.0,3.0],
11        'P2O5': [141.937,2.0,5.0], 'H2O': [18.01388,2.0,1.0]}
12    myData = myData0.copy()
13    myData = myData.add_prefix(my_ph + '_')
14    for el in my_el: # Cation mole proportions
15        myData[el + '_cat_mol_prop'] = myData[my_ph +
16            '_' + el] * Weights[el][1] / Weights[el][0]
17    for el in my_el: # Oxygen mole proportions
18        myData[el + '_oxy_mol_prop'] = myData[my_ph +
19            '_' + el] * Weights[el][2] / Weights[el][0]
20    totals = np.zeros(len(myData.index)) # Ox mole prop tot
21    for el in my_el:
22        totals += myData[el + '_oxy_mol_prop']
23    myData['tot_oxy_prop'] = totals
24    totals = np.zeros(len(myData.index)) # totcations
25    for el in my_el:
26        myData[el + '_num_cat'] = n_ox * myData[el +
27            '_cat_mol_prop'] / myData['tot_oxy_prop']
28        totals += myData[el + '_num_cat']
29    return totals
30
31 my_dataset = pd.read_table('ETN21_cpx_all.txt')
32 my_dataset = my_dataset[(my_dataset.Total>98) &
33     (my_dataset.Total<102)]
34 Elements = {'cpx': ['SiO2', 'TiO2', 'Al2O3',
35     'FeO', 'MgO', 'MnO', 'CaO', 'Na2O', 'Cr2O3']}
36 Cat_Ox_Tolerance = {'cpx': [4,6,0.06]}
37 my_dataset['Tot_cations'] = calc_cations_on_oxygen_basis(
38     myData0 = my_dataset,
39     my_ph = 'cpx',
40     my_el = Elements['cpx'],
41     n_ox = Cat_Ox_Tolerance['cpx'][1])
42
43 my_dataset = my_dataset[(
44     my_dataset['Tot_cations'] < Cat_Ox_Tolerance['cpx'][0] +
45     Cat_Ox_Tolerance['cpx'][2])&(
46     my_dataset['Tot_cations'] > Cat_Ox_Tolerance['cpx'][0] -

```



```
47 Cat_Ox_Tolerance['cpx'][2]]
```

Listing 5.1 Data pre-processing. Initial step.

Compositional Data Analysis (CoDA)

The study of a geochemical data set falls in the field of Compositional Data Analysis (CoDA). In this context, the oxides are expressed as a percentage, so their nominal sum is 100%, defining a ‘closed’, or ‘compositional data set’ (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005). Conducting statistical analysis directly on ‘closed data sets’ can lead to issues (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005) as some statistical approaches require the data to be normally distributed and not constrained to a constant total value (Boschetti et al., 2022).

```
1 from skbio.stats.composition import ilr
2 from sklearn.preprocessing import RobustScaler
3 import matplotlib.pyplot as plt
4 import seaborn as sns
5
6 elms_for_clustering = {'cpx': ['SiO2', 'TiO2',
7                               'Al2O3', 'FeO', 'MgO', 'CaO', 'Na2O']}
8
9 my_dataset = my_dataset[elms_for_clustering['cpx']]
10
11 my_dataset = my_dataset[~((
12     my_dataset < my_dataset.quantile(0.001)) |
13     (my_dataset > my_dataset.quantile(0.999))).any(axis=1)]
14
15 my_dataset_ilr = ilr(my_dataset)
16
17 transformer = RobustScaler(
18     quantile_range=(25.0, 75.0)).fit(my_dataset_ilr)
19
20 my_dataset_ilr_scaled = transformer.transform(my_dataset_ilr)
21
22 fig = plt.figure(figsize=(8,8))
23
24 for i in range(0,6):
25     ax1 = fig.add_subplot(3, 2, i+1)
26     sns.kdeplot(my_dataset_ilr_scaled[:, i], fill=True,
27               color='k', facecolor='#c7ddf4', ax = ax1)
28     ax1.set_xlabel('scaled ilr_' + str(i+1))
29 fig.align_ylabels()
30 fig.tight_layout()
```

Listing 5.2 Compositional Data Analysis (CoDA)

Performing multivariate statistical analysis ‘compositional data sets’ directly is not formally correct and can lead to biases in the results or other unwanted issues (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005). Different data transformations have been proposed to allow the application of standard and advanced statistical methods to compositional data sets. Examples are the additive log-ratio (*alr*), the centered log-ratio (*clr*), and the isometric log-ratio (*ilr*) transformation, respectively (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005). The *ilr* has been shown to work effectively with geochemical data (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005) mapping a compositional data set into a real Euclidean space (Aitchison, 1982, 1984; Aitchison & Egozcue, 2005). We briefly introduced CoDA analysis in section 3.3, also reporting the related equations.

At line 16 of code listing 5.1, we apply the *ilr* transformation to our data, then scaling in agreement with the median and the inter-quartile range (lines 18-21), i.e., applying the *RobustScaler*(). Then we visualize the resulting features (Fig. 5.4).

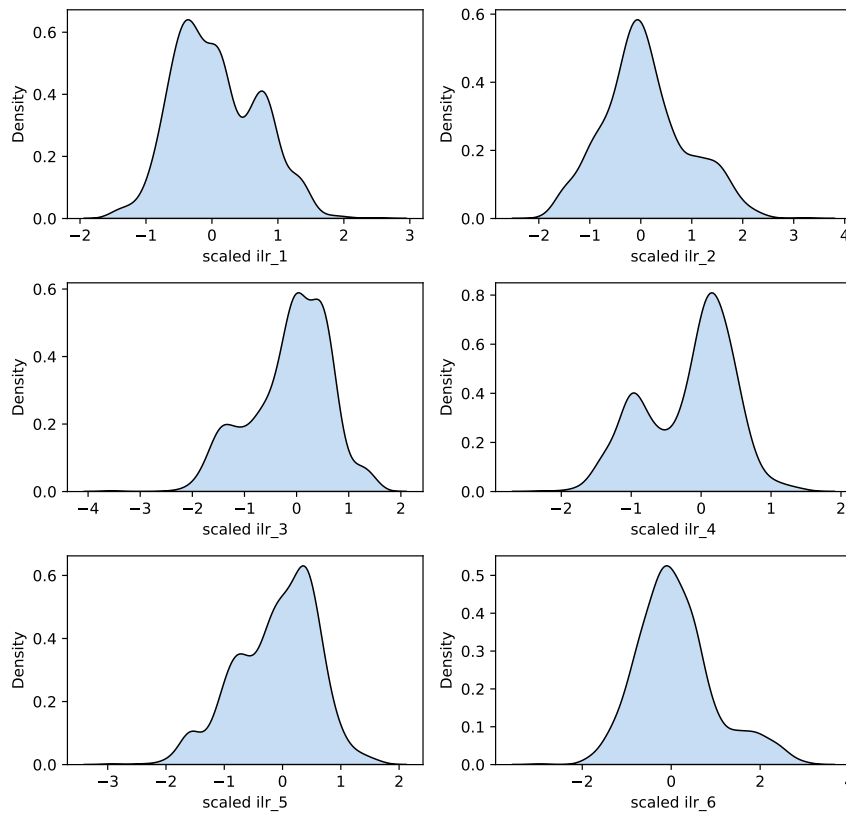


Fig. 5.4 Inspecting *ilr* transformed data.

5.5 Clustering analyses

Code listing 5.3 shows how to develop a hierarchical clustering dendrogram in Python (Fig. 5.5). A dendrogram is a tree diagram used to report the result of a hierarchical clustering estimation.

```

1 import numpy as np
2 from sklearn.cluster import AgglomerativeClustering
3 from scipy.cluster.hierarchy import dendrogram,
   set_link_color_palette
4
5 def plot_dendrogram(model, **kwargs):
6
7     counts = np.zeros(model.children_.shape[0])
8     n_samples = len(model.labels_)
9     for i, merge in enumerate(model.children_):
10         current_count = 0
11         for child_idx in merge:
12             if child_idx < n_samples:
13                 current_count += 1
14             else:
15                 current_count += counts[child_idx - n_samples]
16         counts[i] = current_count
17
18     linkage_matrix = np.column_stack([model.children_,
19                                     model.distances_,
20                                     counts]).astype(float)
21
22     dendrogram(linkage_matrix, **kwargs)
23
24 model = AgglomerativeClustering(linkage='ward',
25                                 affinity='euclidean',
26                                 distance_threshold = 0,
27                                 n_clusters=None)
28
29 model.fit(my_dataset_ilr_scaled)
30
31 fig, ax = plt.subplots(figsize = (10,6))
32 ax.set_title('Hierarchical clustering dendrogram')
33
34 plot_dendrogram(model, truncate_mode='level', p=5,
35                 color_threshold=0,
36                 above_threshold_color='black')
37
38 ax.set_xlabel('Number of points in node')
39 ax.set_ylabel('Height')

```

Listing 5.3 Developing a hierarchical clustering dendrogram in Python

The dendrogram can be oriented both vertically (e.g., Fig. 5.5) and horizontally. The orientation can be easily changed in the `dendrogram()` function by the orientation parameter (i.e., 'top', 'bottom', 'left', or 'right').

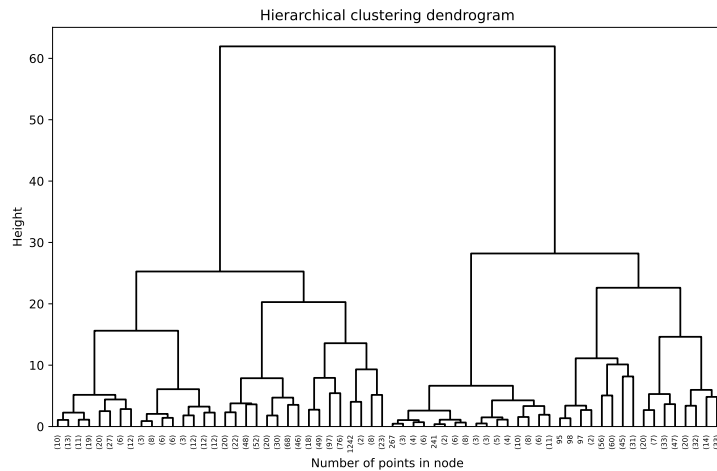


Fig. 5.5 A dendrogram resulting from code listing 5.3.

```

1 th = 16.5
2 fig, ax = plt.subplots(figsize = (10,6))
3 ax.set_title("Hierarchical clustering dendrogram")
4 set_link_color_palette(['#000000', '#C82127', '#0A3A54',
5                         '#0F7F8B', '#BFD7EA', '#F15C61', '#E8BFE7'])
6
7 plot_dendrogram(model, truncate_mode='level', p=5,
8                 color_threshold=th,
9                 above_threshold_color='grey')
10
11 plt.axhline(y = th, color = "k", linestyle = "--", lw=1)
12 ax.set_xlabel("Number of points in node")
13
14 fig, ax = plt.subplots(figsize = (10,6))
15 ax.set_title("Hierarchical clustering dendrogram")
16 ax.set_ylabel('Height')
17
18 plot_dendrogram(model, truncate_mode='lastp', p=6,
19                 color_threshold=0,
20                 above_threshold_color='k')
21
22 ax.set_xlabel("Number of points in node")

```

Listing 5.4 Refining the dendrogram.

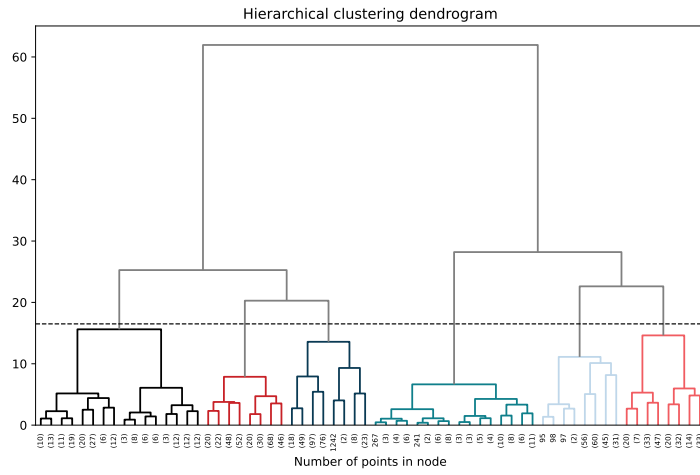


Fig. 5.6 The dendrogram resulting from code listing 5.4.

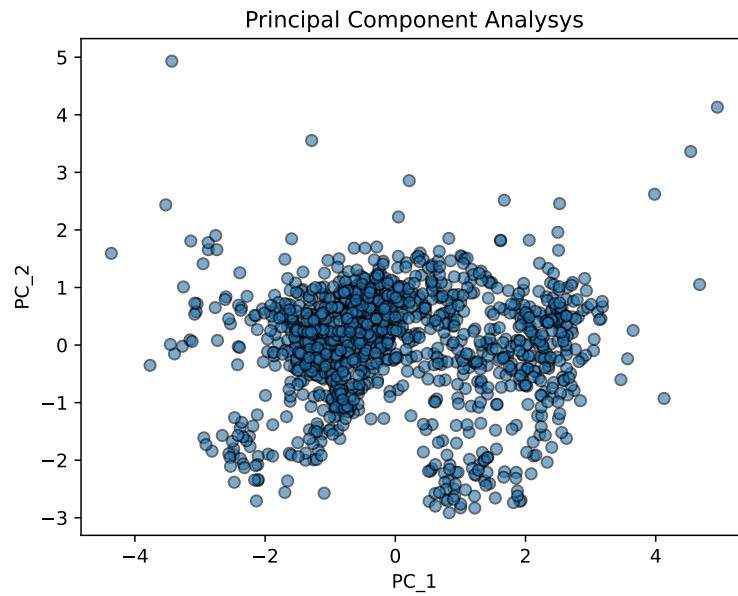


Fig. 5.7 Scatter diagram of the the first two principal components.

When oriented vertically, the vertical scale reports the measure of the distance or the similarity among clusters. If we draw a horizontal line, the number of leaves we intercept (e.g., Fig. 5.6) defines the number of clusters at that specific height. Increasing the height, the number of clusters reduce. In our specific case, the fixing of a threshold at 16.5 defines 6 clusters (code listing 5.4 and Fig. 5.6).

```

1 from sklearn.cluster import AgglomerativeClustering
2 from sklearn.decomposition import PCA
3 import numpy as np
4 import matplotlib.pyplot as plt
5
6 my_colors = {0: '#0A3A54',
7             1: '#E08B48',
8             2: '#BFBFBF',
9             3: '#BD22C6',
10            4: '#FD787B',
11            5: '#67CF62' }
12 #PCA
13 model_PCA = PCA()
14 model_PCA.fit(my_dataset_ilr_scaled)
15 my_PCA = model_PCA.transform(my_dataset_ilr_scaled)
16
17 fig, ax = plt.subplots()
18 ax.scatter(my_PCA[:,0], my_PCA[:,1],
19           alpha=0.6,
20           edgecolors='k')
21 ax.set_title('Principal Component Analysys')
22 ax.set_xlabel('PC_1')
23 ax.set_ylabel('PC_2')
```

Listing 5.5 Plotting the first two principal components

5.6 Dimensionality Reduction

The *ilr* transformed data set consists of 6 features (Fig. 5.4). In the attempt of visualizing the structure of our data, I performed the Principal Component Analysis (PCA; section 4.2). It consists of a linear dimensionality reduction that uses a Singular Value Decomposition of the data set to project it to a lower dimensional space.

Code listing 5.5 shows how to apply the Principal Component Analysis to our data set. Also, it provides us with a binary diagram (Fig. 5.7) reporting the two first principal components.

Visualizing the 6 clusters highlighted in Fig. 5.6 could be a benefit. Code listing 5.6 shows how to do that (Fig. 5.8). Also, Code listing 5.6 shows how to apply and visualize (Fig. 5.9) the KMeans clustering (section 4.7).

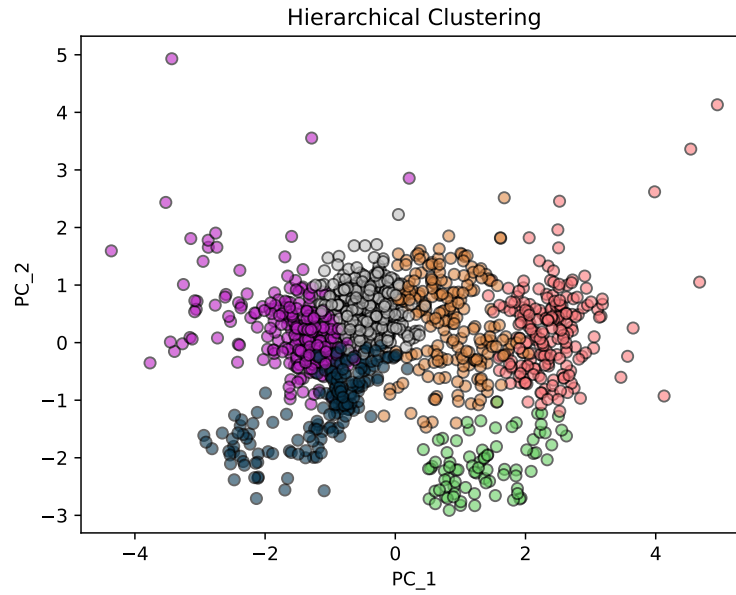


Fig. 5.8 Combining the Principal component analysis with the Hierarchical clustering

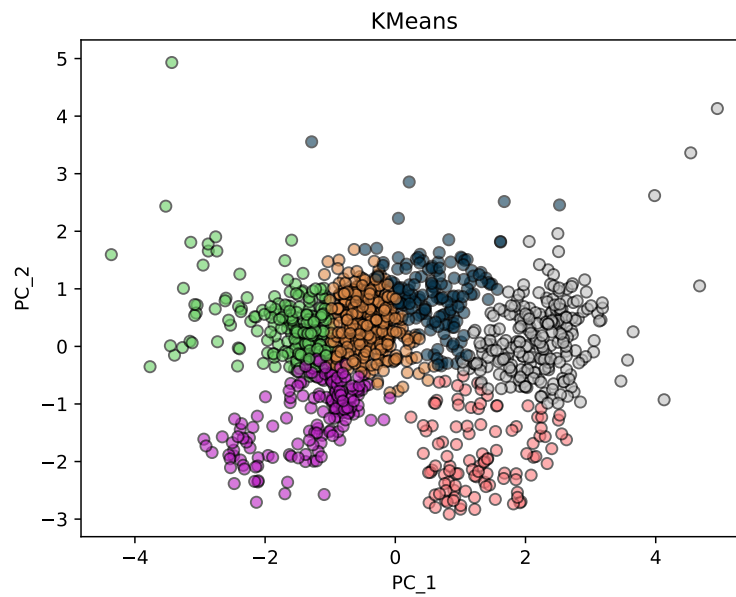


Fig. 5.9 Combining the Principal component analysis with the KMeans clustering

```
1 #AgglomerativeClustering
2 model_AC = AgglomerativeClustering(linkage='ward',
3                                     affinity='euclidean',
4                                     n_clusters=6)
5 my_AC = model_AC.fit(my_dataset_ilr_scaled)
6
7 fig, ax = plt.subplots()
8 label_to_color = [my_colors[i] for i in my_AC.labels_]
9 ax.scatter(my_PCA[:,0], my_PCA[:,1],
10           c=label_to_color, alpha=0.6,
11           edgecolors='k')
12 ax.set_title('Hierarchical Clustering')
13 ax.set_xlabel('PC_1')
14 ax.set_ylabel('PC_2')
15 my_dataset['cluster_HC'] = my_AC.labels_
16
17 #KMeans
18 from sklearn.cluster import KMeans
19 myKM = KMeans(n_clusters=6).fit(my_dataset_ilr_scaled)
20
21 fig, ax = plt.subplots()
22 label_to_color = [my_colors[i] for i in myKM.labels_]
23 ax.scatter(my_PCA[:,0], my_PCA[:,1],
24           c=label_to_color, alpha=0.6,
25           edgecolors='k')
26 ax.set_title('KMeans')
27 ax.set_xlabel('PC_1')
28 ax.set_ylabel('PC_2')
29 my_dataset['cluster_KM'] = myKM.labels_
```

Listing 5.6 Combining the Principal component analysis with the Hierarchical and KMeans clustering methods

Chapter 6

Clustering of Multi-Spectral Data

6.1 Spectral Data from Earth-Observing Satellites

Earth-Observing satellite missions like Sentinel¹ and Landsat² provide us with multi-spectral, hyperspectral and panchromatic data. Going into more details, the Sentinel Earth-observing satellite missions are part of the Copernicus program, developed by the European Space Agency (ESA)³, whereas the Landsat Program is jointly managed by NASA and the U.S. Geological Survey².

Spectral images are two-dimensional representations of surface reflectance or radiation in different bands of the electromagnetic spectrum. Multi-spectral and hyper-spectral data are acquired by multiple sensors, operating at wide and narrow (sometimes quasi-continuous) wavelength ranges, respectively. Differently, panchromatic images derive from detectors covering the entire visible range.

Multi-spectral, hyper-spectral, and panchromatic data can be combined and modulated to produce new indexes (e.g., the Generalized Difference Vegetation Index or the Normalized Difference Snow Index), able to highlight specific phenomena and improve data interpretation (REF).

As an example, the SENTINEL-2 Multi-spectral Instrument (MSI) works on 13 spectral bands. Four bands (i.e., B2, B3, B4, and B8) have a spatial resolution of 10 meters, six bands (i.e., B5, B6, B7, B8a, B11, and B12) of 20 meters, and three bands (i.e., B1, B9, and B10) of 60 meters (Fig. 6.1).

¹ <https://sentinels.copernicus.eu>

² <https://landsat.gsfc.nasa.gov>

³ <https://www.esa.int>

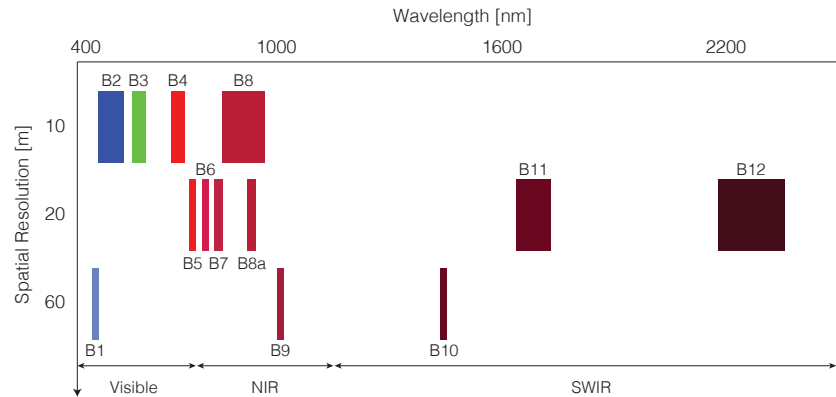


Fig. 6.1 Spectral bands of Sentinel2 satellites. Modified from Majidi Nezhad et al. (2021)

6.2 Import Multi-spectral Data in Python

There are a plethora of access points where download multispectral data. Examples are the USGS Earth Explorer⁴, the Copernicus Open Access Hub⁵, and Theia⁶.

As an example, Figure 6.2 represents the recombination of the B4, B3, and B2 bands as RGB (i.e., Red, Green, and Blue) image of a SENTINEL2 acquisition downloaded from the Theia portal. The location of the picture is the southern New South Wales (Australia)⁷. Each side of the square picture measures about 110 km.

Figure 6.3 reports the data structure of a SENTINEL2 repository downloaded from Theia. In detail, the repository follows the MUSCATE⁸ nomenclature and contains: a metadata file, a quick-look file, many Geo-Tiff image files, and two sub-repositories, i.e., MASKS and DATA, containing supplementary data. The naming enable us to uniquely identify each product and it consists of many tags starting with a platform identification (i.e., SENTINEL2B) followed by the date of acquisition in the format YYYYMMDD-HHmmSS-sss (i.e., 20210621-001635-722), with YYYY year, MM month, DD day, HH hour over 24 hours, mm minuets, SS seconds and sss milliseconds. The subsequent tags refer to product level (i.e., L2A), geographical zone (i.e., T55HDB.C), and product version (i.e., V2-2), respectively. The letter L, a number and a letter characterizes different product levels with the exception of the level L0, i.e., compressed raw data, that is not followed by any letter. Levels L1A, L1B, and L2A correspond to uncompressed raw data, radiometrically corrected radiance data, and orthorectified Bottom-Of-Atmosphere (BOA) reflectance,

⁴

⁵ <https://scihub.copernicus.eu>

⁶ <https://catalogue.theia-land.fr>

⁷ https://bit.ly/ml_geart

⁸ <https://www.theia-land.fr/en/product/sentinel-2-surface-reflectance/>



Fig. 6.2 RGB composite image where the B4, B3, and B2 bands regulate the intensities of the Red, Green, and Blue channels, respectively.

respectively⁹. Spectral Geo-tiff files also report an additional tag, i.e., SRE and FRE, corresponding to the image in ground reflectance without the correction of slope effects and image in ground reflectance with the correction of slope effects, respectively. We will work on FRE data.

To import SENTINEL2 multispectral data, I suggest using Rasterio¹⁰. Rasterio provides us a Python API based on Numpy and GeoJSON (i.e., an open standard format designed for representing geographical features, along with their non-spatial attributes) to read, write and manage Geo-Tiff data.

If you followed the instruction reported in Chapter 2, your Python machine learning environments named *env_ml* and *env_ml.intel* contain Rasterio already. With Rasterio, opening Geo-Tiff files is straightforward (code listing 6.1).

⁹ <https://sentinels.copernicus.eu/web/sentinel/technical-guides/sentinel-2-msi>

¹⁰ <https://rasterio.readthedocs.io/>

Name	Date Modified	Size	Kind
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2	Yesterday at 16:23	--	Folder
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B12.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B11.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B8A.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B8.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B7.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B6.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B5.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B4.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B3.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_SRE_B2.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_QK1_ALL.jpg	Yesterday at 16:23	411 KB	JPEG image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_MTD_ALL.xml	21 Jun 2021 at 22:05	494 KB	XML File
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B12.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B11.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B8A.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B8.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B7.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B6.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B5.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B4.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B3.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_B2.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_ATB_R2.tif	21 Jun 2021 at 22:05	60.3 MB	TIFF image
SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_ATB_R1.tif	21 Jun 2021 at 22:05	241.2 MB	TIFF image
MASKS	6 Jan 2022 at 20:09	--	Folder
DATA	Today at 10:14	--	Folder

Fig. 6.3 Sentinel2 data structure.

```

1 import rasterio
2 import numpy as np
3
4 imagePath = 'SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2/
5             SENTINEL2B_20210621-001635-722_L2A_T55HDB_C_V2-2_FRE_'
6
7 bands_to_be_imported = ['B2', 'B3', 'B4', 'B8']
8
9 bands_dict = {}
10 for band in bands_to_be_imported:
11     with rasterio.open(imagePath+ band + '.tif', 'r',
12                        driver='GTiff') as my_band:
13         bands_dict[band] = my_band.read(1)

```

Listing 6.1 Importing Sentinel2 data in Python using rasterio.

The code listing 6.1 creates a dictionary of NumPy arrays, i.e., *bands_dict*, containing spectral information for the B2, B3, B4, and B8, corresponding to the blue, green, red and near-infrared band, respectively. In the code listing 6.1, we limit the import to 4 bands, all acquired at the same spatial resolution (i.e. 10 m). However, the script can be easily extended to import a larger number of bands. It is straightforward that the combination of data coming from bands acquired at different resolutions (e.g., B2 at 10 m and B6 at 20 m) requires a preliminary re-sampling to a common resolution.

Combining the data of *bands_dict* dictionary, many different representations can be achieved. As an example, Sovdat et al. (2019) reports how to perform the “natural color” representation of Sentinel-2 data.

The achievement of a perfectly balanced image with natural colors is beyond the scope of the present book, therefore, we limit to combine the bands B2, B3, and B4 which roughly correspond to blue, green, and red color perceived by our eyes, respectively.

In detail, a bright, possibly overly saturated (Sovdat et al., 2019), image (i.e., *r-g-b*) can be easily derived and plotted (code listing 6.2; Fig. 6.2) starting from the *bands_dict* dictionary after a contrast stretching (lines from 11 to 17), and values scaling in the interval [0,1]. This is the so called “true color” representation. Sometimes, the bands B3 (i.e., red) and B4 (i.e., green) are combined with B8 (i.e., near-infrared) to achieve a “false color” representation. False color composite images are often used to highlight plant density and health (e.g., Fig. 6.4). Code listing 6.3 reports how to perform a “false color” representation (i.e., *nir_r-g*) of Sentinel2 data.

```

1 import numpy as np
2 from skimage import exposure, io
3 from skimage.transform import resize
4 import matplotlib.pyplot as plt
5
6 r_g_b = np.dstack([bands_dict['B4'],
7                   bands_dict['B3'],
8                   bands_dict['B2']])
9
10 # contrast stretching and rescaling between [0,1]
11 p2, p98 = np.percentile(r_g_b, (2,98))
12 r_g_b = exposure.rescale_intensity(r_g_b, in_range=(p2, p98))
13 r_g_b = r_g_b / r_g_b.max()
14
15 fig, ax = plt.subplots(figsize=(8, 8))
16 ax.imshow(r_g_b)
17 ax.axis('off')
```

Listing 6.2 Plotting a RGB image using the B4, B3, and B2 bands.

```

1 import numpy as np
2 from skimage import exposure, io
3 from skimage.transform import resize
4 import matplotlib.pyplot as plt
5
6 nir_r_g = np.dstack([bands_dict['B8'],
7                   bands_dict['B4'],
8                   bands_dict['B3']])
9
10 # contrast stretching and rescaling between [0,1]
11 p2, p98 = np.percentile(nir_r_g, (2,98))
12 nir_r_g = exposure.rescale_intensity(nir_r_g, in_range=(p2, p98))
13
14 fig, ax = plt.subplots(figsize=(8, 8))
15 ax.imshow(nir_r_g)
```

```
16 ax.axis('off')
```

Listing 6.3 Plotting a false-color RGB composite image using the B8, B4, and B3 bands.

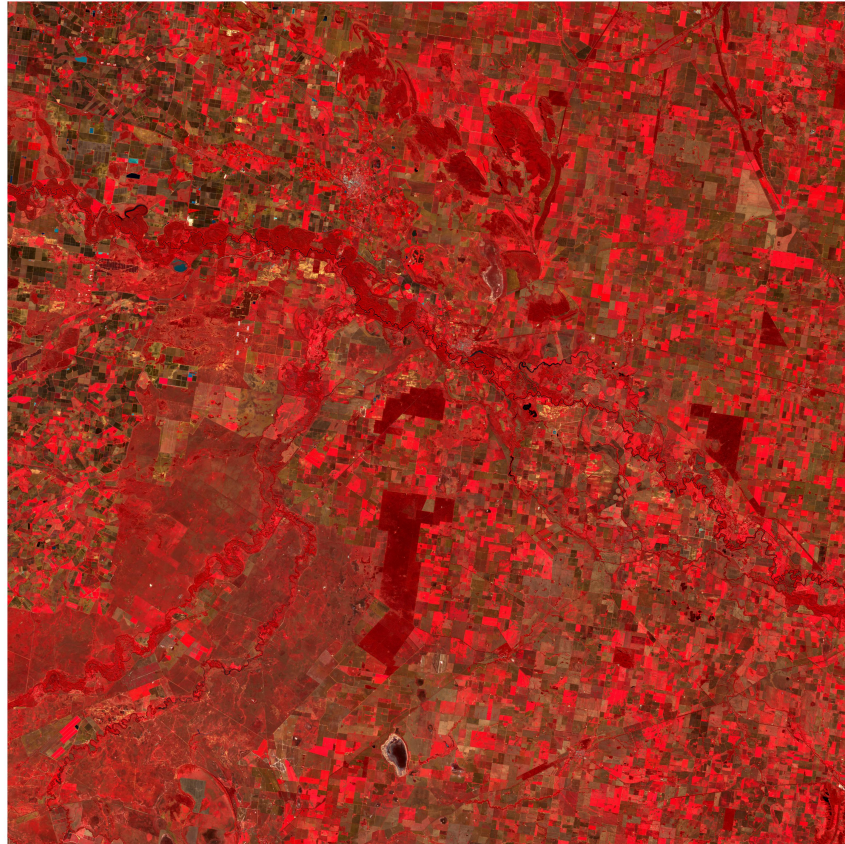


Fig. 6.4 Image resulting by code listing 6.3.

6.3 Descriptive Statistics

One of the first steps of any ML workflow consists of descriptive statistics. For the case of our Sentinel2 data set, code listing 6.5 describes how to perform descriptive statistics with visualization of a four bands (i.e., B2, B3, B4, and B5) array derived from Geo-Tiff data. In detail, at line 5, we create a (10980, 10980, 4) array (i.e.,

`my_array_2d` characterized by a width, height, and dept of 10980, 10980, and 4, respectively) from the dictionary created in the code listing 6.1. In the next step, i.e., line 10, we create a new array (`my_array_1d`) reshaping `my_array_2d` from (10980, 10980, 4) to (120560400, 4). This is the typical aspect of an array that is ready for ML processing in scikit-learn. Converting `my_array_1d` to a pandas DataFrame (i.e., `my_array_1d_pandas`) facilitates the visualization (i.e., lines 18-46) and getting the most basic descriptive statistics (i.e., listing 6.4). Looking at code listing 6.4, we can derive basic information about the central position, dispersion and shape of our input features. To note, Figure 6.5 shows that 99% of reflectance data for B2, B3, B4, and B8 are in the range of 0.015-0.42. However, maximum values are always above 1, i.e., the upper theoretical bound for reflectance data. Outliers at reflectance values above 1 could be the result of specular effects on surface or clouds (Schaepman-Strub et al., 2006).

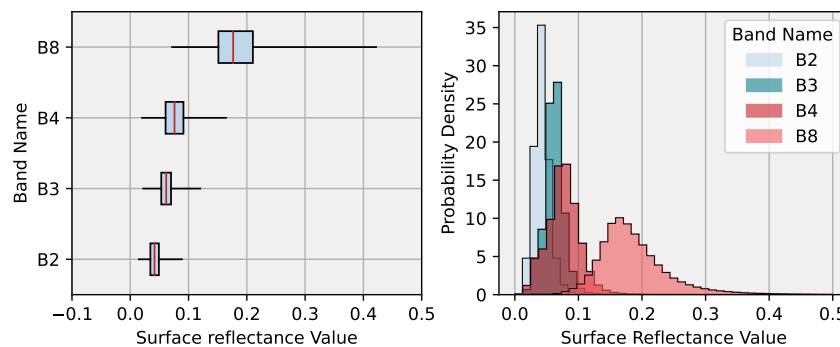


Fig. 6.5 Descriptive statistics.

```
In [1]: my_array_1d_pandas.describe().applymap("{0:.3f}".format)
Out [1]:
```

	B2	B3	B4	B8
count	120560400.000	120560400.000	120560400.000	120560400.000
mean	0.042	0.062	0.076	0.186
std	0.013	0.016	0.026	0.056
min	0.000	0.000	0.000	0.000
25%	0.035	0.053	0.061	0.151
50%	0.042	0.062	0.076	0.177
75%	0.049	0.070	0.091	0.210
max	1.443	1.304	1.277	1.201

Listing 6.4 Importing data from an Excel file into Python.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 import pandas as pd
```

```

4
5 my_array_2d = np.dstack([bands_dict['B2'],
6                           bands_dict['B3'],
7                           bands_dict['B4'],
8                           bands_dict['B8']])
9
10 my_array_1d =my_array_2d[:, :, :4].reshape(
11     (my_array_2d.shape[0] * my_array_2d.shape[1],
12     my_array_2d.shape[2]))
13
14 my_array_1d_pandas = pd.DataFrame(my_array_1d,
15     columns=['B2', 'B3', 'B4', 'B8'])
16
17
18 fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(7,3))
19 my_medianprops = dict(color='#C82127', linewidth = 1)
20 my_boxprops = dict(facecolor='#BFD7EA', edgecolor='#000000')
21 ax1.boxplot(my_array_1d_pandas, vert=False, whis=(0.5, 99.5),
22             showfliers=False, labels=my_array_1d_pandas.columns,
23             patch_artist=True, showcaps=False,
24             medianprops=my_medianprops, boxprops=my_boxprops)
25 ax1.set_xlim(-0.1,0.5)
26 ax1.set_xlabel('Surface reflectance Value')
27 ax1.set_ylabel('Band Name')
28 ax1.grid()
29 ax1.set_facecolor((0.94, 0.94, 0.94))
30
31 colors=['#BFD7EA', '#0F7F8B', '#C82127', '#F15C61']
32 for band, color in zip(my_array_1d_pandas.columns, colors):
33     ax2.hist(my_array_1d_pandas[band], density=True,
34             bins='doane', range=(0,0.5), histtype='step',
35             linewidth=1, fill=True, color=color, alpha=0.6,
36             label=band)
37     ax2.hist(my_array_1d_pandas[band], density=True,
38             bins='doane', range=(0,0.5), histtype='step',
39             linewidth=0.5, fill=False, color='k')
40 ax2.legend(title='Band Name')
41 ax2.set_xlabel('Surface Reflectance Value')
42 ax2.set_ylabel('Probability Density')
43 ax2.xaxis.grid()
44 ax2.set_facecolor((0.94, 0.94, 0.94))
45 plt.tight_layout()
46 plt.savefig('descr_stat_sat.pdf')

```

Listing 6.5 Descriptive statistics using pandas *describe()*.

The presence of large outliers could affect the results of your ML model if not addressed correctly. As a consequence, I suggest defining a strategy to remove the outliers based on robust statistics (e.g., Petrelli (2021)), or to applying a robust scaler.

6.4 Pre-processing and Clustering

In the following paragraphs, I report a simplified workflow to perform the clustering of our Sentinel2 data. As input features, I used the *my_array_1d*, i.e., reflectance data coming from B2, B3, B4, and B8. Please note that many different strategies are reported in the literature for the input feature selection. Examples are the use of band ratios, specific indexes, or combinations among bands, band ratios and indexes (Ge et al., 2020). Due to the presence of large outliers, I opted for the *RobustScaler()* algorithm (line 6 of code listings 6.6 and 6.7) for scikit-learn (Immitzer et al., 2016).

For the first attempt of clustering (code listing 6.6), I selected the K-means algorithm fixing the number of clusters to 5 (line 7). Then, I started the unsupervised learning at line 8. Then, at lines 11 and 12, I collected the labels (i.e., a number from 0 to 4) assigned by the K-means algorithm to each element (i.e., each pixel of the image) of the *my_array_1d* and I reported them to the same 2-dimensional geometry of the original image (i.e. Fig. 6.2), respectively. Finally, I plotted the clusters using different colors (i.e. lines 14 to 17) in Fig. 6.6.

For the second attempt of clustering (code listing 6.7), I selected the Gaussian Mixtures algorithm fixing the number of clusters to 5 (line 7). Figure 6.7 reports the clustering result obtained by the Gaussian Mixtures algorithm.

```

1 from sklearn.preprocessing import RobustScaler
2 from sklearn import cluster
3 import matplotlib.colors as mc
4 import matplotlib.pyplot as plt
5
6 X = RobustScaler().fit_transform(my_array_1d)
7 my_ml_model = cluster.KMeans(n_clusters=5)
8 learning = my_ml_model.fit(X)
9 labels_1d = learning.labels_
10
11 labels_1d = my_ml_model.predict(X)
12 labels_2d = labels_1d.reshape(my_array_2d[:, :, 0].shape)
13
14 cmap = mc.LinearSegmentedColormap.from_list("", ["black", "red", "
        yellow", "green", "blue"])
15 fig, ax = plt.subplots(figsize=[18,18])
16 ax.imshow(labels_2d, cmap=cmap)
17 ax.axis('off')
```

Listing 6.6 Making KMeans clustering.

```

1 from sklearn.preprocessing import RobustScaler
2 from sklearn import mixture
3 import matplotlib.colors as mc
4 import matplotlib.pyplot as plt
5
```

```
6 X = RobustScaler().fit_transform(my_array_1d)
7 my_ml_model = mixture.GaussianMixture(n_components=5,
8   covariance_type="full")
9 labels_1d = my_ml_model.predict(X)
10 labels_2d = labels_1d.reshape(my_array_2d[:, :, 0].shape)
11
12 cmap = mc.LinearSegmentedColormap.from_list("", ["black", "red", "yellow",
13   "green", "blue"])
14 fig, ax = plt.subplots(figsize=[18,18])
15 ax.imshow(labels_2d, cmap=cmap)
16 ax.axis('off')
```

Listing 6.7 Making Gaussian Mixture Models clustering.

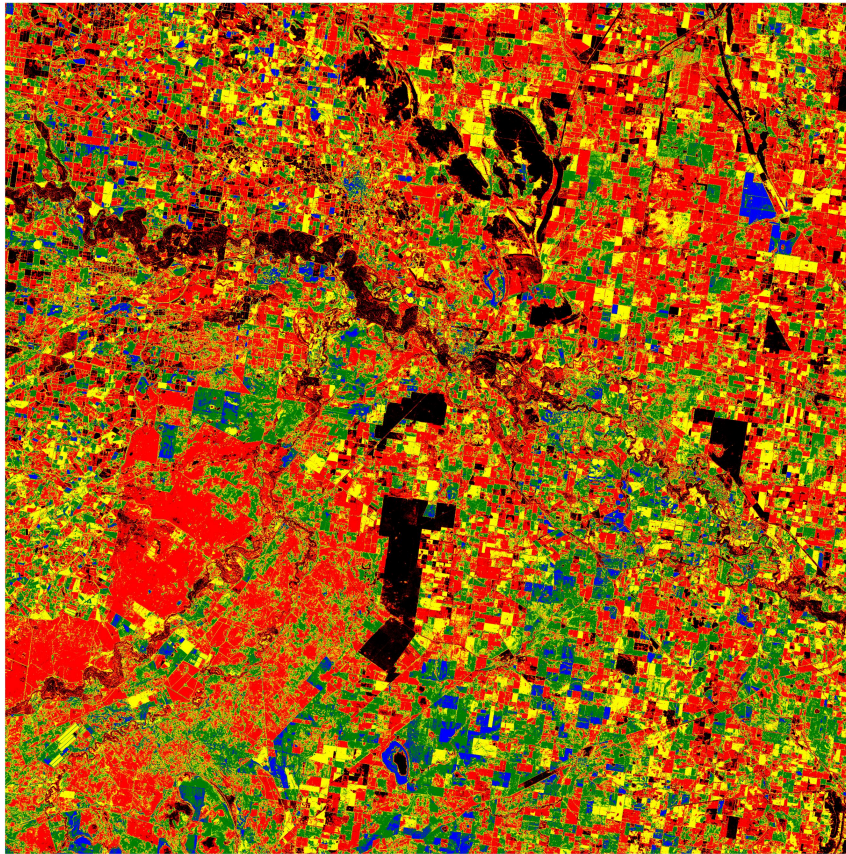


Fig. 6.6 KMeans clustering. Image resulting from code listing 6.6.

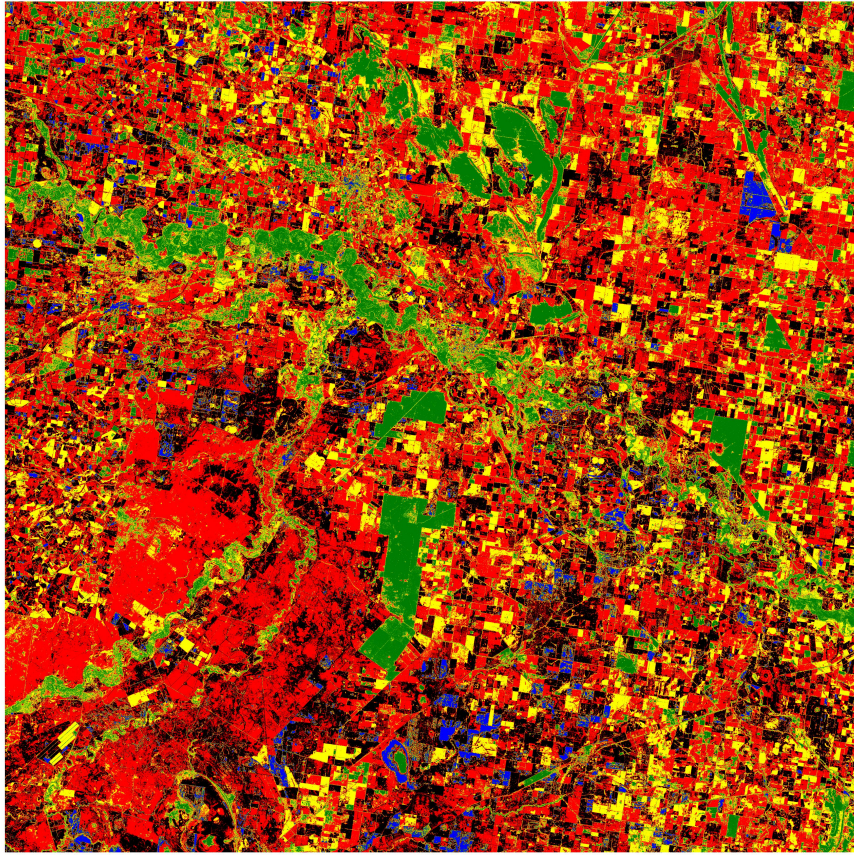


Fig. 6.7 Gaussian Mixture Models. Image resulting from code listing 6.7.

Part III
Supervised Learning

Chapter 7

Supervised Machine Learning Methods

7.1 Supervised Algorithms

Supervised algorithms use the labels of the training data set to learn. In the present chapter, I am going to gently introduce the supervised algorithms for regression and classification reported in Fig.3.5. Also, I will provide some specific references to allow the readers in going deeper into the mathematics behind these ML methods.

7.2 Naive Bayes

Since Bayesian statistic is rarely introduced to geologists in Earth Science courses, I think it could be useful providing an introduction to the Bayes Theorem before describing how it is applied in Machine Learning (e.g., Naive Bayes).

Probabilities: Fig. 7.1 describes our study case where the total number of elements, n_{tot} , in the data set is 10. It consists of 6 porphyritic, 1 holocrystalline, and 3 aphyric igneous rocks. As an example, the probability of randomly picking a rock containing olivines, $P(ol)$, is 3/10. In the Bayesian statistical inference probability $P(ol)$ assumes the name of prior probability. It is the probability of an event before new data is collected.

Conditional Probabilities: Now assuming that we would like to know the probability of picking a rock containing olivines, if I picked a rock characterized by a dark matrix. In this case, the conditional probability, $P(ol|dark)$, is equal to 1/3.

Joint Probabilities: Please take in mind that the term conditional probability is not a synonym of joint probability and these two concepts should not be confused. Also, take care of using the correct notation. In detail, in joint probability, the terms are separated by commas, e.g., $P(ol, dark)$, whereas in conditional probability, the terms are separated by a vertical bar, e.g., $P(ol|dark)$. To note, $P(ol, dark)$ indicates the probability of randomly picking a rock that contains olivines and characterized by a dark matrix, i.e., $P(ol, dark) = 1/10$. On the contrary, $P(ol|dark)$ refers to

the occurrence of a rock containing olivines among those that have a dark matrix, $P(ol|dark) = 1/3$. Joint probabilities and conditional probabilities are related by the following relation:

$$P(ol, dark) = P(ol|dark) \cdot P(dark). \quad (7.1)$$

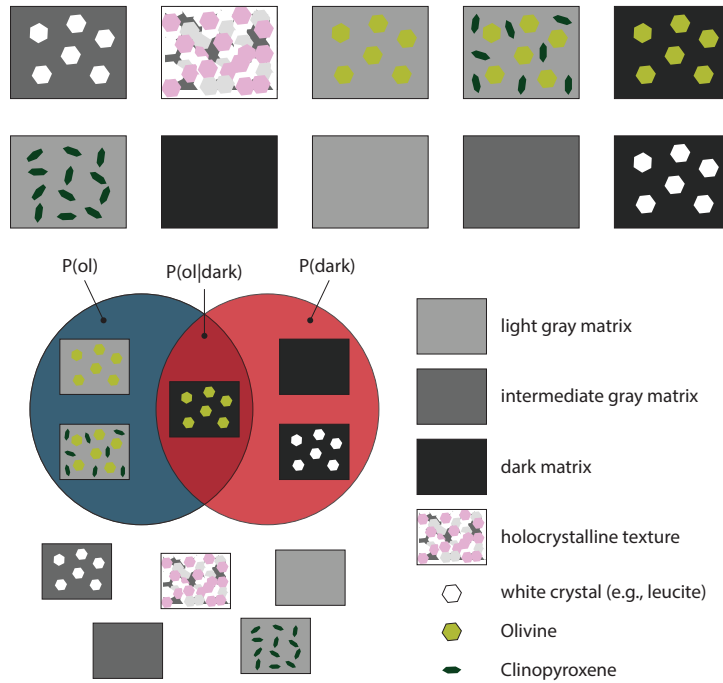


Fig. 7.1 Understanding conditional probabilities and Bayes formulation.

Deriving the Bayes formulation: Similarly to the Eq. 7.1, we could write:

$$P(dark, ol) = P(dark|ol) \cdot P(ol). \quad (7.2)$$

Since $P(dark, ol) = P(ol, dark)$, the right terms of the Eqs. 7.1 and 7.2, must be equal:

$$P(dark|ol) \cdot P(ol) = P(ol|dark) \cdot P(dark). \quad (7.3)$$

Dividing both sides of the Eq. 7.3 by $P(ol)$, we get Bayes formula for our specific case:

$$P(dark|ol) = \frac{P(ol|dark) \cdot P(dark)}{P(ol)}. \quad (7.4)$$

Generalizing the Eq. 7.4, we get the well know Bayes equation:

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}. \quad (7.5)$$

Naive Bayes for classification To understand the Naive Bayes ML algorithm, I propose the same workflow described in **empty citation** Zhang (2004). Assume that the observation you would like to classify is a vector $X = (x_1, x_2, x_3, \dots, x_n)$. Also, c is the label of your class. For simplicity, I assume that c could be only positive (+) or negative (-), i.e., we have two classes only. In this case, the Bayes formula has the following form:

$$P(c|X) = \frac{P(X|c) \cdot P(c)}{P(X)}. \quad (7.6)$$

X is classified as the class $c = +$ if and only if

$$f_b(X) = \frac{P(c = +|X)}{P(c = -|X)} \geq 1, \quad (7.7)$$

where $f_b(X)$ is the Bayesian classifier.

Now assume that all the features are independent, i.e., naive assumption, we can write:

$$P(X|c) = P(x_1, x_2, x_3, \dots, x_n|c) = \prod_{i=1}^n P(x_i|c), \quad (7.8)$$

The resulting classifier [$f_{nb}(X)$], i.e., naive Bayesian classifier, or simply naive Bayes can be written as:

$$f_{nb}(X) = \frac{P(c = +)}{P(c = -)} \prod_{i=1}^n \frac{P(x_i|c = +)}{P(x_i|c = -)}. \quad (7.9)$$

Please note that the naive assumption is a strong constraint for the applicability of the method. In Earth Sciences, attribute interdependence is often violated. In this case, we have two options. The first is to find a way to estimate $P(X|c)$ avoiding the naive assumption (Kubat, 2017). However this first option will inevitably increase the complexity of the problem (Kubat, 2017). More pragmatically, i.e., option 2, we could try reducing the feature dependence by appropriate data pre-processing. As suggested by Kubat (2017), a starting point is to avoid using redundant features.

In scikit-learn the *GaussianNB()* method implements the Gaussian Naive Bayes algorithm for classification with $P(X|c)$ assumed to be Gaussian.

7.3 Quadratic and Linear Discriminant Analysis

Like Naive Bayes, Quadratic and Linear Discriminant Analysis (i.e., QDA and LDA, respectively) rely on the Bayes theorem. Now, assume that $f_c(x)$ is the class-

conditional density of X in class c , and let π_c as the prior probability of class c , with $\sum_{c=1}^K \pi_c = 1$, where K is the number of classes. The Bayes theorem states (Kubat, 2017):

$$P(c|X) = \frac{f_c(x)\pi_c}{\sum_{l=1}^K f_l(x)\pi_l}. \quad (7.10)$$

Now modelling each class density as multivariate Gaussian:

$$f_c(x) = \frac{1}{(2\pi)^{p/2} |\Sigma_c|^{1/2}} e^{-\frac{1}{2}(x-\mu_c)^T \Sigma_c^{-1} (x-\mu_c)}, \quad (7.11)$$

we define the QDA. The LDA constitutes a special case of QDA when assuming that the classes have a common covariance matrix, i.e., $\Sigma_c = \Sigma \forall c$. The main difference between LDA and QDA relies on the resulting decision boundaries, being linear and quadratic functions, respectively.

The algorithms for LDA and QDA are similar, except that separate covariance matrices must be estimated for each class in QDA. When the number of features is large, it implies a dramatic increase in computed parameters. To note considering K classes an p features, LDA and QDA compute $[(K-1)x(p+1)]$ and $\{(K-1)x[p(p+3)/2+1]\}$ parameters, respectively. In scikit-learn, the methods *LinearDiscriminantAnalysis()* and *QuadraticDiscriminantAnalysis()* perform the LDA and QDA, respectively.

7.4 Linear and Nonlinear Models

Sugiyama (2015) defines d-dimensional linear-in-parameter models as:

$$f_{\theta}(x) = \sum_{j=1}^b \theta_j \phi_j(x) = \theta^T \phi(x), \quad (7.12)$$

where x , ϕ , and θ are a d-dimensional input vector, a basis function and its parameters, respectively. Also, b denotes the number of basis functions. As an example, considering a one-dimensional input, the Eq.7.12 reduces to:

$$f_{\theta}(x) = \sum_{j=1}^b \theta_j \phi_j(x) = \theta^T \phi(x), \quad (7.13)$$

where:

$$\phi(x) = (\phi_1(x), \dots, \phi_b(x))^T, \quad (7.14)$$

and

$$\theta = (\theta_1, \dots, \theta_b)^T. \quad (7.15)$$

To note, linear-in-parameter models are linear in terms of θ , and they can handle straight lines, i.e., linear-in-input models (e.g., code listing 7.1 and Fig. 7.2):

$$\phi(x) = (1, x)^T, \quad (7.16)$$

$$\theta = (\theta_1, \theta_2)^T, \quad (7.17)$$

but they can also manage nonlinear functions, e.g., polynomials (e.g., code listing 7.1 and Fig. 7.2):

$$\phi(x) = (1, x, x^2, \dots, x^{b-1})^T, \quad (7.18)$$

$$\theta = (\theta_1, \theta_2, \dots, \theta_b)^T. \quad (7.19)$$

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 x = np.arange(1,6)
5 y = np.array([0,1,2,9,9])
6
7 fig, ax = plt.subplots()
8 ax.scatter(x, y, marker = 'o', s = 100, color = '#c7ddf4',
9           edgcolor = 'k')
10
11 orders = np.array([1,2,4])
12 colors = ['#ff464a', '#342a77', '#4881e9']
13 linestyles = ['-', '--', '-.']
14 for order, color, linestyle in zip(orders, colors, linestyles):
15     betas = np.polyfit(x, y, order)
16     func = np.poly1d(betas)
17     x1 = np.linspace(0.5, 5.5, 1000)
18     y1 = func(x1)
19     ax.plot(x1, y1, color=color, linestyle=linestyle, label="
20           Linear-in-parameters model of order " + str(order))
21 ax.legend()
22 ax.set_xlabel('A quantity relevant in geology\n(e.g., time)')
23 ax.set_ylabel('A quantity relevant in geology\n(e.g., spring flow
24           rate)')
25 fig.tight_layout()

```

Listing 7.1 Polynomial regression as example of linear-in-parameters modelling.

Considering a p -values input vector x , linear-in-parameter models are still able to manage linear-in-input problems, i.e., managing hyper-planes:

$$\phi(x) = (1, x_1, x_2, \dots, x_p)^T, \quad (7.20)$$

$$\theta = (\theta_1, \theta_2, \dots, \theta_b)^T. \quad (7.21)$$

In this case, the number of the basis functions corresponds to the dimension of the input vector plus one, i.e., $b = p + 1$. Some authors prefer reporting the first term of θ separately, naming it bias (i.e., θ_0), and reshaping the formulation as follow:

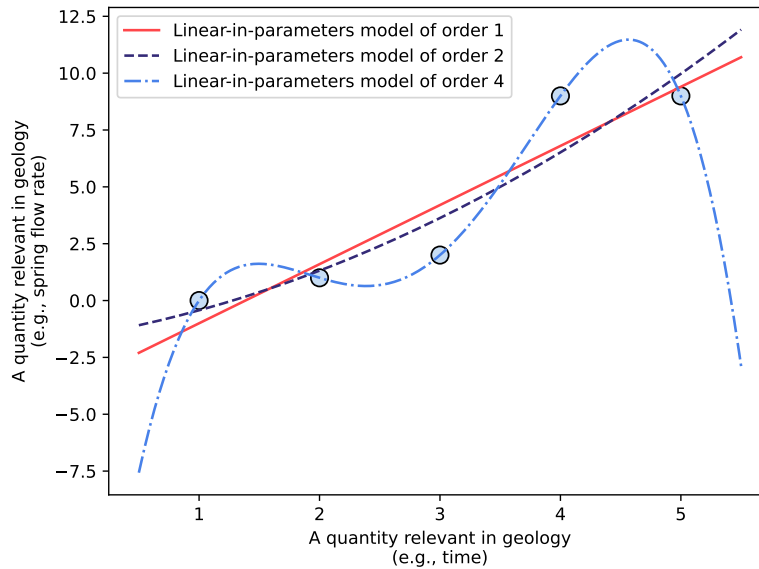


Fig. 7.2 Result of code listing 7.1.

$$\boldsymbol{\phi}(\mathbf{x}) = (x_1, x_2, \dots, x_{b=p})^T, \quad (7.22)$$

$$\boldsymbol{\theta} = (\beta_0, \boldsymbol{\beta}), \quad (7.23)$$

with:

$$\boldsymbol{\beta} = (\beta_1, \beta_2, \dots, \beta_{b=p})^T, \quad (7.24)$$

All the $f_{\boldsymbol{\theta}}(\mathbf{x})$ models that cannot be expressed as linear, in terms of parameters, fall in the field of nonlinear modelling (Sugiyama, 2015).

7.5 Loss Functions, Cost Functions, and Gradient Descent

Most ML algorithms involve the optimization of our model (e.g., $f_{\boldsymbol{\theta}}(\mathbf{x})$ in Eq. 7.13). For the purposes of the present book, the term optimization refers to adjusting model parameters $\boldsymbol{\theta}$ to minimize or maximize a function that measures the agreement between the model and the training data.

As a general term, the function we want to minimize or maximize is named the **objective function (empty citation)**. In the case of minimization, the objective function takes names like cost function, loss function, and error function. These terms are often interchangeable Goodfellow et al. (2016), but sometimes the authors use a specific term, e.g., loss or cost function, to describe a specific task.

As an example, some authors use the term **loss function** to measure how well a model agrees with a single label in the training data set (Goodfellow et al., 2016). The square loss could be an example of a loss function:

$$L(\boldsymbol{\theta}) = [y_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)]^2, \quad (7.25)$$

where y_i and $f_{\boldsymbol{\theta}}(\mathbf{x}_i)$ are the labeled (i.e., true or measured) values and those predicted by our model, respectively. Also, \mathbf{x} and $\boldsymbol{\theta}$ are the inputs and the parameters governing the model, respectively.

Similarly, the **cost function** evaluates the loss function over the entire data set and helps in evaluating the overall performance of a model (Goodfellow et al., 2016). The mean squared-error is an example of cost functions:

$$C(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n [y_i - f_{\boldsymbol{\theta}}(\mathbf{x}_i)]^2, \quad (7.26)$$

where n is the number of elements in the training data set.

Typically, our aim is to minimize the cost function, i.e., $C(\boldsymbol{\theta})$, and the Gradient Descent (GD) is a method to achieve our goal. In detail, GD works by updating the parameters (in our case $\boldsymbol{\theta}$) governing our model, i.e., $f_{\boldsymbol{\theta}}(\mathbf{x})$, in the opposite direction of the cost function gradient (Sugiyama, 2015), i.e., $\nabla C(\boldsymbol{\theta})$:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \gamma \nabla C[\boldsymbol{\theta}]. \quad (7.27)$$

In the simplest example of linear regression with \mathbf{x} in \mathbb{R} :

$$f_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_1 + \theta_2 \cdot x, \quad (7.28)$$

the mean squared-error cost function can be written as:

$$C(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^n [y_i - (\theta_1 + \theta_2 \cdot x_i)]^2, \quad (7.29)$$

Note that the simple linear example in \mathbb{R} can be easily generalized to \mathbb{R}^d . Also, note that the example of linear regression proposed here has a well known and easy to apply least squares analytical solution in the case of linearity (i.e., the relationship between \mathbf{x} and the mean of \mathbf{y} is linear), independence (i.e., the observations are independent of each other), and normality (for any fixed value of \mathbf{x} , \mathbf{y} is normally distributed (Petrelli, 2021)). However, it provides a self explanatory example of how GD does work.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3 line_colors = ['#F15C61', '#0F7F8B', '#0A3A54', '#C82127']
4
5 # linear data set with noise
```

```

6 n = 100
7 theta_1, theta_2 = 3, 1 # target value for theta_1 & theta_2
8 x = np.linspace(-10, 10, n)
9 np.random.seed(40)
10 noise = np.random.normal(loc=0.0, scale=1.0, size=n)
11 y = theta_1 + theta_2 * x + noise
12 fig, (ax1, ax2) = plt.subplots(2, 1, figsize=(6, 12))
13 ax1.scatter(x, y, c='#BFD7EA', edgecolor='k')
14
15 my_theta_1, my_theta_2 = 0, 0 # arbitrary initial values
16 gamma = 0.0005 # learning rate
17 t_final = 10001 # number of iterations
18 n = len(x)
19 to_plot, cost_function = [1, 25, 500, 10000], []
20 # Gradient Descent
21 for i in range(t_final):
22     #Eq. 4.30
23     D_theta_1 = (-2/n)*np.sum(y-(my_theta_1 + my_theta_2*x))
24     #Eq. 4.31
25     D_theta_2 = (-2/n)*np.sum(x*(y-(my_theta_1+my_theta_2*x)))
26
27     my_theta_1 = my_theta_1 - gamma * D_theta_1 #Eq. 4.32
28     my_theta_2 = my_theta_2 - gamma * D_theta_2 #Eq. 4.33
29     cost_function.append((1/n) * np.sum(y - (my_theta_1 +
30     my_theta_2 * x))**2)
31
32     if i in to_plot:
33         color_index = to_plot.index(i)
34         my_y = my_theta_1 + my_theta_2 * x
35         ax1.plot(x,my_y, color=line_colors[color_index],
36                 label='iter: {:.0f}'.format(i) + ' - ' +
37                     r'$\theta_1 = $' + '{:.2f}'.format(my_theta_1) +
38                     ' - ' +
39                     r'$\theta_2 = $' + '{:.2f}'.format(my_theta_2))
40 ax1.set_xlabel('x')
41 ax1.set_ylabel('y')
42 ax1.legend()
43 cost_function = np.array(cost_function)
44 iterations = range(t_final)
45 ax2.plot(iterations,cost_function, color='#C82127',
46         label='mean squared-error cost function Eq.4.29')
47 ax2.set_xlabel('Iteration')
48 ax2.set_ylabel('Cost Function Value')
49 ax2.legend()
50 fig.tight_layout()

```

Listing 7.2 A simple example of Gradient Descent in Python.

To develop a GD, the first step consists of computing the partial derivative of $C(\theta)$ respect θ_1 and θ_2 , respectively. Therefore we can write:

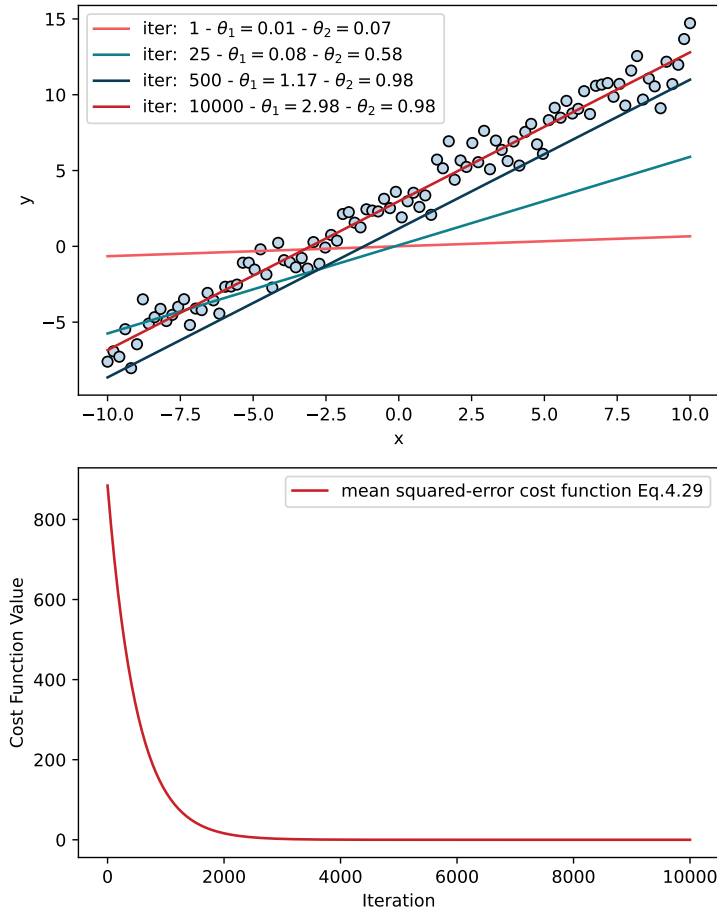


Fig. 7.3 Linear fitting estimates and cost function evolution resulting from code listing 7.2.

$$D_{\theta_1} = \frac{-2}{n} \sum_{i=1}^n \cdot [y_i - (\theta_1 + \theta_2 \cdot x_i)] \quad (7.30)$$

$$D_{\theta_2} = \frac{-2}{n} \sum_{i=1}^n \cdot [y_i - (\theta_1 + \theta_2 \cdot x_i)] \cdot x_i \quad (7.31)$$

Then the GD optimizes the parameters of our model, with an iterative approach:

$$\theta_1^{t+1} = \theta_1^t - \gamma D_{\theta_1}, \quad (7.32)$$

$$\theta_2^{t+1} = \theta_2^t - \gamma D_{\theta_2}, \quad (7.33)$$

where γ is an adequately chosen learning rate. The code listing 7.2 and Fig. 7.3 report how to develop the GD optimization described by Eq.7.28-7.32.

The Stochastic Gradient Descent (SGD) algorithm (Bottou, 2012) simplifies the GD by estimating the gradient of $C(\boldsymbol{\theta})$ on the basis of a single, randomly picked, example $\hat{f}_{\boldsymbol{\theta}_t}(\mathbf{x}_t)$:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \gamma \nabla C[y, \hat{f}_{\boldsymbol{\theta}_t}(\mathbf{x}_t)]. \quad (7.34)$$

The *SGDClassifier()* and *SGDRegressor()* in *sklearn.linear_model* allow the execution of a SGD in the field of classification and regression, respectively. Often, we use an approach that is something in between GD and SGD by estimating the gradient using a small random portion of the training data set. This approach takes the name of mini-batch GD.

To summarize, GD always uses the whole learning data set. Differently to GD, SGD and mini-batches GD compute the gradient using a single sample and a small portion of the training data set, respectively.

SGD and mini-batches GD work better than GD for error surfaces characterized by many local maxima and minima. In this case, the GD will stop at the first local minimum whereas SGD and mini-batches GD, being much noisier than GD, will tend to explore neighbour areas of the gradient hopefully finding better solutions. To note, the pure SGD is really noisy, whereas mini-batches GD tends to average the computed gradient, resulting more stable than SGD. In ML, the use of SGD and mini-batches GD largely exceeds the GD because it is too much computationally expensive, with a minimum gain in accuracy for convex problems. In the case of many local maxima and minima, SGD and mini-batches GD are also more efficient than GD in terms of accuracy, being able to “jump” over local minima, hopefully finding better solutions.

7.6 Ridge Regression

Ridge Regression is a least squares method that shrinks the regression coefficients with a penalty on their size (Hastie et al., 2017). Mathematically, with the labeled data set (\mathbf{x}_i, y_i) , where y_i are the labels and \mathbf{x}_i the predictor variables, i.e., the inputs, with $\mathbf{x}_i = (x_{i1}, x_{i2}, \dots, x_{ip})^T$ (Hastie et al., 2017; Tibshirani, 1996).

The cost function in Ridge Regression can be expressed as (Hastie et al., 2017; Tibshirani, 1996):

$$C(\beta_0, \boldsymbol{\beta}) = \frac{1}{2n} \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^p \theta_j^2. \quad (7.35)$$

where the λ parameter is named regularization penalty. The ridge regression performs the so-called L2-norm regularization by adding a penalty equivalent to the square of the magnitude of coefficients, i.e., the second term of Eq. 7.35.

In the limit case $\lambda = 0$, we go back to Ordinary Least Square. A correct choice of λ will help you in avoiding over-fitting issues. On the contrary, if λ starts approaching very large values, you are probably experiencing under-fitting issues.

7.7 Least Absolute Shrinkage and Selection Operator (LASSO)

The “Least Absolute Shrinkage and Selection Operator,” also known as the LASSO, is a method to solve linear problems by minimizing the residual sum of squares subject to the sum of the absolute value of the coefficients being less than a constant (Tibshirani, 1996). The main characteristic of LASSO relies on the tendency to prefer solutions with fewer non-zero coefficients, thus reducing the number of features of the solution. The LASSO cost function can be expressed as (Tibshirani, 1996):

$$C(\beta_0, \boldsymbol{\beta}) = \frac{1}{2n} \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^p |\beta_j|. \quad (7.36)$$

Differently from the ridge regression, the LASSO algorithm performs the so-called L1-norm regularization by adding a penalty equivalent to the sum of coefficients absolute values, i.e., the second term of Eq. 7.36.

To note, the LASSO makes dimensionality reduction, i.e., it reduces the number of features of the solution, and shrinkage, whereas ridge regression, in contrast, only shrinks (Hastie et al., 2017; Tibshirani, 1996).

7.8 Elastic-Net

The Elastic-Net (H. Zou & Hastie, 2005) is a linear regression model that performs both L1- and L2-norm regularization (Friedman et al., 2010):

$$C(\beta_0, \boldsymbol{\beta}) = \frac{1}{2n} \sum_{i=1}^n \left(y_i - \beta_0 - \sum_{j=1}^p x_{ij} \beta_j \right)^2 + \lambda \sum_{j=1}^p \left[\frac{1-\alpha}{2} \beta_j^2 + \alpha |\beta_j| \right]. \quad (7.37)$$

In the case of $\alpha = 1$, Elastic-net is the same as LASSO. Also, for $\alpha = 0$, Elastic-net approaches ridge regression. For $0 < \alpha < 1$, the penalty term (i.e., the second term of Eq. 7.37) is between the L1- and L2-norm regularization.

7.9 Support Vector Machines

Support Vector Machines (SVMs) are a set of supervised ML algorithms that works remarkably well in the context of classification (Cortes & Vapnik, 1995). The strength of SVMs mainly relies on these three features: (1) SVMs are efficient in high-dimensional spaces; (2) SVMs effectively model real-world problems; (3) SVMs perform well on data sets with many attributes, even in the case of a low number of cases which might be available to train the model (Cortes & Vapnik, 1995). SVMs numerically implement the following idea: inputs are mapped to a some high-dimension feature space F by some non-linear mapping (Cortes & Vapnik, 1995). In the space Z a linear decision surface is then constructed (Cortes & Vapnik, 1995).

To start, consider a labeled training data set (y_i, \mathbf{x}_i) where \mathbf{x}_i is p -dimensional, i.e., $\mathbf{x}_i = (x_{1i}, x_{2i}, \dots, x_{pi})$, with $i=1, 2, \dots, n$ where n is the number of samples. Also assume that the label y_i is equal to 1 for the first class and -1 for the second class, defining a two classes classification problem, i.e., $y_i \in \{-1, 1\}$.

Also, define a linear classifier, based on the following linear-in-inputs discriminant function:

$$f(\mathbf{x}) = \mathbf{w}^T \cdot \mathbf{x} + b, \quad (7.38)$$

The decision boundary between the two classes, i.e., regions classified as positive and negative, that is defined by Eq. 7.38 is an hyperplane.

The two classes are linearly separable if there exist a vector \mathbf{w} and a scalar b , such that:

$$(\mathbf{w}^T \mathbf{x}_i + b)y_i \geq 1, \forall i = 1, 2, \dots, n. \quad (7.39)$$

It means that we are able to correctly classify all samples. It follows the definition of the optimal hyperplane as the one which separates the training data set with a maximal margin, i.e., $m(\mathbf{w})$:

$$m(\mathbf{w}) = \frac{1}{\|\mathbf{w}\|} \quad (7.40)$$

Finally, the maximum-margin classifier, i.e. hard margin support vector machine, is the discriminant function that maximizes $m(\mathbf{w})$, which is equivalent to minimizing $\|\mathbf{w}\|^2$:

$$\min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|^2 \quad (7.41)$$

subject to:

$$(\mathbf{w}^T \mathbf{x}_i + b)y_i \geq 1, \forall i = 1, 2, \dots, n \quad (7.42)$$

The hard margin support vector machine requires a strong assumption, i.e., the linear separability of classes, which can be considered as an exception, not the rule. To allow errors, i.e., $\xi = (\xi_1, \xi_2, \dots, \xi_n)$, we can introduce the concept of soft margin support vector machine:

$$\min_{\mathbf{w}, b, \xi} \left[\frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^n \xi_i \right] \quad (7.43)$$

subject to

$$(\mathbf{w}^T \mathbf{x}_i + b)y_i \geq 1 - \xi_i, \xi_i \geq 0, \forall i = 1, 2, \dots, n \quad (7.44)$$

where $C > 0$ is a tunable parameter that controls the margin errors. The linear classifier defined by Eq. 7.38 can be generalized to non linear inputs by defining the discriminant function as (Cortes & Vapnik, 1995):

$$f(\mathbf{x}) = \mathbf{w}^T \cdot \phi(\mathbf{x}) + b \quad (7.45)$$

where $\phi(\mathbf{x})$ is a function that maps non linearly separable inputs \mathbf{x} to a feature space F of higher dimension. Now, if we express the weight vector \mathbf{w} as a linear combination of the training examples, i.e., $\mathbf{w} = \sum_{i=1}^n \alpha_i \mathbf{x}_i$, it follows that, in the feature space F , we have:

$$f(\mathbf{x}) = \sum_{i=1}^n \alpha_i \phi(\mathbf{x}_i)^T \phi(\mathbf{x}) + b \quad (7.46)$$

The idea behind Eq. 7.45 and Eq. 7.46 is to map a non linear classification function to a feature space F of higher dimension where the classification function is linear Fig. 7.4. Now defining a kernel function $K(\mathbf{x}_i, \mathbf{x})$ as:

$$K(\mathbf{x}_i, \mathbf{x}) = \phi(\mathbf{x}_i)^T \phi(\mathbf{x}) \quad (7.47)$$

we have:

$$f(\mathbf{x}) = \sum_{i=1}^n \alpha_i K(\mathbf{x}_i, \mathbf{x}) + b \quad (7.48)$$

please note: using the kernel function, we do not need to know or compute $\phi()$, allowing a linear transformation to the problem at higher dimension. The scikit-learn implementation of support vector machines, e.g., `SVC()` and `SVR()` allow the use of linear, polynomial, sigmoid, and radial Basis kernel Functions [$K(\mathbf{x}_i, \mathbf{x})$, Table 7.1].

Table 7.1 Kernel functions in scikit-learn for the `SVC()` and `SVR()` methods

Kernel Function	Equation	Identifier
linear	$K(\mathbf{x}_i, \mathbf{x}) = (\mathbf{x}_i \cdot \mathbf{x}')$	kernel='linear'
polynomial	$K(\mathbf{x}_i, \mathbf{x}) = (\mathbf{x}_i \cdot \mathbf{x}' + r)^d$	kernel='poly'
sigmoid	$K(\mathbf{x}_i, \mathbf{x}) = \tanh(\mathbf{x}_i \cdot \mathbf{x}' + r)$	kernel='sigmoid'
radial basis function	$K(\mathbf{x}_i, \mathbf{x}) = \exp(-\lambda \ \mathbf{x}_i - \mathbf{x}'\ ^2)$	kernel='rbf'

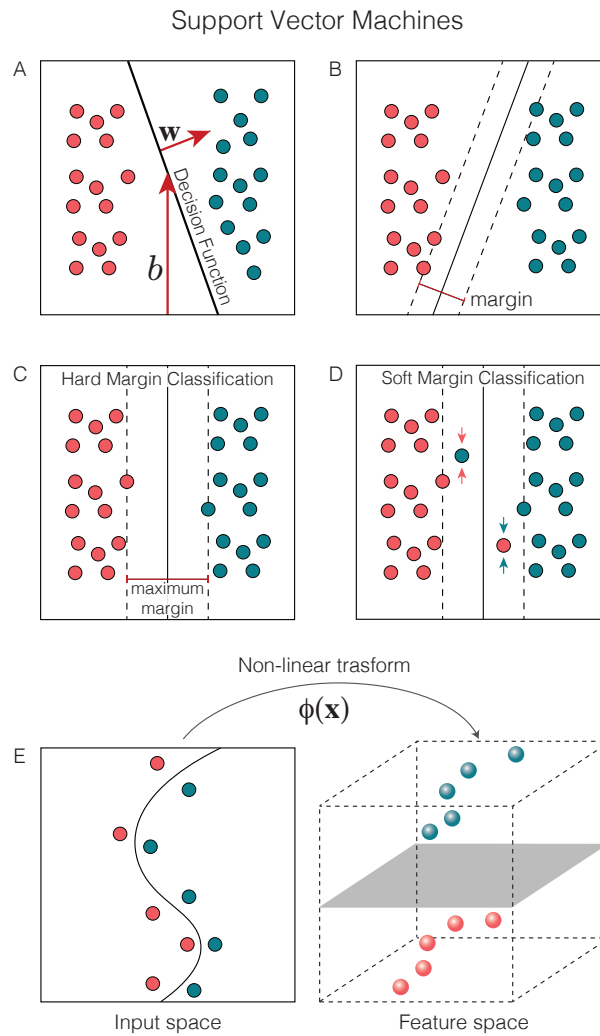


Fig. 7.4 Support Vector Machines. Redrawn from (Sugiyama, 2015)

7.10 Supervised Nearest Neighbors

The supervised k -nearest neighbors is a ML algorithm that uses similarities, such as distance functions (Bentley, 1975) to fulfill regression and classification tasks. In detail, the k -nearest neighbors method predicts numerical targets using a metric that is typically the inverse-distance-weighted average of k -nearest neighbors (Bentley, 1975). The weights can be uniform or calculated by a kernel function. The Euclidean distance metric is commonly used to measure the distance between two instances,

but other metrics are available (See Table 7.2). Please note that Minkowski distance reduces to the Manhattan and Euclidean distance when p is equal to 1 and 2, respectively. Bentley (1975) reports an extensive and detailed description of the k -nearest neighbors algorithm.

In scikit-learn the *KNeighborsClassifier()* and *KNeighborsRegressor()* methods perform classification and regression tasks based on k -nearest neighbors, respectively.

Table 7.2 Selected distance metrics that can be used in Supervised Nearest Neighbors and other ML algorithms

Distance	Identifier	Arguments	Equation
Euclidean	'euclidean'	none	$\sqrt{\sum_{j=1}^D x_j - y_j ^2}$
Manhattan	'manhattan'	none	$\sum_{j=1}^D x_j - y_j $
Chebyshev	'chebyshev'	none	$\max x_j - y_j $
Minkowski	'minkowski'	$p, (w = 1)$	$\left(\sum_{j=1}^D w x_j - y_j ^p\right)^{1/p}$

7.11 Trees Based Methods

Decision Trees. Before start describing how decision trees work let me introduce few definitions highlighted in Fig. 7.5. **Root Node:** The starting node of a decision tree. It contains the entire data set involved in the process. **Parent Node:** A node that gets split into sub-nodes is defined as Parent Node. **Child Node:** sub-nodes deriving by a parent node are defined as Child Nodes. **Leaf or Terminal Nodes:** Nodes that terminate the tree and that are not split to generate additional child nodes are defined as Terminal or Leaf Nodes.

The decision tree (Breiman et al., 1984) algorithm and its modifications (e.g., Random Forests and Extra Trees) split the input space into sub-regions allowing regression and classification tasks [Kubat (2017); Fig. 7.5]. In detail, each node maps a region in the input space, which is further divided within the node into sub-regions using splitting criteria. Therefore, the workflow of a decision tree consists of progressively splitting the input space by a sequence of decisions (i.e., splittings) into non-overlapping regions, with a one-to-one correspondence between leaf nodes and input regions (Kubat, 2017). The decision tree algorithm, despite its attractiveness due to the simplicity of the algorithm formulation and the easy interpretation of the results, it is often prone to over-fitting and under-fitting issues (cf. section 3.5), making it less accurate than other predictors (Song & Lu, 2015). Also, it does not

work effectively in the presence of highly-correlated input features (Song & Lu, 2015).

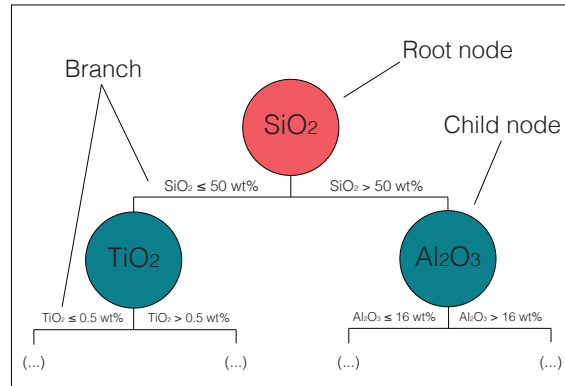


Fig. 7.5 The Decision Tree algorithm.

To avoid over-fitting and under-fitting issues, a more robust algorithm, named ensemble predictors, have been developed. Examples are random forest, gradient boosting regression, and extremely randomized tree methods. More details on the single decision tree model are available in (Breiman et al., 1984).

Random Forest. The Random Forest algorithm (Breiman, 2001) is based on the “bagging”, i.e., bootstrap aggregation, a technique that averages the prediction over a collection of bootstrap samples, thereby reducing its variance (Hastie et al., 2017). In detail, the random forest uses the bagging for creating multiple versions of a predictor, i.e., multiples trees, then evaluated to obtain an aggregated predictor (Hastie et al., 2017). Specifically, for a given training data set with sample size n , the bagging produces k new training sets, each with sample size n , by uniformly sampling from the original training data set with replacement, i.e., by bootstrapping (Hastie et al., 2017). Then, k decision trees are trained by using the k newly created training sets and coupled by averaging for regression or majority voting for classification, respectively (Hastie et al., 2017). A detailed description of the Random Forest algorithm is available in the literature (Breiman, 2001; Hastie et al., 2017).

Extremely Randomized Trees. The Extremely Randomized Trees algorithm (Geurts et al., 2006) is similar to the Random Forest with two main differences: (1) it splits nodes by choosing fully random cut points and (2) it uses the entire learning sample rather than a bootstrapped replica to grow the trees (Geurts et al., 2006). The predictions of the trees are aggregated to yield the final prediction by majority vote in the classification and by arithmetic averaging in the regression (Geurts et al., 2006). A complete description of the Extremely Randomized Trees algorithm is given in (Geurts et al., 2006).

Chapter 8

Well Log Data Facies Classification by Machine Learning

8.1 Motivation

Facies recognition in wells by well-log data analysis is a common task in many geological fields like trap reservoir characterization, sedimentology analyses, and depositional-environment interpretations (Hernandez-Martinez et al., 2013; Wood, 2021). I started conceiving this chapter when I discovered the FORCE 2020¹ Machine Learning competition (Bormann) and the SEG 2016² ML contest (M. Hall & Hall, 2017). In these two contests, students and early career researchers have been involved in the attempt of identifying correct lithofacies in a blind data set of well-log data (i.e., gamma-ray, resistivity, photoelectric effect, etc...) using an ML algorithm of their selection to be trained on a labeled data set made available for all the competitors. The competitors of the 2016 edition have been supported by a tutorial by Brendon Hall (B. Hall, 2016) and Hall and Hall (M. Hall & Hall, 2017). Also, (Bestagini et al., 2017), in a pleasant paper, describe a strategy to achieve the final goal for the 2016 edition. It is worth noticing that the starter notebook³ of the FORCE 2020 Machine Learning competition contains all you need to begin. In detail, it shows how to import the training data set, perform the inspection of the imported data set, and start developing a model based on the Random Forest algorithm.

Here, I will focus on the FORCE 2020 Machine Learning competition. In detail, I will progressively develop a ML workflow (i.e., descriptive statistics, algorithm selection, model optimization, training of the selected model, and application to the blind data set) discussing each step and aiming at making everything as much easy as possible.

¹ <https://github.com/bolgebrygg/Force-2020-Machine-Learning-competition>

² <https://github.com/seg/2016-ml-contest>

³ https://bit.ly/force2020_ml_start

8.2 Inspection of the Data Sets and Pre-Processing

For the FORCE 2020 Machine Learning competition⁴, a starter Jupyter Notebook has been made available on GitHub together with a labeled train dataset (i.e., the compressed train.zip file containing a single file: train.csv) and two tests (i.e., leaderboard_test_features.csv and hidden_test.csv)⁵. Nowadays, all three files are labeled, i.e., they also contain the correct solution either in a column named FORCE_2020_LITHOFACIES_LITHOLOGY or in a separate file. The above data set contains well-log data for more than 90 wells from offshore Norway (**Hall2016FaciesLearning; Bormann2020FORCECompetition**).

We start importing the three data sets using pandas and looking at the spatial distribution of the investigated wells (code listing 8.1; Fig. 8.1).

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3
4 data_sets = ['train.csv', 'hidden_test.csv', '
               leaderboard_test_features.csv']
5 labels = ['Train data', 'Hidden test data', 'Leaderboard test
            data']
6 colors = ['#BFD7EA', '#0A3A54', '#C82127']
7
8 fig, ax = plt.subplots()
9
10 for my_data_set, my_color, my_label in zip(data_sets, colors,
        labels):
11
12     my_data = pd.read_csv(my_data_set, sep=';')
13     my_Weels = my_data.drop_duplicates(subset=['WELL'])
14     my_Weels = my_Weels[['X_LOC', 'Y_LOC']].dropna() / 100000
15
16     ax.scatter(my_Weels['X_LOC'], my_Weels['Y_LOC'],
17               label=my_label, s=80, color=my_color,
18               edgecolor='k', alpha=0.8)
19
20 ax.set_xlabel('X_LOC')
21 ax.set_ylabel('Y_LOC')
22 ax.set_xlim(4,6)
23 ax.set_ylim(63,70)
24 ax.legend(ncol=3)
25 plt.tight_layout()

```

Listing 8.1 Spatial distribution of the investigated wells.

Observing Fig. 8.1, it emerges that the distribution of the wells seems to define three main clusters. As a geologist, I could expect that closer wells may show similar

⁴ <https://xeek.ai/challenges/force-well-logs/overview>

⁵ https://bit.ly/force2020_ml_data

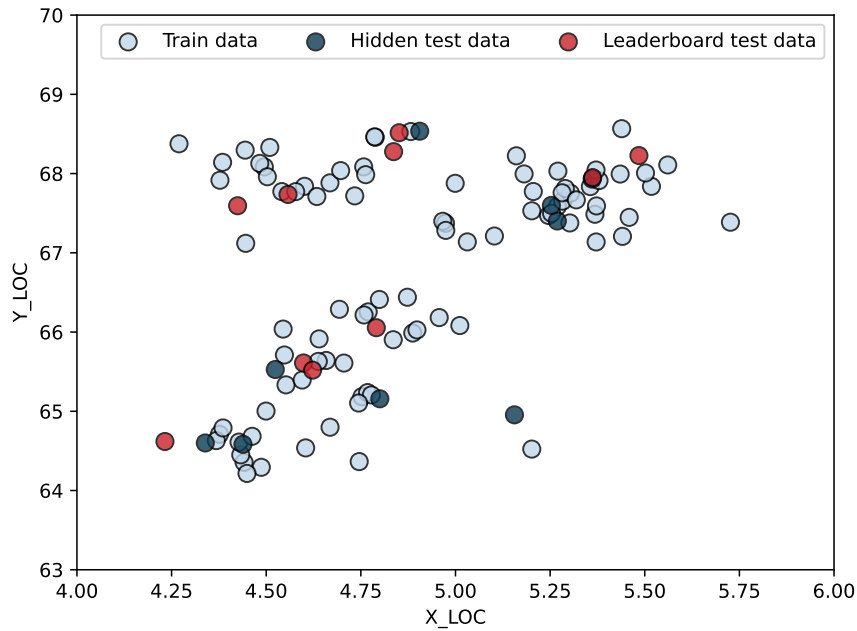


Fig. 8.1 Result of code listing 8.1. Spatial distribution of the investigated wells.

lithofacies distributions. Therefore, the position of the well could have a significant role in the training of our ML model. To include the spatial distribution of the wells in a ML model, many different strategies could be adopted. Among these, including X_LOC and Y_LOC as model features is the easiest one. More refined strategies may include a preliminary clustering of wells spatial distribution and a learning approach based on the result of the clustering. Keeping in mind that we would like to develop a smart and simple workflow, I opted for the first option, i.e., simply including X_LOC and Y_LOC as model features.

Going ahead and looking at Fig. 8.2 (the result of code listing 8.2), two main characteristics of the investigated data sets clearly appear.

The first relates to feature persistence. Many features, e.g., SGR, DTS, RMIC, and ROPA, contain more than 60% of missing values (i.e., upper panel of Fig. 8.2). As a consequence, a strategy to deal with missing values is mandatory. In agreement with the aim of simplicity of the ML workflow presented in the present chapter, I decided to use only the features containing less than 40% of missing values. Also, within them, I decided to replace missing values with the average of each feature. In statistics, the procedure of substituting missing values with substituted values is named feature imputation (Q. Zou et al., 2015). In scikit-learn, *SimpleImputer()* and *IterativeImputer()* may help in feature imputation.

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 lithology_keys = {30000: 'Sandstone',
6                  65030: 'Sandstone/Shale',
7                  65000: 'Shale',
8                  80000: 'Marl',
9                  74000: 'Dolomite',
10                 70000: 'Limestone',
11                 70032: 'Chalk',
12                 88000: 'Halite',
13                 86000: 'Anhydrite',
14                 99000: 'Tuff',
15                 90000: 'Coal',
16                 93000: 'Basement'}
17
18 train_data = pd.read_csv('train.csv', sep=';')
19
20 class_abundance = np.vectorize(lithology_keys.get)(
21     train_data['FORCE_2020_LITHOFACIES_LITHOLOGY'].values)
22 unique, counts = np.unique(class_abundance, return_counts=True)
23
24 my_colors = ['#0F7F8B'] * len(unique)
25 my_colors[np.argmax(counts)] = '#C82127'
26 my_colors[np.argmin(counts)] = '#0A3A54'
27
28 fig, (ax1, ax2) = plt.subplots(2,1, figsize=(7,14))
29
30 ax2.barh(unique, counts, color=my_colors)
31 ax2.set_xscale("log")
32 ax2.set_xlim(1e1, 1e6)
33 ax2.set_xlabel('Number of Occurrences')
34 ax2.set_title('Class Inspection')
35
36 Feature_presence = train_data.isna().sum()/train_data.shape
37     [0]*100
38
39 Feature_presence = Feature_presence.drop(
40     labels=['FORCE_2020_LITHOFACIES_LITHOLOGY',
41            'FORCE_2020_LITHOFACIES_CONFIDENCE', 'WELL
42     '])
43
44 Feature_presence.sort_values().plot.barh(color='#0F7F8B', ax=ax1)
45 ax1.axvline(40, color='#C82127', linestyle='--')
46 ax1.set_xlabel('Percentage of Missing Values')
47 ax1.set_title('Feature Inspection')
48
49 plt.tight_layout()

```

Listing 8.2 Inspect feature persistence and class balancing.

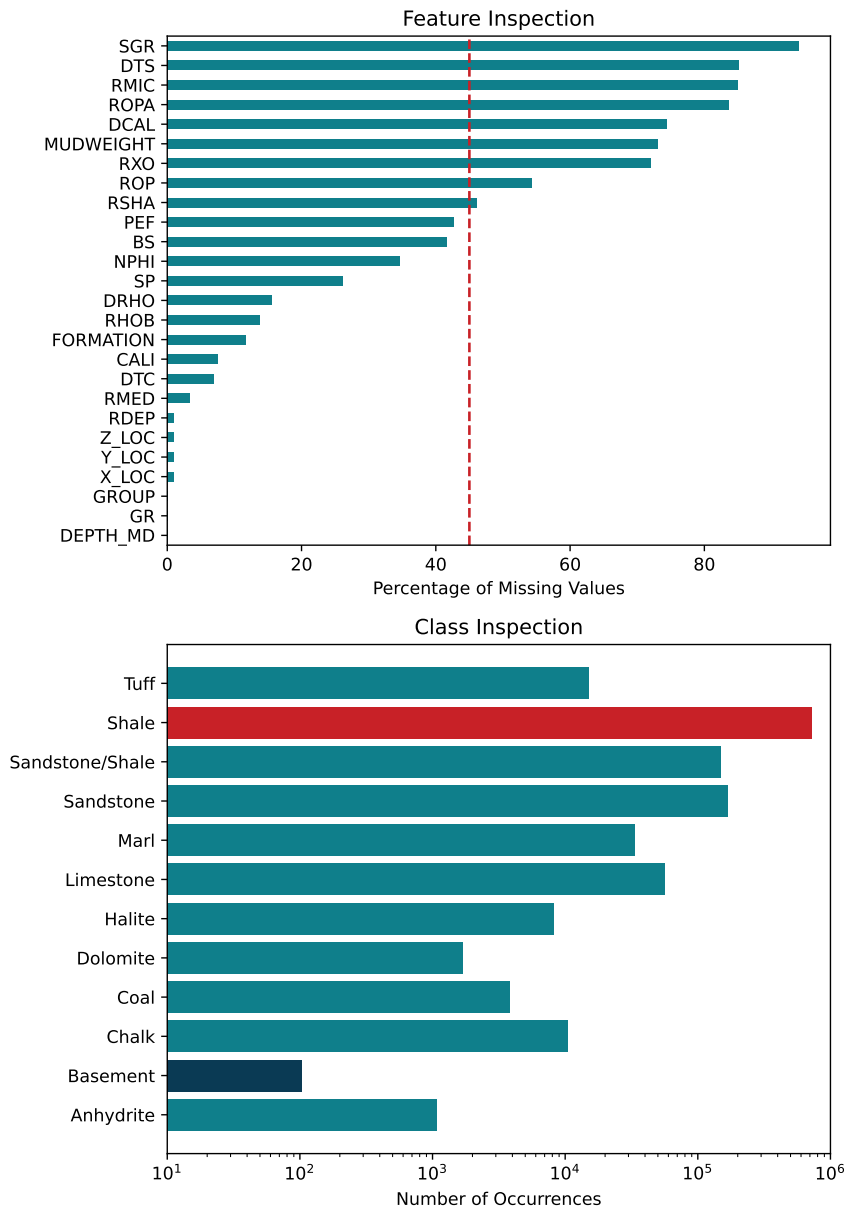


Fig. 8.2 Result of code listing 8.2. Inspect feature persistence and class balancing.

The second key characteristic of the investigated data set is straightforward when observing the class distribution (i.e., lower panel of Fig. 8.2): the training data set is highly imbalanced with some classes exceeding 10^5 occurrences and others, like Anhydrite and Basement only occurring 10^3 or 10^2 times, respectively. As strategy to account for the imbalance of training data set is also mandatory.

Some ML algorithms, such as the one reported in the present chapter, can try accounting the imbalanced nature of training data sets by tuning their hyper-parameters. More refined strategies may involve (a) under-sampling majority classes, (b) over-sampling minority classes, (c) combining over- and under-sampling methods, and (d) create ensemble balanced sets (Lemaître et al., 2017).

```

1 import numpy as np
2
3 fig = plt.figure(figsize=(8,4))
4
5 train_data['log_RDEP'] = np.log10(train_data['RDEP'])
6
7 to_be_plotted = ['RDEP', 'log_RDEP']
8
9 for index, my_feature in enumerate(to_be_plotted):
10     ax = fig.add_subplot(1,2,index+1)
11     min_val = np.nanpercentile(train_data[my_feature],1)
12     max_val = np.nanpercentile(train_data[my_feature],99)
13     my_bins = np.linspace(min_val,max_val,30)
14     ax.hist(train_data[my_feature], bins=my_bins,
15            density = True, color='#BFD7EA',
16            edgcolor='k')
17     ax.set_ylabel('Probability Density')
18     ax.set_xlabel(my_feature)
19
20 plt.tight_layout()

```

Listing 8.3 Inspect feature persistence and class balancing.

Looking at the histogram distribution of the selected features (code listing 8.3 and Fig. 8.3), it appears that selected features are highly skewed. It could be a problem for some ML algorithms, e.g., the ones assuming a normal distribution for the investigated features. As a consequence, I decided to apply a log-transformation to selected features to reduce the amount of skewness (e.g., Fig. 8.3, right panel).

As reported in section 3.3, data augmentation aims at increasing the generalization capability of ML models by increasing the amount of information in our data sets. It consists of adding modified copies (e.g., flipper or rotated images in the case of image classification) of the available data or combining the existing features to generate new ones. As an example, Bestagini et al. (2017) suggest three different approaches for data augmentation, i.e., applying quadratic expansion to the feature vector, considering second order interaction terms, and defining the augmented gradient feature vector. In the attempt to partially mimic the data augmentation strategy proposed by Bestagini

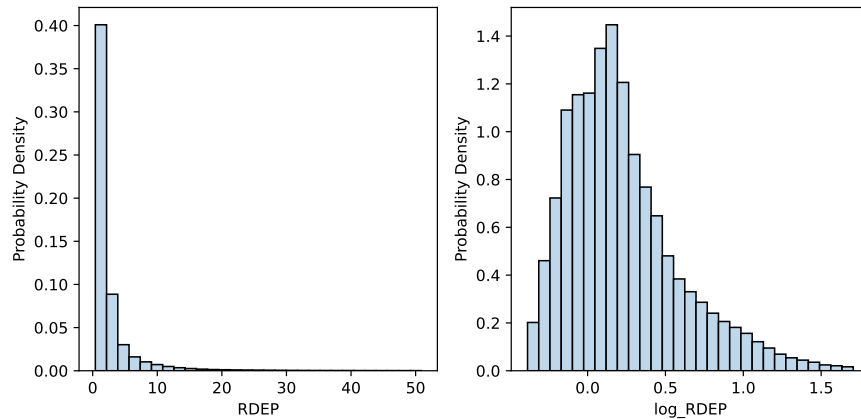


Fig. 8.3 Result of code listing 8.3. Log-transformation of selected features.

et al. (2017), I defined a function to calculate the augmented gradient feature vector (code listing 8.4):

```

1 def calculate_delta(dataFrame):
2     delta_features = ['CALI', 'log_RMED', 'log_RDEP', 'RHOB', '
3     DTC', 'DRHO', 'log_GR', 'NPHI', 'log_PEF', 'SP']
4     wells = dataFrame['WELL'].unique()
5     for my_feature in delta_features:
6         values = []
7         for well in wells:
8             col_values = dataFrame[dataFrame['WELL'] == well][
9             my_feature].values
10            col_values_ = np.array([col_values[0]]+list(
11            col_values[:-1]))
12            delta_col_values = col_values - col_values_
13            values = values + list(delta_col_values)
14            dataFrame['Delta_' + my_feature] = values
15     return dataFrame

```

Listing 8.4 Function to calculate the augmented gradient feature vector.

Summarizing, our pre-processing strategy starts with: (a) selecting the features characterized by missing values below 40%; (b) replacing missing values with the average value of each feature within each data set; (c) performing a log-transformation of the features showing a highly skewed distribution; (d) performing data augmentation. For the steps from a to d, I prepared a bunch of functions (code listing 8.5) and I combined them in a pandas *pipe()* chain to automate the pre-processing (code listing 8.6). Also, the *pre_processing_pipeline()* function (code listing 8.6) stores the imported .csv files in a single HDF5 file. As quickly introduced in the section 3.3, HDF5, i.e., Hierarchical Data Format version 5, is a high performance library to

manage, process, and store your heterogeneous data. Using the HDF5, I store all the data sets of interest as pandas DataFrames, ready for fast reading and writing (I/O). In detail, at lines 2 to 5, the function check if the output file exists. In this case, the function removes the existing file. Then, at line 15, it appends each processed data set to a newly created file.

```

1 import os
2 import pandas as pd
3 import numpy as np
4
5 def replace_inf(dataFrame):
6     to_be_replaced = [np.inf, -np.inf]
7     for replace_me in to_be_replaced:
8         dataFrame = dataFrame.replace(replace_me, np.nan)
9     return dataFrame
10
11 def log_transform(dataFrame):
12     log_features = ['RDEP', 'RMED', 'PEF', 'GR']
13     for my_feature in log_features:
14         dataFrame.loc[dataFrame[my_feature] < 0, my_feature] =
15             dataFrame[dataFrame[my_feature] > 0].RDEP.min()
16         dataFrame['log_' + my_feature] = np.log10(dataFrame[
17             my_feature])
18     return dataFrame
19
20 def calculate_delta(dataFrame):
21     delta_features = ['CALI', 'log_RMED', 'log_RDEP', 'RHOB',
22                     'DTC', 'DRHO', 'log_GR', 'NPHI',
23                     'log_PEF', 'SP']
24     wells = dataFrame['WELL'].unique()
25     for my_feature in delta_features:
26         values = []
27         for well in wells:
28             my_val = dataFrame[dataFrame['WELL'] == well][
29                 my_feature].values
30             my_val_ = np.array([my_val[0]] +
31                               list(my_val[:-1]))
32             delta_my_val = my_val - my_val_
33             values = values + list(delta_my_val)
34         dataFrame['Delta_' + my_feature] = values
35     return dataFrame
36
37 def feature_selection(dataFrame):
38     features = ['CALI', 'Delta_CALI', 'log_RMED',
39               'Delta_log_RMED', 'log_RDEP',
40               'Delta_log_RDEP', 'RHOB', 'Delta_RHOB',
41               'SP', 'Delta_SP', 'DTC', 'Delta_DTC',
42               'DRHO', 'Delta_DRHO', 'log_GR', 'Delta_log_GR',
43               'NPHI', 'Delta_NPHI', 'log_PEF', 'Delta_log_PEF']
44     dataFrame = dataFrame[features]
45     return dataFrame

```

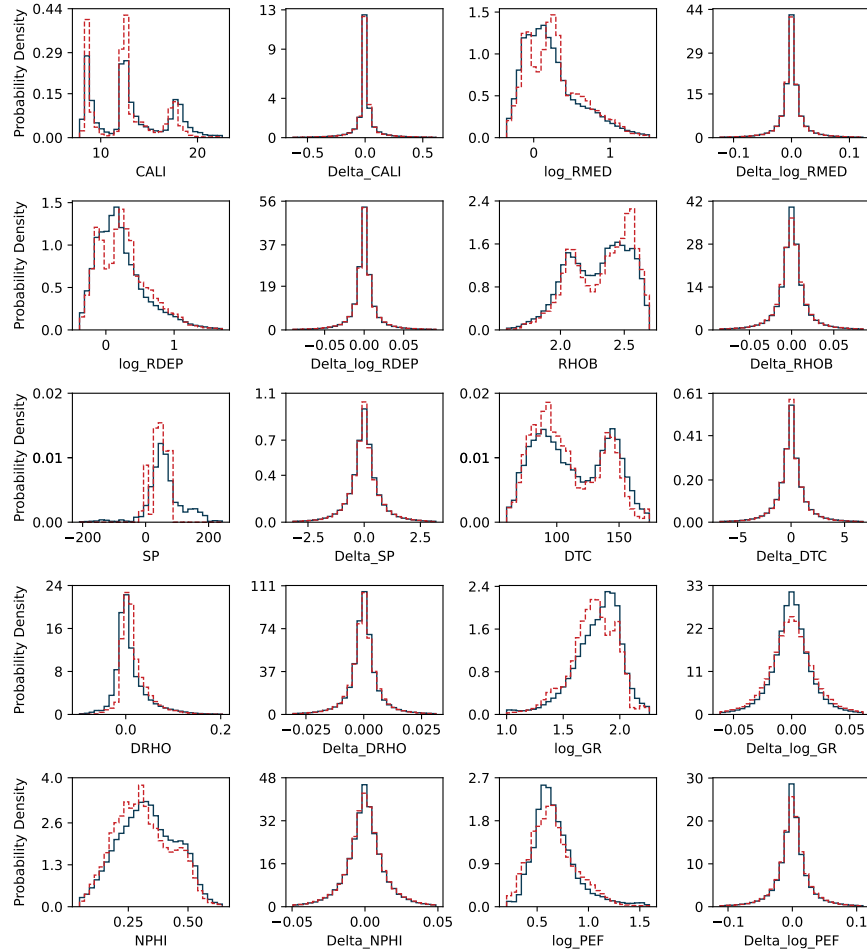
Listing 8.5 Defining the pre-processing functions.**Fig. 8.4** Result of code listing 8.7. Log-transformation of selected features.

Fig. 8.4 shows the results of code listing 8.7 and it describes most of the numerical features that we will use during the training. They derive by the application of the pre-processing strategy developed within code listings 8.5 and 8.6. All the features reported in Fig. 8.4 are of continuous numerical nature. However, the investigated data sets also contain categorical features, e.g., GROUP and FORMATIONS.

As reported in section 3.3, most of ML algorithms support the use of categorical features, but only after an encoding to their numerical counterparts. Code listing 8.8

reports the *pipe()* chain of code listing 8.6, i.e., *pre_processing_pipeline()*, with the addition of a categorical encoder to make the FORMATIONS ready for the investigation by a ML algorithm.

```

1 def pre_processing_pipeline(input_files, out_file):
2
3     try:
4         os.remove(out_file)
5     except OSError:
6         pass
7
8     for ix, my_file in enumerate(input_files):
9         my_dataset = pd.read_csv(my_file, sep=';')
10
11         try:
12             my_dataset['FORCE_2020_LITHOFACIES_LITHOLOGY'].to_hdf(
13 (
14                 out_file, key=my_file[:-4] + '_target')
15         except:
16             my_target = pd.read_csv('leaderboard_test_target.csv'
17 , sep=';')
18             my_target['FORCE_2020_LITHOFACIES_LITHOLOGY'].to_hdf(
19                 out_file, key=my_file[:-4] + '_target')
20
21         if ix==0:
22             # Fitting the categorical encoders
23             my_encoder = OrdinalEncoder()
24             my_encoder.set_params(handle_unknown='
25 use_encoded_value',
26                                 unknown_value=-1,
27                                 encoded_missing_value=-1).fit(
28                 my_dataset[['FORMATION']])
29
30         my_dataset = (my_dataset.
31             pipe(replace_inf).
32             pipe(log_transform).
33             pipe(calculate_delta).
34             pipe(feature_selection))
35         my_dataset.to_hdf(out_file, key=my_file[:-4])
36
37         my_dataset.to_hdf(out_file, key= my_file[:-4])
38
39 my_files = ['train.csv', 'leaderboard_test_features.csv', '
40 hidden_test.csv']
41
42 pre_processing_pipeline(input_files=my_files, out_file='ml_data.
43 h5')
```

Listing 8.6 Combining the pre-processing functions in a pandas *pipe()*.

In detail I used the *OrdinalEncoder()* method, available in scikit-learn. Also, Code listing 8.8 reports a modified version of the *feature_selection()* function to

include the encoded feature FORMATION. Now, since many ML algorithms are sensitive to differences in magnitude among the features and because they have been developed to work with standard (i.e., centered to zero and with unit variance) normally distributed data set, our pre-processing strategy only needs an additional step: data standardization. To standardize our features, I opted for the *StandardScaler()* method in scikit-learn. As reported in section 3.3, the *StandardScaler()* removes the mean for each feature and scales it to unit variance. Code listing 8.8, develops our final pre-processing strategy that includes data standardization. Then, the proposed pre-processing strategy is applied to the three main data sets utilized in the present chapter, i.e., `train`, `leaderboard_test`, and `hidden_test`, respectively (lines .

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 train_data = pd.read_hdf('ml_data.h5', 'train')
6 test_data = pd.read_hdf('ml_data.h5', 'leaderboard_test_features'
7 )
8
9 show_axes = [1,5,9,13,17]
10 fig = plt.figure(figsize=(9, 15))
11 for i, my_feature in enumerate(train_data.columns[0:20], start=1)
12 :
13     ax = fig.add_subplot(5,4,i)
14     min_val = np.nanpercentile(train_data[my_feature],1)
15     max_val = np.nanpercentile(train_data[my_feature],99)
16     my_bins = np.linspace(min_val,max_val,30)
17     ax.hist(train_data[my_feature], bins=my_bins, density = True,
18             histtype='step', color='#0A3A54')
19     ax.hist(test_data[my_feature], bins=my_bins, density = True,
20             histtype='step', color='#C82127', linestyle='--')
21     ax.set_xlabel(my_feature)
22     ymin, ymax = ax.get_ylim()
23     if ymax >=10:
24         ax.set_yticks(np.round(np.linspace(ymin, ymax, 4), 0))
25     elif ((ymax<10)&(ymax>1)):
26         ax.set_yticks(np.round(np.linspace(ymin, ymax, 4), 1))
27     else:
28         ax.set_yticks(np.round(np.linspace(ymin, ymax, 4), 2))
29
30     if i in show_axes:
31         ax.set_ylabel('Probability Density')
32
33 plt.tight_layout()
34 fig.align_ylabels()

```

Listing 8.7 Descriptive statistics.

```

1 import os
2 import pandas as pd
3 import numpy as np
4 from sklearn.preprocessing import OrdinalEncoder
5 from sklearn.impute import SimpleImputer
6
7 def replace_inf(dataFrame):
8     to_be_replaced = [np.inf, -np.inf]
9     for replace_me in to_be_replaced:
10        dataFrame = dataFrame.replace(replace_me, np.nan)
11    return dataFrame
12
13 def log_transform(dataFrame):
14    log_features = ['RDEP', 'RMED', 'PEF', 'GR']
15    for my_feature in log_features:
16        dataFrame.loc[dataFrame[my_feature] < 0, my_feature] =
17        dataFrame[
18            dataFrame[my_feature] > 0].RDEP.min()
19        dataFrame['log_' + my_feature] = np.log10(dataFrame[
20            my_feature])
21    return dataFrame
22
23 def calculate_delta(dataFrame):
24    delta_features = ['CALI', 'log_RMED', 'log_RDEP', 'RHOB',
25                    'DTC', 'DRHO', 'log_GR', 'NPHI',
26                    'log_PEF', 'SP', 'BS']
27    wells = dataFrame['WELL'].unique()
28    for my_feature in delta_features:
29        values = []
30        for well in wells:
31            my_val = dataFrame[dataFrame['WELL'] == well][
32                my_feature].values
33            my_val_ = np.array([my_val[0]] +
34                               list(my_val[:-1]))
35            delta_my_val = my_val - my_val_
36            values = values + list(delta_my_val)
37            dataFrame['Delta_' + my_feature] = values
38    return dataFrame
39
40 def categorical_encoder(dataFrame, my_encoder, cols):
41    dataFrame[cols] = my_encoder.transform(dataFrame[cols])
42    return dataFrame
43
44 def feature_selection(dataFrame):
45    features = ['CALI', 'Delta_CALI', 'log_RMED', '
46                Delta_log_RMED',
47                'log_RDEP', 'Delta_log_RDEP', 'RHOB', 'Delta_RHOB',
48                '
49                SP', 'Delta_SP', 'DTC', 'Delta_DTC', 'DRHO', '
50                Delta_DRHO',
51                'log_GR', 'Delta_log_GR', 'NPHI', 'Delta_NPHI',
52                'log_PEF', 'Delta_log_PEF', 'BS', 'Delta_BS',
53                'FORMATION', 'X_LOC', 'Y_LOC', 'DEPTH_MD']
54    dataFrame = dataFrame[features]

```

```

49     return dataframe
50
51 def pre_processing_pipeline(input_files, out_file):
52
53     try:
54         os.remove(out_file)
55     except OSError:
56         pass
57
58     for ix, my_file in enumerate(input_files):
59         my_dataset = pd.read_csv(my_file, sep=';')
60
61         try:
62             my_dataset['FORCE_2020_LITHOFACIES_LITHOLOGY'].to_hdf(
63 (
64                 out_file, key=my_file[:-4] + '_target')
65         except:
66             my_target = pd.read_csv('leaderboard_test_target.csv'
67 , sep=';')
68             my_target['FORCE_2020_LITHOFACIES_LITHOLOGY'].to_hdf(
69                 out_file, key=my_file[:-4] + '_target')
70
71         if ix==0:
72             # Fitting the categorical encoders
73             my_encoder = OrdinalEncoder()
74             my_encoder.set_params(handle_unknown='
75 use_encoded_value',
76                                     unknown_value=-1,
77                                     encoded_missing_value=-1).fit(
78                 my_dataset[['FORMATION']])
79
80         my_dataset = (my_dataset.
81             pipe(replace_inf).
82             pipe(log_transform).
83             pipe(calculate_delta).
84             pipe(categorical_encoder,
85                 my_encoder=my_encoder, cols=['
86 FORMATION'])).
87             pipe(feature_selection))
88         my_dataset.to_hdf(out_file, key=my_file[:-4])
89
90         imputer = SimpleImputer(missing_values=np.nan, strategy='
91 mean')
92         imputer = imputer.fit(my_dataset[my_dataset.columns])
93         my_dataset[my_dataset.columns] = imputer.transform(
94             my_dataset[my_dataset.columns])
95         my_dataset.to_hdf(out_file, key= my_file[:-4])
96
97 my_files = ['train.csv', 'leaderboard_test_features.csv', '
98 hidden_test.csv']
99
100 pre_processing_pipeline(input_files=my_files, out_file='ml_data.
101 h5')

```

Listing 8.8 Pre processing *pipe()* chain, including the categorical features.

8.3 Model Selection and Training

After data pre-processing, the next fundamental step is model selection, optimization and training. I remember the reader that we are dealing with a classification problem, therefore we will select among supervised algorithms. In the following we will test the *Extremely Randomized Trees* algorithm, i.e., *ExtraTreesClassifier()* in scikit-learn. Selecting the *ExtraTreesClassifier()* is an arbitrary choice and I invite the reader to also explore different ML methods like, such an example, Support Vector Machines.

In our specific case, the *ExtraTreesClassifier()* depends on many hyper-parameters like, e.g., the number of trees, the number of investigated features, and the criterion of splitting.

```
1 import pandas as pd
2 from sklearn.ensemble import ExtraTreesClassifier
3 from sklearn.model_selection import train_test_split
4 from sklearn.model_selection import GridSearchCV
5 import joblib as jb
6 from sklearn.preprocessing import StandardScaler
7
8 X = pd.read_hdf('ml_data.h5', 'train').values
9 y = pd.read_hdf('ml_data.h5', 'train_target').values
10
11 X_train, X_test, y_train, y_test = train_test_split(
12     X, y, test_size=0.2, random_state=10, stratify=y)
13
14 scaler = StandardScaler()
15 X_train = scaler.fit_transform(X_train)
16
17 param_grid = {
18     'criterion': ['entropy', 'gini'],
19     'min_samples_split': [2, 5, 8, 10],
20     'max_features': ['sqrt', 'log2', None],
21     'class_weight': ['balanced', None]
22 }
23
24 classifier = ExtraTreesClassifier(n_estimators=250,
25                                 n_jobs=-1)
26
27 CV_rfc = GridSearchCV(estimator=classifier, param_grid=param_grid
28                       , cv=3, verbose=10)
29 CV_rfc.fit(X_train, y_train)
30 jb.dump(CV_rfc, 'ETC_grid_search_results_rev_2.pkl')
```

Listing 8.9 Grid search using *GridSearchCV()*.**Table 8.1** Hyper-parameters utilized in the grid search to optimize the *ExtraTreesClassifier()* algorithm. Descriptions are from scikit-learn documentation.

parameter	Description ⁶	values
criterion	The function to measure the quality of a split.	['entropy', 'gini']
min_samples_split	The minimum number of samples required to split an internal node	[2, 5, 8, 10]
max_features	The number of features to consider when looking for the best split	['sqrt', 'log2', None]
class_weight	Weights associated with classes	['balanced', None]

All these hyper-parameters may assume different values which may positively or negatively affect the classification capability of the model. To find the best combination for the investigated hyper-parameters, the easiest approach is to perform a grid search. It consists of defining the most relevant values for each hyper-parameter, then training and evaluating the models resulting for all possible combinations. Table 8.1 reports the hyper-parameters I selected for the grid search. Also, Table 8.1 highlights the values that I selected for the grid search. To perform the grid search in python, I used the *GridSearchCV()* method in scikit-learn (code listing 8.9). In detail, after the importing of all required libraries (lines from 1 to 6), in code listing 8.9, I import the pre-processed training data set (line 8) with labels (line 9). Then, I split the training data set into two, i.e., a portion for the training and validation within the grid search, i.e., X_{train} , and a portion, never involved during the training, i.e., X_{test} , to test the results obtained during the grid search and further testing against potential issues like the over-fitting. The next step (lines 14 and 15) consists of scaling the data set involved in the grid search to zero mean and unit variance. From lines 17 to 22, I defined the set of parameters involved in the grid search. The combination of the selected hyper-parameters results in a grid of 48 models, each repeated three times ($cv=3$ at line 27) through cross validation (see section 3.5) for a total of 144 attempts. Running the code listing 8.9 in my MacBook pro, equipped with a 2.3 GHz Quad-Core Intel™ Core i7 and 32GB of RAM, lasted about 8 hours. The top panel of Fig. 8.5 displays the accuracy scores of all the 48 models, ordered by their ranking (code listing 8.10), and it highlights that best performing models achieve accuracy scores larger than 0.95. The achievement of such high performances may suggest that we are over-fitting the training data set, therefore, as a first step, I used the three best performing models (code listing 8.10) on the test data set, i.e., X_{test} . The bottom panel of Fig. 8.5 highlights that the accuracy scores on X_{test} are of the same order of those resulting from the grid search cross validation, i.e. ~ 0.96 , not supporting the idea of strong over-fitting.

Also, I run the three best performing models to predict our unknowns samples, i.e., the leaderboard and the hidden test data sets, respectively. Looking at the accuracy scores (Fig. 8.6) on the leaderboard and hidden test data sets, i.e. from 0.79 to 0.81 as accuracy scores, we highlight that our ML models still perform in a satisfactory fashion on independent test data sets, and therefore we move to the next section where we will check our models in light of the evaluation criteria of the FORCE2020 challenge.

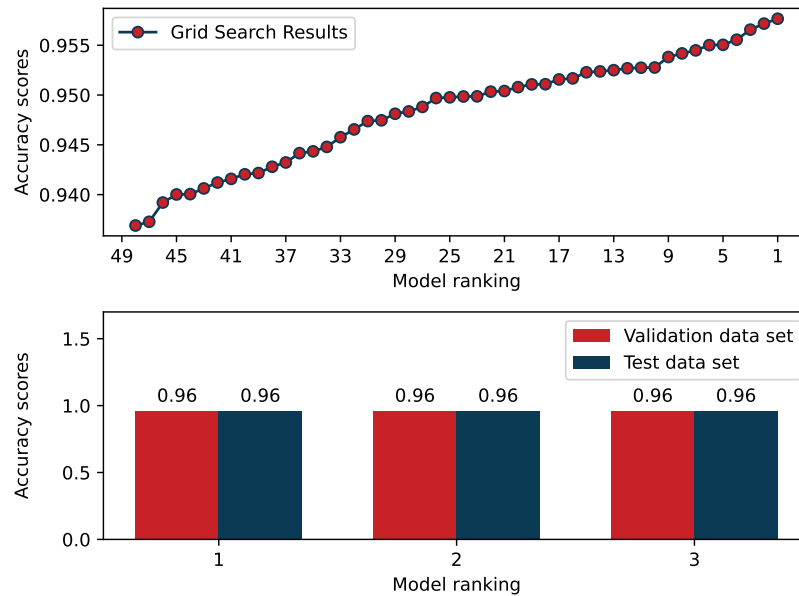


Fig. 8.5 Result of code listing 8.10.

```

1 from joblib import load
2 import numpy as np
3 import matplotlib.pyplot as plt
4 import pandas as pd
5 from sklearn.ensemble import ExtraTreesClassifier
6 from sklearn.model_selection import train_test_split
7 from sklearn.preprocessing import StandardScaler
8
9 CV_rfc = load('ETC_grid_search_results_rev_2.pkl')
10
11 my_results = pd.DataFrame.from_dict(CV_rfc.cv_results_)
12 my_results = my_results.sort_values(by=['rank_test_score'])
13
14 # Plot the results of the GridSearch
15 fig = plt.figure()

```

```

16 ax1 = fig.add_subplot(2,1,1)
17 ax1.plot(my_results['rank_test_score'], my_results['
    mean_test_score'], marker='o',
18         markeredgcolor='#0A3A54', markerfacecolor='#C82127',
    color='#0A3A54',
19         label='Grid Search Results')
20 ax1.set_xticks(np.arange(1,50,4))
21 ax1.invert_xaxis()
22 ax1.set_xlabel('Model ranking')
23 ax1.set_ylabel('Accuracy scores')
24 ax1.legend()
25
26 # Selecting the best three performing models
27 my_results = my_results[my_results['mean_test_score']>0.956]
28
29 # Load and scaling
30 X = pd.read_hdf('ml_data.h5', 'train').values
31 y = pd.read_hdf('ml_data.h5', 'train_target').values
32
33 X_train, X_test, y_train, y_test = train_test_split(
34     X, y, test_size=0.2, random_state=10, stratify=y)
35
36 scaler = StandardScaler()
37 X_train = scaler.fit_transform(X_train)
38 X_test = scaler.transform(X_test)
39
40 leaderboard_test_features = pd.read_hdf('ml_data.h5', '
    leaderboard_test_features').values
41 hidden_test = pd.read_hdf('ml_data.h5', 'hidden_test').values
42
43 leaderboard_test_features_scaled = scaler.transform(
    leaderboard_test_features)
44 hidden_test_scaled = scaler.transform(hidden_test)
45
46 # Apply the three best performing model on the test dataset and
    on the unknowns
47 leaderboard_test_res = {}
48 hidden_test_res = {}
49 test_score = []
50 rank_model = []
51 for index, row in my_results.iterrows():
52     classifier = ExtraTreesClassifier(n_estimators=250, n_jobs=8,
    random_state=64, **row['params'])
53     classifier.fit(X_train, y_train)
54     my_score = classifier.score(X_test, y_test)
55     test_score.append(my_score)
56     rank_model.append(row['rank_test_score'])
57
58     my_leaderboard_test_res = classifier.predict(
    leaderboard_test_features_scaled)
59     my_hidden_test_res = classifier.predict(hidden_test_scaled)
60     leaderboard_test_res['model_ranked_' + str(row['
    rank_test_score'])] = my_leaderboard_test_res

```

```

61 hidden_test_res['model_ranked_' + str(row['rank_test_score'])
    ] = my_hidden_test_res
62
63 leaderboard_test_res_pd = pd.DataFrame.from_dict(
    leaderboard_test_res)
64 hidden_test_res_pd = pd.DataFrame.from_dict(hidden_test_res)
65 leaderboard_test_res_pd.to_hdf('ml_data.h5', key= '
    leaderboard_test_res')
66 hidden_test_res_pd.to_hdf('ml_data.h5', key= 'hidden_test_res')
67
68 # plot the resultson the test dataset
69 ax2 = fig.add_subplot(2,1,2)
70 labels = my_results['rank_test_score']
71 validation_res = np.around(my_results['mean_test_score'], 2)
72 test_res = np.around(np.array(test_score),2)
73 x = np.arange(len(labels))
74 width = 0.35
75 rects1 = ax2.bar(x - width/2, validation_res, width, label='
    Validation data set', color='#C82127')
76 rects2 = ax2.bar(x + width/2, test_res, width, label='Test data
    set', color='#0A3A54')
77 ax2.set_ylabel('Accuracy scores')
78 ax2.set_xlabel('Model ranking')
79 ax2.set_ylim(0,1.7)
80 ax2.set_xticks(x, labels)
81 ax2.legend()
82 ax2.bar_label(rects1, padding=3)
83 ax2.bar_label(rects2, padding=3)
84 fig.align_ylabels()
85 fig.tight_layout()

```

Listing 8.10 Applying the three best performing model on the test data set and on unknown samples.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 from sklearn.metrics import accuracy_score
4 import pandas as pd
5
6 leaderboard_test_res= pd.read_hdf('ml_data.h5', '
    leaderboard_test_res')
7 hidden_test_res = pd.read_hdf('ml_data.h5', 'hidden_test_res')
8
9 leaderboard_test_target = pd.read_hdf('ml_data.h5', '
    leaderboard_test_features_target').values
10 hidden_test_target = pd.read_hdf('ml_data.h5', '
    hidden_test_target').values
11
12 leaderboard_accuracy_scores = []
13 hidden_accuracy_scores = []
14

```

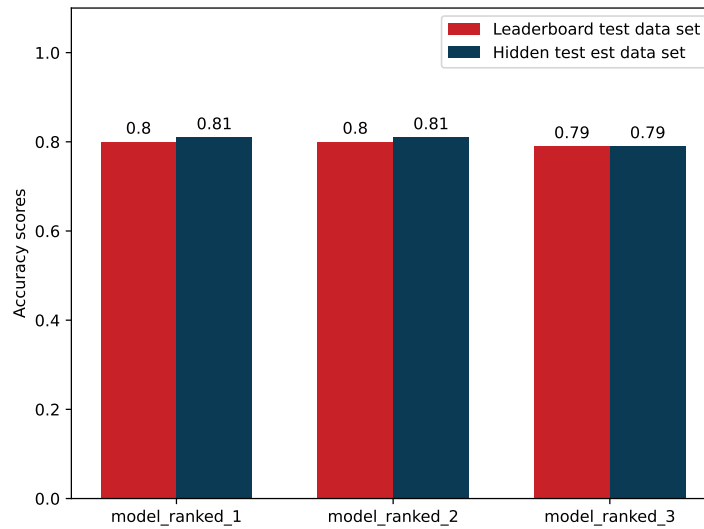



Fig. 8.6 Result of code listing 8.11.

```

15 for (leaderboard_column, leaderboard_data), (hidden_column,
    hidden_data) in zip(leaderboard_test_res.iteritems(),
    hidden_test_res.iteritems()):
16
17     leaderboard_accuracy_scores.append(np.around(accuracy_score(
    leaderboard_data, leaderboard_test_target),2))
18     hidden_accuracy_scores.append(np.around(accuracy_score(
    hidden_data, hidden_test_target),2))
19
20
21 # plot the resultson the test dataset
22 plt, ax1 = plt.subplots()
23 labels = leaderboard_test_res.columns
24 x = np.arange(len(labels))
25 width = 0.35
26 rects1 = ax1.bar(x - width/2, leaderboard_accuracy_scores, width,
    label='Leaderboard test data set', color='#C82127')
27 rects2 = ax1.bar(x + width/2, hidden_accuracy_scores, width,
    label='Hidden test est data set', color='#0A3A54')
28 ax1.set_ylabel('Accuracy scores')
29 #ax1.set_xlabel('Model ranking')
30 ax1.set_ylim(0,1.1)
31 ax1.set_xticks(x, labels)
32 ax1.legend()
33 ax1.bar_label(rects1, padding=3)
34 ax1.bar_label(rects2, padding=3)

```

Listing 8.11 Plotting the results obtained on the Leaderboard and on the hidden test data sets.

8.4 Final evaluation

To evaluate the goodness of each model, the FORCE2020 challenge utilized a custom scoring strategy based no penalty matrix (code listing 8.12).

```

1 import numpy as np
2
3 A = np.load('penalty_matrix.npy')
4 def score(y_true, y_pred):
5     S = 0.0
6     y_true = y_true.astype(int)
7     y_pred = y_pred.astype(int)
8     for i in range(0, y_true.shape[0]):
9         S -= A[y_true[i], y_pred[i]]
10    return S/y_true.shape[0]

```

Listing 8.12 Custom scoring function.

In code listing 8.12, *y_true* and *y_pred* are the expected, i.e., correct, and the predicted values, respectively, converted into integer indexes ranging from 0 to 11, as reported in Table 8.2.

Table 8.2 connecting the labeling in the target files with lithofacies names and the indexing of the score function

Label	Lithofacies	Index
30000	'Sandstone'	0
65030	'Sandstone/Shale'	1
65000	'Shale'	2
80000	'Marl'	3
74000	'Dolomite'	4
70000	'Limestone'	5
70032	'Chalk'	6
88000	'Halite'	7
86000	'Anhydrite'	8
99000	'Tuff'	9
90000	'Coal'	10
93000	'Basement'	11

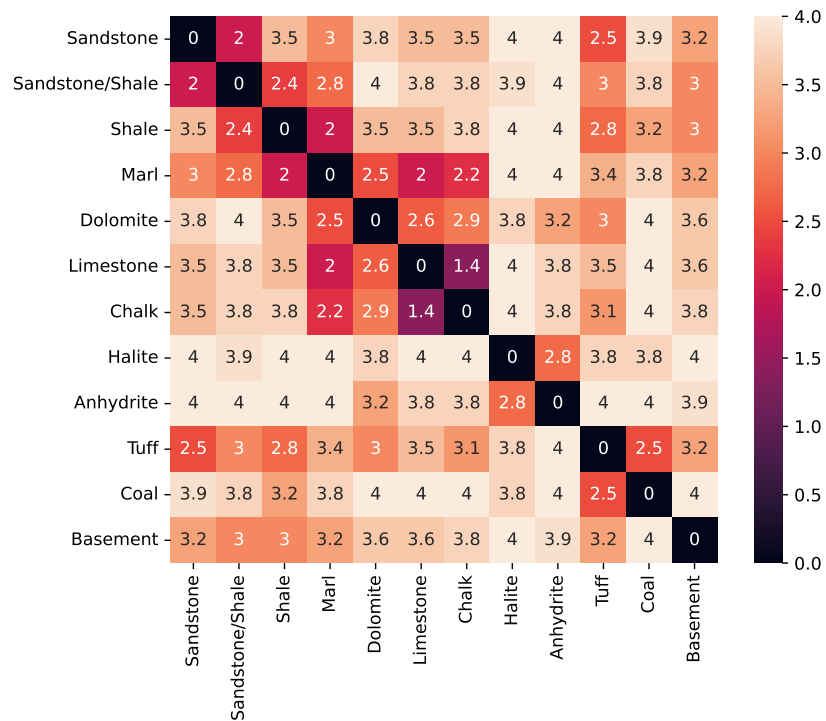


Fig. 8.7 Result of code listing 8.13.

```

1 import numpy as np
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 A = np.load('penalty_matrix.npy')
6
7 my_labels = ['Sandstone', 'Sandstone/Shale', 'Shale', 'Marl', '
8             'Dolomite',
9             'Limestone', 'Chalk', 'Halite', 'Anhydrite', 'Tuff', '
10            'Coal', 'Basement']
11
12 fig, ax = plt.subplots(figsize=(15, 12))
13 ax.imshow(A)
14 ax = sns.heatmap(A, annot=True, xticklabels = my_labels,
15                 yticklabels = my_labels)
16 fig.tight_layout()

```

Listing 8.13 the penalty matrix.

The main objective is to strongly penalize the errors made on easy to recognize lithologies, and than those on lithologies that are difficult. To achieve this goal, the

`score()` function weight each true-valueprediction pair using the penalty matrix (code listing 8.13) reported in Fig. 8.7. In detail, the `score()` function return the value of the penalty matrix corresponding to each true-valueprediction pair (e.g., 4 if you confuse a Halite for a Sandtone; Fig. 8.7). Then, it sums all the scoring values and it finally calculates an ‘average’ score dividing the resulting value by the number of predictions.

```

1 import numpy as np
2 import pandas as pd
3
4 A = np.load('penalty_matrix.npy')
5 def score(y_true, y_pred):
6     S = 0.0
7     y_true = y_true.astype(int)
8     y_pred = y_pred.astype(int)
9     for i in range(0, y_true.shape[0]):
10        S -= A[y_true[i], y_pred[i]]
11    return S/y_true.shape[0]
12
13 target = np.full(1000, 5) # Limestone
14 predicted = np.full(1000, 5) # Limestone
15 print("Case 1: " + str(score(target, predicted)))
16
17 predicted = np.full(1000, 6) # Chalk
18 print("Case 2: " + str(score(target, predicted)))
19
20 predicted = np.full(1000, 7) # Halite
21 print("Case 3: " + str(score(target, predicted)))
22
23 hidden_test_target = pd.read_hdf('ml_data.h5',
24                                 'hidden_test_target').values
25 predicted = np.random.randint(0, high=12,
26                               size=1000) # Random predictions
27 print("Case 4: " + str(score(target, predicted)))
28
29 ''' Output:
30
31 Case 1: 0.0
32 Case 2: -1.375
33 Case 3: -4.0
34 Case 4: -3.04625
35
36 '''

```

Listing 8.14 Custom scoring function.

Looking at Fig. 8.7 and code listing 8.12, we can argue that when the prediction is correct, the contribution to the score is equal to zero. Therefore, if you correctly guess all the predictions, the score function return a value equal to zero (e.g., code listing 8.14 - Case 1). On the contrary, systematically predicting Chalk on a data

set of limestone samples will return -1.375 (code listing 8.14 - Case 2). Finally, systematically predicting Halyte on a data set of limestone samples will return -4.0 (code listing 8.14 - Case 3), much more penalized than the Case 2. Finally, considering the hidden test data set, a dummy model providing random predictions, will obtain a score close to -3 (code listing 8.14 - Case 4).

Fig. 8.8 displays the application of the scoring strategy described above on the leaderboard and hidden test, respectively. Despite the simplicity of the model, the best two performing models derived by the grid search of code listing 8.9 are comparable with those on the top three (<-0.50) of the FORCE2020 challenge⁷ on the hidden test data set. For the leaderboard test data set, the results are still satisfactory. However, they should rank within the 30th and the 40th position.

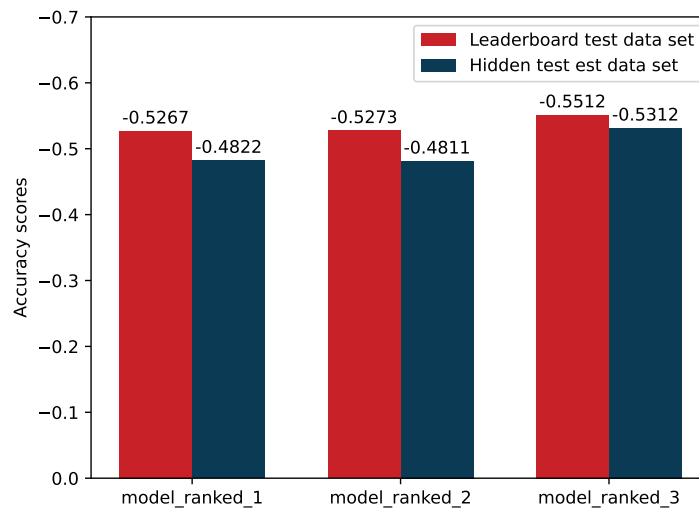


Fig. 8.8 Result of code listing 8.15.

```

1 import matplotlib.pyplot as plt
2 import pandas as pd
3 import numpy as np
4
5 A = np.load('penalty_matrix.npy')
6 def score(y_true, y_pred):
7     S = 0.0
8     y_true = y_true.astype(int)
9     y_pred = y_pred.astype(int)

```

⁷ <https://github.com/bolgebrygg/Force-2020-Machine-Learning-competition>

```

10     for i in range(0, y_true.shape[0]):
11         S -= A[y_true[i], y_pred[i]]
12     return S/y_true.shape[0]
13
14 lithology_numbers = {30000: 0, 65030: 1, 65000: 2, 80000: 3,
15                     74000: 4, 70000: 5,
16                     70032: 6, 88000: 7, 86000: 8, 99000: 9,
17                     90000: 10, 93000: 11}
18
19 # Load test data
20 leaderboard_test_res = pd.read_hdf('ml_data.h5', '
21     leaderboard_test_res')
22 hidden_test_res = pd.read_hdf('ml_data.h5', 'hidden_test_res')
23
24 leaderboard_test_target = pd.read_hdf('ml_data.h5', '
25     leaderboard_test_features_target').values
26 leaderboard_test_target = np.vectorize(lithology_numbers.get)(
27     leaderboard_test_target)
28 hidden_test_target = pd.read_hdf('ml_data.h5', '
29     hidden_test_target').values
30 hidden_test_target = np.vectorize(lithology_numbers.get)(
31     hidden_test_target)
32
33 leaderboard_accuracy_scores = []
34 hidden_accuracy_scores = []
35 for (leaderboard_column, leaderboard_data), (hidden_column,
36     hidden_data) in zip(leaderboard_test_res.iteritems(),
37     hidden_test_res.iteritems()):
38
39     leaderboard_data = np.vectorize(lithology_numbers.get)(
40     leaderboard_data)
41     leaderboard_accuracy_scores.append(np.around(score(
42     leaderboard_data, leaderboard_test_target),4))
43     hidden_data = np.vectorize(lithology_numbers.get)(
44     hidden_data)
45     hidden_accuracy_scores.append(np.around(score(hidden_data,
46     hidden_test_target),4))
47
48 # plot the results
49 plt, ax1 = plt.subplots()
50 labels = leaderboard_test_res.columns
51 x = np.arange(len(labels))
52 width = 0.35
53 rects1 = ax1.bar(x - width/2, leaderboard_accuracy_scores, width,
54     label='Leaderboard test data set', color='#C82127')
55 rects2 = ax1.bar(x + width/2, hidden_accuracy_scores, width,
56     label='Hidden test est data set', color='#0A3A54')
57 ax1.set_ylabel('Accuracy scores')
58 ax1.set_ylim(0,-0.7)
59 ax1.set_xticks(x, labels)
60 ax1.legend()
61 ax1.bar_label(rects1, padding=-12)
62 ax1.bar_label(rects2, padding=-12)

```

Listing 8.15 Final scoring on the leaderbord and hidden test data set

Chapter 9

Machine Learning Regression in Petrology

9.1 Motivation

Dechypering magma storage depths and temperatures in feeding systems of active volcanoes is a central issue in volcanology and petrology (e.g., Putirka, 2008). As an example, the depiction of magma storage depths helps the characterization of volcanic plumbing systems (e.g., Petrelli et al., 2018; Ubide and Kamber, 2018; Ubide et al., 2021). Also, magma temperature estimates are mandatory for the application of diffusion-based geo-chronometers (e.g., Costa et al., 2020). To date, the development of a geo-barometer or a geo-thermometer is mainly based on entropy and volume changes during equilibrium reactions between melts and crystals (Putirka, 2008). It is a robust and widely applied strategy (Putirka, 2008, and references therein). As an example, the calibration process for a cpx thermometer or a barometer consists of five main steps: (a) determine chemical equilibria associated with significant changes in entropy and volume, respectively (Putirka, 2008); (b) get a suitable experimental data set where T and P are known (e.g., the LEPR data set; Hirschmann et al., 2008); (c) compute the components of the crystal phase from chemical analyses; (d) choose the regression strategy; and finally (e) validate your model (Putirka, 2008). Recently, many authors demonstrated the potential of machine learning (ML) thermo-barometry (e.g., Jorgenson et al., 2022; Petrelli et al., 2020). Differently from Petrelli et al. (2020), I will focus on orthopyroxenes in equilibrium with the melt phase and orthopyroxenes alone.

9.2 The LEPR data set and data pre processing

LEPR (Hirschmann et al., 2008) is the acronym of the Library of Experimental Phase Relations. It includes a large number of petrological experiments (>5000) simulating igneous systems at temperatures between 500 and 2500C and pressures up to >25

GPa. The LEPR data set can be easily downloaded from a dedicated portal¹. The entries corresponding to each experiment in the data set include both experimental data (i.e., the composition of starting materials, the temperature and pressure of the experiments, the phases present at the end of the experiments and related compositions) and metadata (e.g., author, laboratory, device, oxygen fugacity, etc...). From the LEPR portal, I downloaded an ExcelTM file, i.e., LEPR_download.xls, containing all the experiments. Within the ExcelTM file, the sheet named 'Experiments' contains all meta data and relevant information like the composition of starting materials, the temperature and pressure of the experiments, and the phases present at the end of the experiments. The sheets named with the name of a phase (e.g., Liquid, Clinopyroxene, Olivine, etc...) contain the chemical composition for that specific phase in the different experiments. An index characterizes each experiment, linking the information in the different sheets.

As pre-processing strategy, I decided to proceed as follow (code listing 9.1): 1) define all the required functions (lines from 5 to 95); 2) define a function for the pre-processing (i.e., *data_pre_processing()*) (lines from 98 to 161). The *data_pre_processing()* function includes: a) the import of the LEPR data set from ExcelTM (lines 11 and 121), 2) create a pandas *pipe()* for basic operations like adjusting column names, convert all Iron data as $Fe_{o_{tot}}$, filtering the features, and imputing NaN to zero (lines from 124 to 124); 3) start storing phase information in a .hd5 file (line 132, 134, and 135); 4) combine all relevant data in a single pandas DataFrame (lines from 137 to 139); 5) make some filtering based on SiO_2 , pressure [$P(GPa)$], and temperature [$T(C)$], respectively (lines from 141 to 147); 6) remove the entries characterized by chemical analysis that don't fit the chemical formula of the orthopyroxene (lines from 149 to 151) 7) shuffle the data set (lines 153-154); 7) separate the labels from the input features (lines 156-157); 8) store everything in a .hd5 file (lines 159-160).

The statement starting at line 163 triggers the data pre-processing to develop a thermometer or a barometer based on the liquid-orthopyroxene equilibrium. The result is a hdf5 file named ml_data.h5 containing a DataFrame named 'Liquid_Orthopyroxene' containing the pre-processed experimental data from LEPR. Also, it stores the labels, i.e., T and P, in DataFrame named 'labels'. Finally, it contains all the original data of our interest in three DataFrames named 'Liquid', 'Orthopyroxene', and 'starting_material', respectively.

Figs. 9.1 and 9.2 show the probability densities for the different chemical elements in the melt (Fig. 9.1) and orthopyroxene (Fig. 9.2) phases, respectively, based on kernel densities estimates (Code listing 9.2). In detail, the code listing 9.2 imports 'Liquid_Orthopyroxene' DataFrame from the hdf5 file named ml_data.h5 (line 5). Then it plots two figures (i.e., Fig. 9.1) and Fig. 9.2) to highlight the univariate distribution of each feature that we are going to 'squeeze' to develop our ML model. To note, the investigated features are of compositional nature (i.e., they always close to 100 wt.%), a significant characteristic that deserves additional discussion.

¹ <https://lepr.earthchem.org/>

```

1 import os
2 import pandas as pd
3 import numpy as np
4
5 Elements = {
6     'Liquid': ['SiO2', 'TiO2', 'Al2O3', 'FeOtot', 'MgO',
7               'MnO', 'CaO', 'Na2O', 'K2O'],
8     'Orthopyroxene': ['SiO2', 'TiO2', 'Al2O3', 'FeOtot',
9                       'MgO', 'MnO', 'CaO', 'Na2O', 'Cr2O3']}
10
11 def calculate_cations_on_oxygen_basis(
12     myData0, myphase, myElements, n_oxygens):
13
14     Weights = {'SiO2': [60.0843, 1.0, 2.0],
15               'TiO2': [79.8788, 1.0, 2.0],
16               'Al2O3': [101.961, 2.0, 3.0],
17               'FeOtot': [71.8464, 1.0, 1.0],
18               'MgO': [40.3044, 1.0, 1.0],
19               'MnO': [70.9375, 1.0, 1.0],
20               'CaO': [56.0774, 1.0, 1.0],
21               'Na2O': [61.9789, 2.0, 1.0],
22               'K2O': [94.196, 2.0, 1.0],
23               'Cr2O3': [151.9982, 2.0, 3.0],
24               'P2O5': [141.937, 2.0, 5.0],
25               'H2O': [18.01388, 2.0, 1.0]}
26
27     myData = myData0.copy()
28     # Cation mole proportions
29     for el in myElements:
30         myData[el + '_cat_mol_prop'] = myData[myphase +
31                                               '_' + el] * Weights[el][1] / Weights[el][0]
32     # Oxygen mole proportions
33     for el in myElements:
34         myData[el + '_oxy_mol_prop'] = myData[myphase +
35                                               '_' + el] * Weights[el][2] / Weights[el][0]
36     # Oxygen mole proportions totals
37     totals = np.zeros(len(myData.index))
38     for el in myElements:
39         totals += myData[el + '_oxy_mol_prop']
40     myData['tot_oxy_prop'] = totals
41     # totcations
42     totals = np.zeros(len(myData.index))
43     for el in myElements:
44         myData[el + '_num_cat'] = n_oxygens * myData[el +
45                                                       '_cat_mol_prop'] / myData['tot_oxy_prop']
46         totals += myData[el + '_num_cat']
47     return totals
48
49 def filter_by_cryst_formula(dataFrame, myphase, myElements):
50
51     c_o_Tolerance = {'Orthopyroxene': [4, 6, 0.025]}
52
53     dataFrame['Tot_cations'] = calculate_cations_on_oxygen_basis(
54         myData0 = dataFrame, myphase = myphase,

```

```

55     myElements = myElements ,
56     n_oxygens = c_o_Tolerance[myphase][1])
57
58     dataframe = dataframe[
59         (dataframe['Tot_cations'] < c_o_Tolerance[myphase][0]
60          + c_o_Tolerance[myphase][2]) &
61         (dataframe['Tot_cations'] > c_o_Tolerance[myphase][0]
62          - c_o_Tolerance[myphase][2])]
63
64     dataframe = dataframe.drop(columns=['Tot_cations'])
65     return dataframe
66
67 def adjustFeOtot(dataframe):
68     for i in range(len(dataframe.index)):
69         try:
70             if pd.to_numeric(dataframe.Fe2O3[i])>0:
71                 dataframe.loc[i, 'FeOtot'] = (
72                     pd.to_numeric(dataframe.FeO[i]) + 0.8998 *
73                     pd.to_numeric(dataframe.Fe2O3[i]))
74             else:
75                 dataframe.loc[i,
76                             'FeOtot'] = pd.to_numeric(dataframe.FeO[i])
77         except:
78             dataframe.loc[i, 'FeOtot'] = 0
79     return dataframe
80
81 def adjust_column_names(dataframe):
82     dataframe.columns = [c.replace('Wt: ', '')
83                          for c in dataframe.columns]
84     dataframe.columns = [c.replace(' ', '')
85                          for c in dataframe.columns]
86     return dataframe
87
88 def select_base_features(dataframe, my_elements):
89     dataframe = dataframe[my_elements]
90     return dataframe
91
92 def data_imputation(dataframe):
93     dataframe = dataframe.fillna(0)
94     return dataframe
95
96 def data_pre_processing(phase_1, phase_2, out_file):
97
98     try:
99         os.remove(out_file)
100    except OSError:
101        pass
102
103    starting = pd.read_excel('LEPR_download.xls',
104                           sheet_name='Experiment')
105    starting= adjust_column_names(starting)
106    starting.name = ''
107    starting = starting[['Index', 'T(C)', 'P(GPa)']]
108    starting.to_hdf(out_file, key='starting_material')

```

```

109
110 phases = [phase_1, phase_2]
111
112 for ix, my_phase in enumerate(phases):
113     my_dataset = pd.read_excel('LEPR_download.xls',
114                               sheet_name = my_phase)
115     my_dataset = (my_dataset.
116                  pipe(adjust_column_names).
117                  pipe(adjustFeOtot).
118                  pipe(select_base_features,
119                        my_elements= Elements[my_phase]).
120                  pipe(data_imputation))
121
122     my_dataset = my_dataset.add_prefix(my_phase + '_')
123     my_dataset.to_hdf(out_file, key=my_phase)
124
125 my_phase_1 = pd.read_hdf(out_file, phase_1)
126 my_phase_2 = pd.read_hdf(out_file, phase_2)
127
128 my_dataset = pd.concat([starting,
129                         my_phase_1,
130                         my_phase_2], axis=1)
131
132 my_dataset = my_dataset[(my_dataset['Liquid_SiO2'] > 35)&
133                        (my_dataset['Liquid_SiO2'] < 80)]
134
135 my_dataset = my_dataset[(
136     my_dataset['Orthopyroxene_SiO2'] > 0)]
137
138 my_dataset = my_dataset[(my_dataset['P(GPa)'] <= 2)]
139
140 my_dataset = my_dataset[(my_dataset['T(C)'] >= 650)&
141                        (my_dataset['T(C)'] <= 1800)]
142
143 my_dataset = filter_by_cryst_formula(dataFrame = my_dataset,
144                                     myphase = phase_2,
145                                     myElements = Elements[phase_2])
146
147 my_dataset = my_dataset.sample(frac=1,
148                               random_state=50).reset_index(drop=True)
149
150 my_labels = my_dataset[['Index', 'T(C)', 'P(GPa)']]
151 my_dataset = my_dataset.drop(columns=['T(C)', 'P(GPa)'])
152
153 my_labels.to_hdf(out_file, key='labels')
154 my_dataset.to_hdf(out_file,
155                  key= phase_1 + '_' + phase_2)
156
157 data_pre_processing(phase_1='Liquid' ,
158                   phase_2='Orthopyroxene' ,
159                   out_file='ml_data.h5')

```

Listing 9.1 Implementation of our pre-processing strategy.

```

1 import pandas as pd
2 import matplotlib.pyplot as plt
3 import seaborn as sns
4
5 my_dataset = pd.read_hdf('ml_data.h5', 'Liquid_Orthopyroxene')
6
7 Elements = {
8     'Liquid': ['SiO2', 'TiO2', 'Al2O3', 'FeOtot', 'MgO',
9              'MnO', 'CaO', 'Na2O', 'K2O', 'H2O'],
10    'Orthopyroxene': ['SiO2', 'TiO2', 'Al2O3', 'FeOtot',
11                     'MgO', 'MnO', 'CaO', 'Na2O', 'K2O', 'Cr2O3']}
12
13 fig = plt.figure(figsize=(7,9))
14 x_labels_melt = [r'SiO$_2$ ', r'TiO$_2$ ', r'Al$_2$O$_3$ ',
15                 r'FeO$_t$ ', r'MnO', r'MgO', r'CaO',
16                 r'Na$_{20}$ ', r'K$_{20}$ ', r'H$_{20}$ ']
17 for i, col in enumerate(Elements['Liquid']):
18     ax1 = fig.add_subplot(5, 2, i+1)
19     sns.kdeplot(my_dataset['Liquid_' + col], fill=True,
20               color='k', facecolor='#BFD7EA', ax = ax1)
21     ax1.set_xlabel(x_labels_melt[i] + ' [wt. %] the melt')
22     if i in [0,2,4,6,8]:
23         ax1.set_ylabel('Prob. Density')
24     else:
25         ax1.set_ylabel=None)
26
27 fig.align_ylabels()
28 fig.tight_layout()
29
30 fig1 = plt.figure(figsize=(7,9))
31 x_labels_cpx = [r'SiO$_2$ ', r'TiO$_2$ ', r'Al$_2$O$_3$ ',
32                r'FeO$_t$ ', r'MnO', r'MgO', r'CaO',
33                r'Na$_{20}$ ', r'K$_{20}$ ', r'Cr$_2$O$_3$ ']
34 for i, col in enumerate(Elements['Orthopyroxene']):
35     ax2 = fig1.add_subplot(5, 2, i+1)
36     sns.kdeplot(my_dataset['Orthopyroxene_' + col], fill=True,
37               color='k', facecolor='#BFD7EA', ax = ax2)
38     ax2.set_xlabel(x_labels_cpx[i] + ' [wt. %] in opx')
39     if i in [0,2,4,6,8]:
40         ax2.set_ylabel('Prob. Density')
41     else:
42         ax2.set_ylabel=None)
43
44 fig1.align_ylabels()
45 fig1.tight_layout()

```

Listing 9.2 Descriptive statistics applied to our orthopyroxenes.

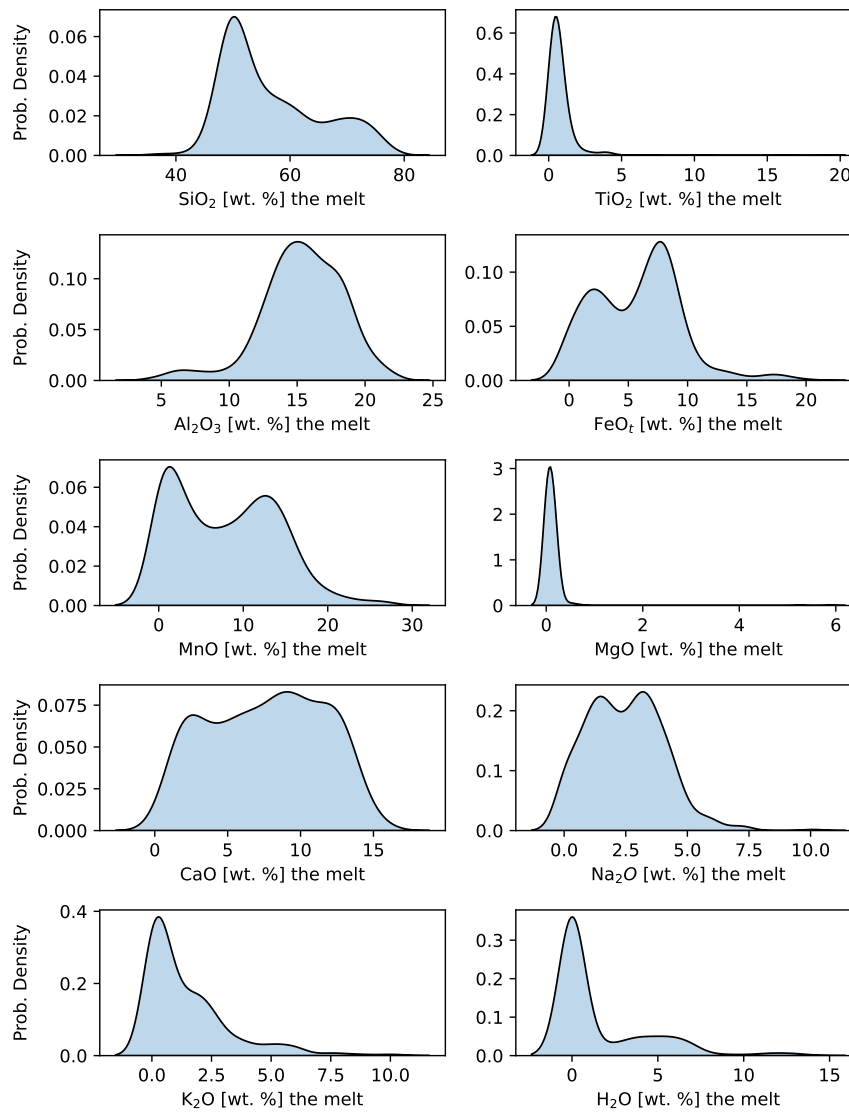


Fig. 9.1 Result of code listing 9.2. Descriptive statistics on the melt phase.

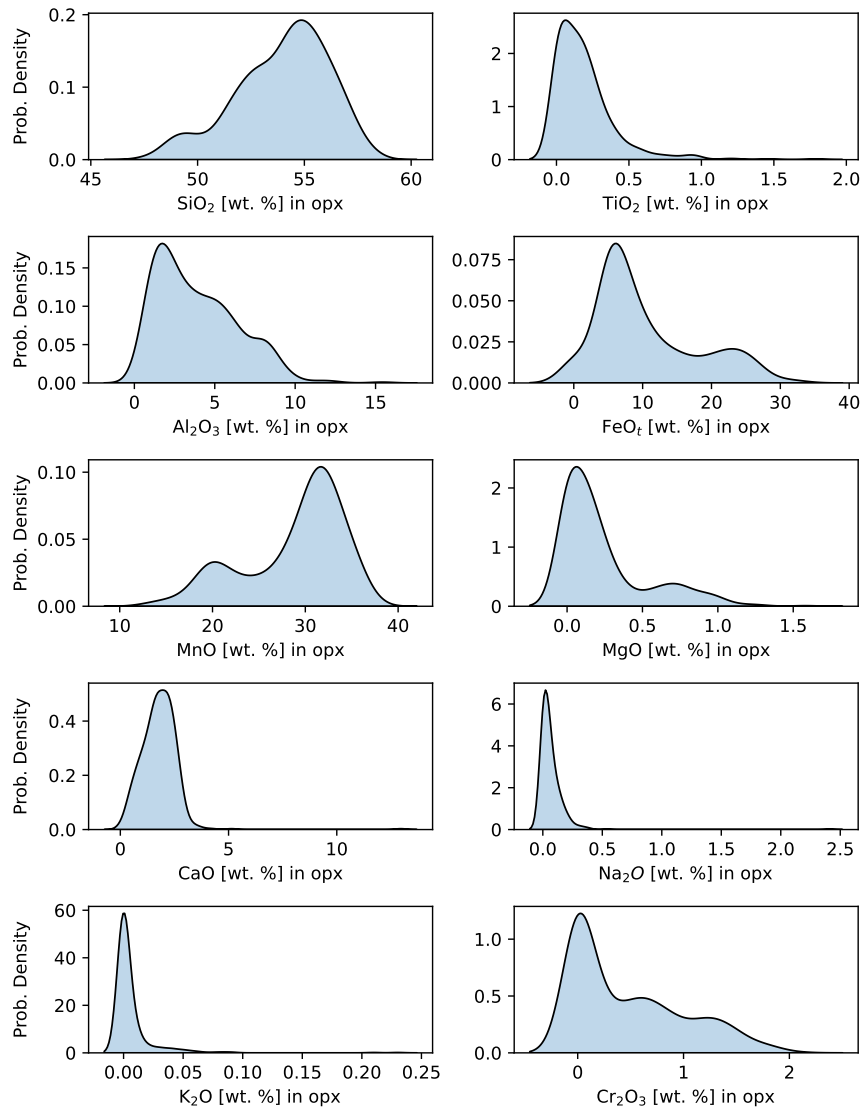


Fig. 9.2 Result of code listing 9.2. Descriptive statistics on the opx phase.

9.3 Compositional data analysis

In section 3.3, we introduced the basic concepts of compositional data analysis. Also, in section 3.3, we discussed why most of the advanced statistical techniques cannot be applied to compositional data without a proper transformation. Indeed, they assume to deal with independent data in the range form $-\infty$ to ∞ . Intrinsically, compositional features are in the range form 0 to 100 (or from 0 to 1) and they are not independent, since changing the value of one element will automatically affect the abundance of the other components (Aitchison, 1982). Decision tree ensembles like Random Forest (Song & Lu, 2015) and Extremely Randomized Trees (Geurts et al., 2006) do not have any specific assumption on the data structure. Therefore, they can be applied to un-transformed data (Aitchison, 1982). However, recent studies reports that tree ensembles experience an improvement in the performances when they are applied to log-ratio pairwise transformed data (Tolosana-Delgado et al., 2019). Although tree based ensembles does not strictly require a CoDA transformation, they benefit from the introduction of new features (i.e., pairwise log-ratios) derived from the existing ones (i.e., the augmentation of the feature input space). It results in a reduction of the overfitting leading to an improved generalization capability. In the present chapter, we will compare the results of the Extremely Randomized Trees algorithm on both un-transformed and un-transformed plus log-ratio pairwise transformed data, as suggested by Tolosana-Delgado et al. (2019). To introduce the log-ratio pairwise transformation in our pre-processing strategy, we simply need the addition of a new function to code listing 9.1, and then call it within pre-processing function. The code listing 9.3 reports the final version of our pre-processing strategy, now including the log-ratio pairwise transformation.

```

1 import os
2 import pandas as pd
3 import numpy as np
4
5 Elements = {
6     'Liquid': ['SiO2', 'TiO2', 'Al2O3', 'FeOtot', 'MgO',
7              'MnO', 'CaO', 'Na2O', 'K2O'],
8     'Orthopyroxene': ['SiO2', 'TiO2', 'Al2O3', 'FeOtot',
9                      'MgO', 'MnO', 'CaO', 'Na2O', 'Cr2O3']}
10
11 def calculate_cations_on_oxygen_basis(
12     myData0, myphase, myElements, n_oxygens):
13
14     Weights = {'SiO2': [60.0843, 1.0, 2.0],
15               'TiO2': [79.8788, 1.0, 2.0],
16               'Al2O3': [101.961, 2.0, 3.0],
17               'FeOtot': [71.8464, 1.0, 1.0],
18               'MgO': [40.3044, 1.0, 1.0],
19               'MnO': [70.9375, 1.0, 1.0],
20               'CaO': [56.0774, 1.0, 1.0],
21               'Na2O': [61.9789, 2.0, 1.0],
22               'K2O': [94.196, 2.0, 1.0],

```

```

23         'Cr2O3': [151.9982, 2.0, 3.0],
24         'P2O5': [141.937, 2.0, 5.0],
25         'H2O': [18.01388, 2.0, 1.0]}
26
27     myData = myData0.copy()
28     # Cation mole proportions
29     for el in myElements:
30         myData[el + '_cat_mol_prop'] = myData[myphase +
31             '_' + el] * Weights[el][1] / Weights[el][0]
32     # Oxygen mole proportions
33     for el in myElements:
34         myData[el + '_oxy_mol_prop'] = myData[myphase +
35             '_' + el] * Weights[el][2] / Weights[el][0]
36     # Oxygen mole proportions totals
37     totals = np.zeros(len(myData.index))
38     for el in myElements:
39         totals += myData[el + '_oxy_mol_prop']
40     myData['tot_oxy_prop'] = totals
41     # totcations
42     totals = np.zeros(len(myData.index))
43     for el in myElements:
44         myData[el + '_num_cat'] = n_oxygens * myData[el +
45             '_cat_mol_prop'] / myData['tot_oxy_prop']
46         totals += myData[el + '_num_cat']
47     return totals
48
49 def filter_by_cryst_formula(dataFrame, myphase, myElements):
50
51     c_o_Tolerance = {'Orthopyroxene': [4, 6, 0.025]}
52
53     dataFrame['Tot_cations'] = calculate_cations_on_oxygen_basis(
54         myData0 = dataFrame, myphase = myphase,
55         myElements = myElements,
56         n_oxygens = c_o_Tolerance[myphase][1])
57
58     dataFrame = dataFrame[
59         (dataFrame['Tot_cations'] < c_o_Tolerance[myphase][0]
60          + c_o_Tolerance[myphase][2]) &
61         (dataFrame['Tot_cations'] > c_o_Tolerance[myphase][0]
62          - c_o_Tolerance[myphase][2])]
63
64     dataFrame = dataFrame.drop(columns=['Tot_cations'])
65     return dataFrame
66
67 def adjustFeOtot(dataFrame):
68     for i in range(len(dataFrame.index)):
69         try:
70             if pd.to_numeric(dataFrame.Fe2O3[i]) > 0:
71                 dataFrame.loc[i, 'FeOtot'] = (
72                     pd.to_numeric(dataFrame.FeO[i]) + 0.8998 *
73                     pd.to_numeric(dataFrame.Fe2O3[i]))
74             else:
75                 dataFrame.loc[i,
76                     'FeOtot'] = pd.to_numeric(dataFrame.FeO[i])

```

```

77     except:
78         dataframe.loc[i, 'FeOtot'] = 0
79     return dataframe
80
81 def adjust_column_names(dataFrame):
82     dataframe.columns = [c.replace('Wt: ', '')
83                          for c in dataframe.columns]
84     dataframe.columns = [c.replace(' ', '')
85                          for c in dataframe.columns]
86     return dataframe
87
88 def select_base_features(dataFrame, my_elements):
89     dataframe = dataframe[my_elements]
90     return dataframe
91
92 def data_imputation(dataFrame):
93     dataframe = dataframe.fillna(0)
94     return dataframe
95
96 def pwlr(dataFrame, my_phases):
97
98     for my_pahase in my_phases:
99         my_indexes = []
100        column_list = Elements[my_pahase]
101
102        for col in column_list:
103            col = my_pahase + '_' + col
104            my_indexes.append(dataFrame.columns.get_loc(col))
105            my_min = dataframe[col][dataframe[col] > 0].min()
106            dataframe.loc[dataframe[col] == 0,
107                          col] = dataframe[col].apply(
108                              lambda x: np.random.uniform(
109                                  np.nextafter(0.0, 1.0), my_min))
110
111        for ix in range(len(column_list)):
112            for jx in range(ix+1, len(column_list)):
113                col_name = 'log_' + dataframe.columns[
114                    my_indexes[jx]] + '_' + dataframe.columns[
115                        my_indexes[ix]]
116                dataframe.loc[:, col_name] = np.log(
117                    dataframe[dataframe.columns[my_indexes[jx]]] / \
118                    dataframe[dataframe.columns[my_indexes[ix]]])
119        return dataframe
120
121 def data_pre_processing(phase_1, phase_2, out_file):
122
123     try:
124         os.remove(out_file)
125     except OSError:
126         pass
127
128     starting = pd.read_excel('LEPR_download.xls',
129                             sheet_name='Experiment')
130     starting= adjust_column_names(starting)

```

```
131 starting.name = ''
132 starting = starting[['Index', 'T(C)', 'P(GPa)']]
133 starting.to_hdf(out_file, key='starting_material')
134
135 phases = [phase_1, phase_2]
136
137 for ix, my_phase in enumerate(phases):
138     my_dataset = pd.read_excel('LEPR_download.xls',
139                               sheet_name = my_phase)
140
141     my_dataset = (my_dataset.
142                  pipe(adjust_column_names).
143                  pipe(adjustFeOtot).
144                  pipe(select_base_features,
145                        my_elements= Elements[my_phase])).
146                  pipe(data_imputation))
147
148     my_dataset = my_dataset.add_prefix(my_phase + '_')
149     my_dataset.to_hdf(out_file, key=my_phase)
150
151 my_phase_1 = pd.read_hdf(out_file, phase_1)
152 my_phase_2 = pd.read_hdf(out_file, phase_2)
153
154 my_dataset = pd.concat([starting,
155                         my_phase_1,
156                         my_phase_2], axis=1)
157
158 my_dataset = my_dataset[(my_dataset['Liquid_SiO2'] > 35)&
159                         (my_dataset['Liquid_SiO2'] < 80)]
160
161 my_dataset = my_dataset[(
162     my_dataset['Orthopyroxene_SiO2'] > 0)]
163
164 my_dataset = my_dataset[(my_dataset['P(GPa)'] <= 2)]
165
166 my_dataset = my_dataset[(my_dataset['T(C)'] >= 650)&
167                         (my_dataset['T(C)'] <= 1800)]
168
169 my_dataset = filter_by_cryst_formula(dataFrame = my_dataset,
170                                     myphase = phase_2,
171                                     myElements = Elements[phase_2])
172
173 my_dataset = my_dataset.sample(frac=1,
174                                random_state=50).reset_index(drop=True)
175
176 my_labels = my_dataset[['Index', 'T(C)', 'P(GPa)']]
177 my_dataset = my_dataset.drop(columns=['T(C)', 'P(GPa)'])
178
179 my_labels.to_hdf(out_file, key='labels')
180 my_dataset.to_hdf(out_file, key= phase_1 + '_' + phase_2)
181
182 my_dataset = pwlr(my_dataset,
183                  my_phases= [phase_1, phase_2])
184 my_dataset.to_hdf(out_file,
```

```

185         key= phase_1 + '_' + phase_2 + '_lrpwt')
186
187 data_pre_processing(phase_1='Liquid' ,
188                   phase_2='Orthopyroxene' ,
189                   out_file='ml_data.h5')

```

Listing 9.3 Final implementation of our pre-processing strategy.

9.4 Model training and error assessment

In agreement with Petrelli et al. (2020) , we train the Extremely Randomized Trees algorithm on the pre-processed data set. Also, we will use a Monte Carlo simulation to propagate the errors and asses the goodness of the model. The strategy of the Monte Carlo approach consist of repeating many times the: 1) random splitting of the data set; 2) the training of the algorithm starting from a different random seeding. The code listing 9.4 shows the implementation of the Monte Carlo strategy. It consists of defining a function named *monte_carlo_simulation()* (line 9). Within the *monte_carlo_simulation()* function, we repeat *n* times the procedures of train-validation splitting (lines from 16 to 18), normalization to zero mean and unit variance (lines from 20 to 22), training (lines from 24 to 26), prediction (line 27), error assessment (lines from 29 to 35), and the storing of the results (lines from 36 to 42). Starting from line 45, I then perform four Monte Carlo simulations, two (Liquid plus opx and opx only, respectively) with the raw data, and two with the transformed ones.

```

1 import pandas as pd
2 import numpy as np
3 from sklearn.preprocessing import StandardScaler
4 from sklearn.ensemble import ExtraTreesRegressor
5 from sklearn.model_selection import train_test_split
6 from sklearn.metrics import r2_score
7 from sklearn.metrics import mean_squared_error
8
9 def monte_carlo_simulation(X, y, indexes, n, key_res):
10
11     r2 = []
12     RMSE = []
13
14     for i in range(n):
15         my_res = {}
16         X_train, X_valid, y_train, y_valid, \
17             indexes_train, indexes_valid = train_test_split(
18             X, y.ravel(), indexes, test_size=0.2)
19
20         scaler = StandardScaler().fit(X_train)
21         X_train = scaler.transform(X_train)

```

```

22     X_valid = scaler.transform(X_valid)
23
24     regressor = ExtraTreesRegressor(n_estimators=450,
25                                     max_features=1).fit(
26                                     X_train, y_train)
27     my_prediction = regressor.predict(X_valid)
28
29     my_res = {'indexes_valid': indexes_valid,
30              'prediction': my_prediction}
31
32     my_res_pd = pd.DataFrame.from_dict(my_res)
33     r2.append(r2_score(y_valid, my_prediction))
34     RMSE.append(np.sqrt(mean_squared_error(y_valid,
35                                           my_prediction)))
36     my_res_pd.to_hdf('ml_data.h5',
37                    key= key_res + '_res_' + str(i))
38
39     my_scores = {'r2_score': r2,
40                 'root_mean_squared_error': RMSE}
41     my_scores_pd = pd.DataFrame.from_dict(my_scores)
42     my_scores_pd.to_hdf('ml_data.h5', key = key_res + '_scores')
43
44
45 my_keys = ['Liquid_Orthopyroxene', 'Liquid_Orthopyroxene_lrpwt']
46
47 for my_key in my_keys:
48
49     # Liquid plus opx calibration
50     liquid_opx = pd.read_hdf('ml_data.h5', my_key)
51     print(liquid_opx.columns)
52     X_liquid_opx = liquid_opx.values
53     my_labels = pd.read_hdf('ml_data.h5', 'labels')
54     my_y = my_labels['T(C)'].values
55     my_indexes = my_labels['Index'].values
56     monte_carlo_simulation(X = X_liquid_opx, y = my_y,
57                           indexes = my_indexes,
58                           n =1000, key_res = my_key)
59
60     # opx only calibration
61     opx = liquid_opx.loc[:,
62                         ~liquid_opx.columns.str.startswith('Liquid')]
63     X_opx = opx.values
64     my_key = my_key.replace("Liquid_", "")
65     monte_carlo_simulation(X = X_opx,
66                           y = my_y, indexes = my_indexes,
67                           n =1000, key_res = my_key)

```

Listing 9.4 Training of the model in a Monte Carlo simulation.

9.5 Results Evaluation

Figures 9.3 and 9.4 show the results of the Monte Carlo simulations (deriving from the code listing 9.5). In detail, in both figures 9.3 and 9.4, the upper panels refer to raw data, whereas the lower panels display the results on raw data plus the features deriving from the log-ratio pairwise transformation.

It is noteworthy that the addition of the features deriving from the log-ratio pairwise transformation strongly improves the performance of the orthopyroxene only calibration of the thermometer (Fig. 9.3). In this case, both the Root Mean Squared Error and the r^2 improves by 14° and 0.04, respectively.

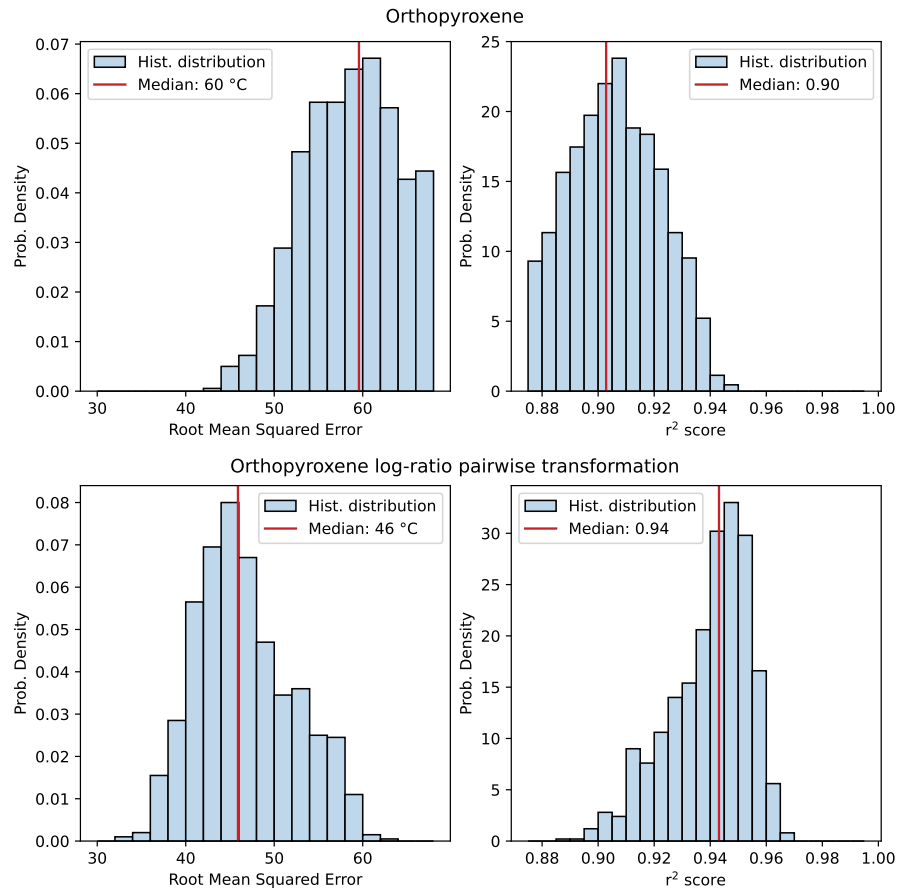


Fig. 9.3 Result of code listing 9.5, i.e., the Monte Carlo simulation the orthopyroxene only system.

Differently from the orthopyroxene only calibration, It seems that the Liquid plus orthopyroxene system does not benefit by the addition of the features deriving from

the log-ratio pairwise transformation (Fig. 9.4). In this case, both the Root Mean Squared Error only differ by 4°C and the r^2 is stable to 0.95.

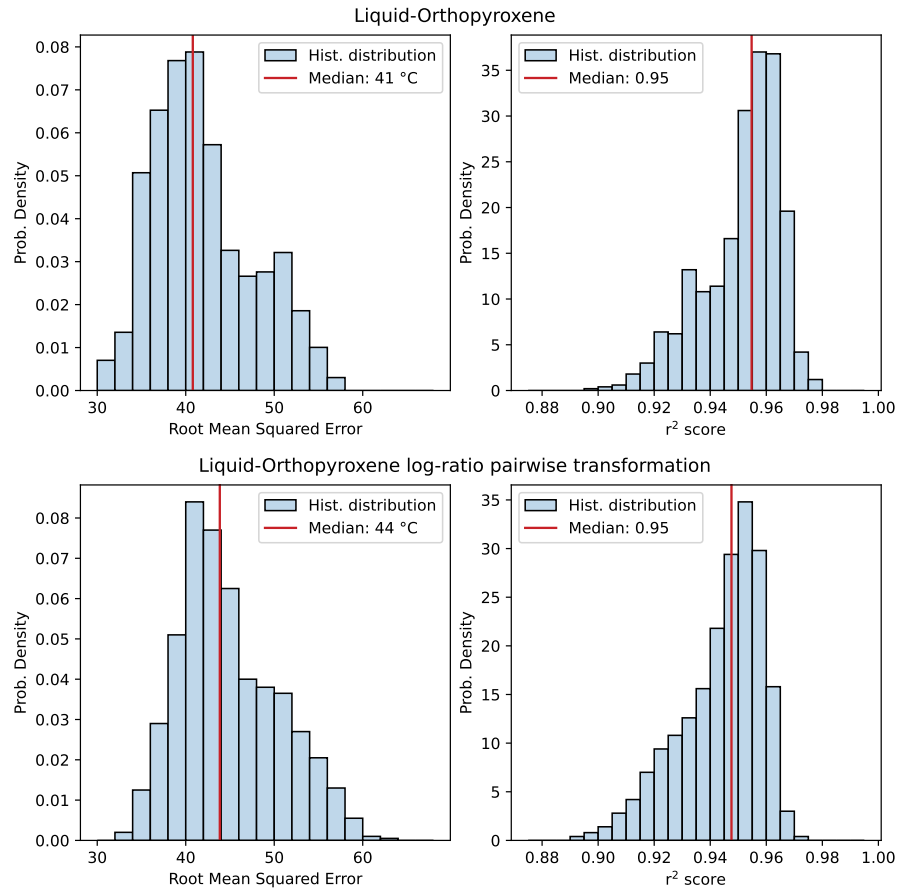


Fig. 9.4 Result of code listing 9.5, i.e., the Monte Carlo simulation for the liquid-orthopyroxene system.

```

1 import pandas as pd
2 import numpy as np
3 import matplotlib.pyplot as plt
4
5 for my_key in ['Orthopyroxene', 'Liquid_Orthopyroxene']:
6
7     fig = plt.figure(figsize=(8,8),constrained_layout=True)
8     subfigs = fig.subfigures(nrows=2, ncols=1)
9     for j, (trans, my_title) in enumerate(zip(['', '_lrpwt'],
10 [my_key, my_key+' log-ratio pairwise transformation']))):

```



```

11 my_scores = pd.read_hdf('ml_data.h5',
12                        my_key + trans + '_scores')
13
14 RMSE_ML_valid_median_T = np.median(
15     my_scores['root_mean_squared_error'])
16 R2_valid_median_T = np.median(my_scores['r2_score'])
17
18 subfigs[j].suptitle(my_title.replace('_', '-'))
19
20 # left panel
21 ax = subfigs[j].add_subplot(1, 2, 1)
22 bins = np.arange(30, 70, 2)
23 ax.hist(my_scores['root_mean_squared_error'], bins=bins,
24         density = True, color = '#BFD7EA',
25         edgecolor = 'k',
26         label='Hist. distribution')
27 ax.axvline(RMSE_ML_valid_median_T,
28           color='#C82127',
29           label='Median: {:.0f} C '.format(
30               RMSE_ML_valid_median_T))
31 ax.set_xlabel('Root Mean Squared Error')
32 ax.set_ylabel('Prob. Density')
33 ax.legend()
34
35 # right panel
36 ax = subfigs[j].add_subplot(1, 2, 2)
37 bins = np.arange(0.875, 1, 0.005)
38 ax.hist(my_scores['r2_score'], bins = bins,
39         density = True, color = '#BFD7EA',
40         edgecolor='k',
41         label='Hist. distribution')
42 ax.axvline(R2_valid_median_T, color='#C82127',
43           label='Median: {:.2f}'.format(
44               R2_valid_median_T))
45 ax.set_xlabel(r'r$^2$ score')
46 ax.set_ylabel('Prob. Density')
47 ax.legend()

```

Listing 9.5 Plotting the results of the Monte Carlo simulation.

Part IV
Scaling Your Machine Learning Models

Chapter 10

Parallel Computing and Scaling with Dask

10.1 Warming Up: Basic Definitions

Processor, CPU, core: the traditional definition of processor and Central Processing Unit (CPU) can be expressed as (Caesar Wu, 2015): “a microprocessor chip that sequentially (i.e., one by one) executes a series of basic processing tasks based on an input”. Modern CPUs largely exceed the traditional definition. They integrate many components and also host a cache memory. In modern CPUs, the very basic processing tasks can be duplicated and executed by self-contained execution blocks that fit the traditional definition of a processor (Caesar Wu, 2015). These self-contained execution blocks are typically named “cores” (Caesar Wu, 2015).

Multi-core processors and parallel hardware: Multi-core processors, chip multi-processor (CMP), and parallel hardware are often used as synonyms (Peter Pacheco, 2020). A CMP incorporate many processors and cache memory on a chip. Parallel hardware is nowadays ubiquitous since it is almost impossible to find a modern laptop, desktop, or server that doesn’t use a multicore processor (Peter Pacheco, 2020).

Graphics processing unit (GPU): “GPUs are multi-core processing units made of massively parallel, smaller, and more specialized cores than those generally found in high-performance CPUs. GPU architecture efficiently processes vector data (an array of numbers) and is often referred to as vector architecture.”¹

Field programmable gate arrays (FPGA): “FPGAs are integrated circuits with a programmable hardware fabric. Unlike CPUs and GPUs, which are software-programmable fixed architectures, FPGAs are reconfigurable. When writing software targeting an FPGA, compiled instructions become hardware components that are laid out on the FPGA fabric in space, and those components can all execute in parallel.”²

Distributed computing: “A computer system consisting of a multiplicity of processors, each with its own local memory, connected via a network. Loading or store

¹ <https://intel.ly/39XimzH>

² <https://intel.ly/39XimzH>

instructions issued by a processor can only address the local memory and different mechanisms are provided for global communication.” (Padua David, 2011).

Serial programs: Serial programs are those that were conceived and written to run on a single processor (Peter Pacheco, 2020). If you run it in the presence of multiple processors or a distributed architecture, the performance will not improve magically, since the instructions will be executed, sequentially, within one of the available cores.

Parallel computing: Parallel computing is a type of computation in which many calculations or processes are carried out simultaneously (Peter Pacheco, 2020). Parallel computing takes advantage form multiple processors (i.e., CMP, GPU, and FPGA) or a distributed architecture (Peter Pacheco, 2020).

10.2 Basics of Dask

Dask³ aims at overcoming single-machine restrictions by adding object scalability to Python scientific libraries like pandas, NumPy, and scikit-learn (Daniel, 2019). It consists of three main layers: 1) scheduler, 2) low-level APIs, and 3) high-level APIs (Fig. 10.1). We will mainly interact with the high-level APIs that governs Dask arrays, Dask DataFrames, and Dask ML, allowing us to scale NumPy, pandas, and scikit-learn objects, respectively. In detail, our main goal is to allow single machines to work with medium data sets and deploy clusters to elaborate large data sets or large models.

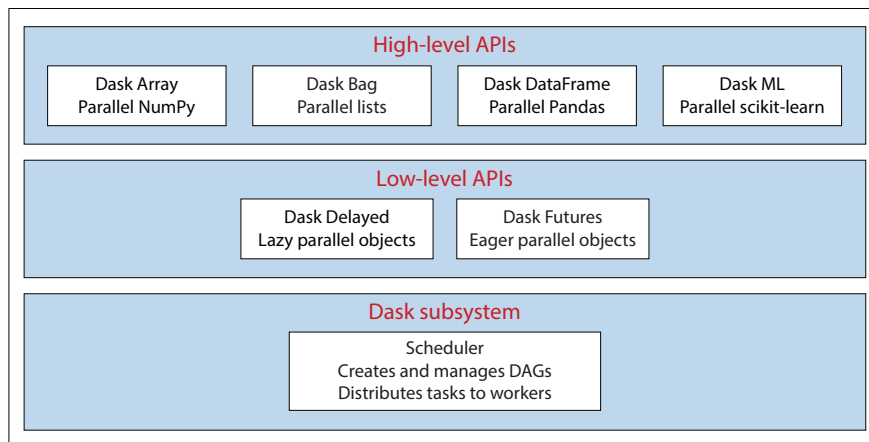


Fig. 10.1 Dask fundamentals, modified from (Daniel, 2019)

³ <https://www.dask.org>

Dask Array

Dask arrays combine many NumPy arrays, arranged into chunks (i.e., a single NumPy array) within a grid (Fig. 10.2).

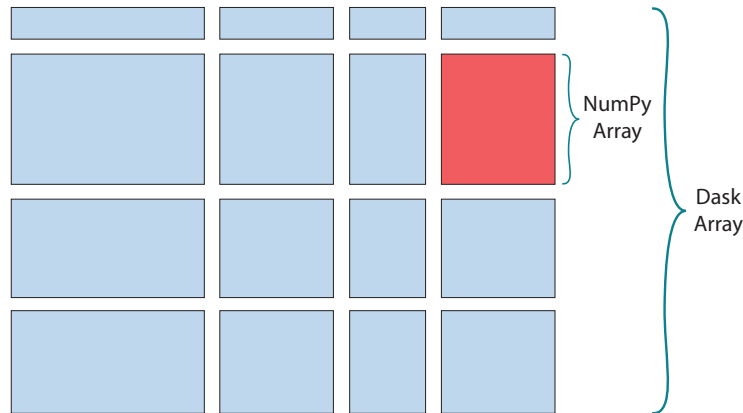


Fig. 10.2 Dask Arrays, modified from <https://examples.dask.org/array.html>

```
[1]: import dask.array as da
x = da.random.random((100000, 100000), chunks=(1000, 1000))
x
```

```
[1]:
```

	Array	Chunk
Bytes	74.51 GiB	7.63 MiB
Shape	(100000, 100000)	(1000, 1000)
Count	10000 Tasks	10000 Chunks
Type	float64	numpy.ndarray

Fig. 10.3 Defining a Dask array

Dask arrays allow most NumPy operations and allow parallelizing current codes easily. Defining a Dask array is the same as defining a NumPy array, with the only difference that you need to import `dask.array` instead of NumPy (Fig. 10.3). As an example, Fig. 10.3 shows how to create a $10^5 \times 10^5$ Dask array, made of random numbers. Optionally, you can also define the dimension of the chunks, i.e., the NumPy arrays that constitute the building blocks of the Dask array. Please note that in Jupyter Notebooks, you can get many information on the Dask array you created.

As an example, Fig. 10.3 shows that the total size of x will be 74.51 GiB (i.e., Gibibytes, GiB, with 1 GiB \sim 1.074 GB). Also, the size of a single chunk will be 7.63 MiB. I'm using the future tense since x has not been fully generated yet, but only 'lazy' evaluated (see section 10.3 for further details)

Dask Data Frame

A Dask DataFrame is the parallel counterpart of a pandas DataFrame (Fig. 10.4).

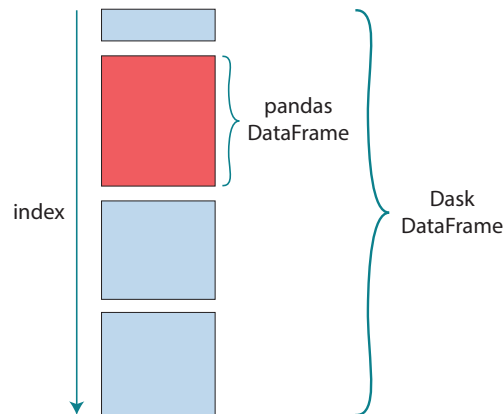


Fig. 10.4 Dask Arrays, modified from <https://examples.dask.org/dataframe.html>

```
[1]: import dask.dataframe as dd
[3]: train_dataset = dd.read_hdf('ml_data.h5', key='train')
[4]: train_dataset
[4]: Dask DataFrame Structure:
      CALI  Delta_CALI  log_RMED  Delta_log_RMED  log_RDEP  Delta_log_RDEP
npartitions=2
      float64  float64  float64  float64  float64  float64
      ...      ...      ...      ...      ...      ...
      ...      ...      ...      ...      ...      ...

Dask Name: read-hdf, 2 tasks
```

Fig. 10.5 Importing a pandas DataFrame stored in an HDF5 files as Dask DataFrame

Table 10.1 Dask methods to import and create a Dask DataFrame. Please note that most of them are equivalent to pandas methods, i.e., Tab. 3.1 (modified from <https://docs.dask.org/en/stable/dataframe-api.html>).

Method	Description
<code>read_table()</code>	Read general delimited file
<code>read_csv()</code>	Read comma-separated values (csv) files
<code>read_fwf()</code>	Read fixed-width files
<code>read_parquet()</code>	Read parquet files
<code>read_hdf()</code>	Read Hierarchical Data Format (HDF) files
<code>read_json()</code>	Create a Dask DataFrame from a set of JSON files
<code>read_orc()</code>	Create a Dask DataFrame from ORC file(s)
<code>read_sql_table()</code>	Read SQL database table
<code>read_sql_query()</code>	Read SQL query
<code>read_sql()</code>	Read SQL query or database table
<code>from_array()</code>	Read any sliceable array
<code>from_bcolz()</code>	Read BColz CTable
<code>from_dask_array()</code>	Create a Dask DataFrame from a Dask Array
<code>from_delayed()</code>	Create a Dask DataFrame from many Dask Delayed objects
<code>from_map()</code>	Create a Dask DataFrame collection from a custom function map
<code>from_pandas()</code>	Construct a Dask DataFrame from a Pandas DataFrame
<code>from_dict()</code>	Construct a Dask DataFrame from a Python Dictionary
<code>Bag_to_dataframe()</code>	Create Dask Dataframe from a Dask Bag

In detail, Dask DataFrames are composed of many smaller pandas DataFrames, split along an index. A Dask DataFrames may live on a local disk for medium data sets, or in a cluster in the case of large data sets. One Dask DataFrame operation triggers many operations on the constituent pandas DataFrames.

As an example, we could import the data set that we developed in Chapter 8 and that we saved as HDF5. Fig. 10.5 shows a portion of Jupyter Notebook highlighting how to generate a Dask DataFrame from the file named *ml_data.h5*. Please note that the procedure is not dissimilar from that in pandas. The only difference consists of importing a *dask.DataFrame* instead of a *pandas.DataFrame*. Also note that Dask decided to split the DataFrame into two portions and that instead of the real values, all rows are filled with ellipsis (...). This is because the data set was evaluated only lazily (please refer to section 10.3 for further details). The physically import *train_dataset*, Dask requires an additional step, i.e., the use of the `compute()` method (Fig. 10.6)

```
[5]: train_dataset.compute()
```

	CALI	Delta_CALI	log_RMED	Delta_log_RMED	log_RDEP	Delta_log_RDEP
0	19.480835	0.000000	0.207206	0.000000	0.254954	0.000000
1	19.468800	-0.012035	0.208997	0.001791	0.254220	-0.000735
2	19.468800	0.000000	0.211243	0.002246	0.255449	0.001230
3	19.459282	-0.009518	0.209942	-0.001301	0.255638	0.000189
4	19.453100	-0.006182	0.204847	-0.005096	0.254137	-0.001501
...
1170506	8.423170	0.001802	0.247442	0.000026	0.241466	0.000024
1170507	8.379244	-0.043926	0.247442	0.000026	0.241466	0.000024
1170508	8.350248	-0.028996	0.247442	0.000026	0.241466	0.000024
1170509	8.313779	-0.036469	0.247442	0.000026	0.241466	0.000024
1170510	8.294910	-0.018868	0.247442	0.000026	0.241466	0.000024

1170511 rows x 26 columns

Fig. 10.6 Physically importing a pandas DataFrame stored in an HDF5 files as Dask DataFrame

Dask ML

By model scaling, we could solve two common issues related to 2) data size and 1) model size (Tab. 10.2, Fig. 10.7). Regarding the size of your data set, when it comfortably fits the free RAM of the computing environment, i.e., you are working with a small data set (Tab. 10.2), pandas, NumPy, and scikit-learn are the libraries of choice to develop a ML strategy. In this case the scaling along the x-dimension of Fig. 10.7 is not required and not recommended.

Table 10.2 Data set classification in function of the size range. Modified from Daniel (2019)

Data set Size	Approximate Size Range	Fits in RAM?	Fits on local disk?
Small data set	Less than the free RAM on your system [e.g., 16 GB]	Yes	Yes
Medium data set	Larger than the free RAM on your system and less than capability of the local disk [e.g., 2 TB]	No	Yes
Large data set	Larger than the capability of the local disk	No	No

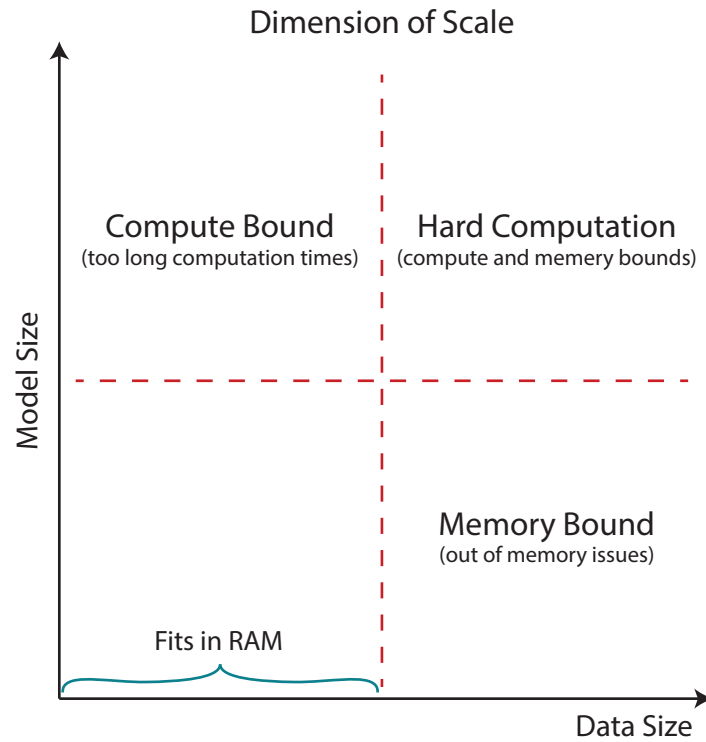


Fig. 10.7 Dimension of scale, modified from <https://ml.dask.org>

```
In [1]: import numpy as np
In [2]: my_dataset = np.random.normal(loc=1.0, scale=2.0, size=100000000)
In [3]: np.mean(my_dataset)
Out[3]: 0.9995566509046069
In [4]: np.std(my_dataset)
Out[4]: 1.9999512502789483
In [5]: my_dataset.nbytes / 1024**3
Out[5]: 0.7450580596923828
```

Fig. 10.8 Working with a small data set, i.e., well-fitting your RAM budget

As an example, code listing 10.8 shows how to use Numpy to define (line 2) a small set of data (i.e, my_data) composed of 10^8 normally distributed pseudo-random numbers, characterized by a mean value and standard deviation equal to 1 and 2, respectively. Then, lines 3 and 4 simply check that the mean and the standard deviation of my_data are 1 and 2, respectively. Finally, line 5 estimated the memory usage of my_data, resulting equal to about 0.745 GiB.

However, when the size of the data set reaches the upper bound of the RAM (including any virtual memory generated using the hard disk), memory errors start occurring (code listing 10.9). As an example, increasing the size of my_data to $2.5 \cdot 10^9$, in a Linux system with 16GB of free memory, I got a ‘Memory error’ since the operating system was ‘Unable to allocate 18.6 GiB for an array with shape (2500000000,) and data type float64’. I was clearly experiencing a data size issue since I generated a ‘medium data set’ (Tab. 10.2).

```
In [1]: import numpy as np

In [2]: my_dataset = np.random.normal(loc=1.0, scale=2.0, size=2500000000)

-----
MemoryError                                Traceback (most recent call
last)
Input In [2], in <module>
----> 1 my_dataset = np.random.normal(loc=1.0, scale=2.0, size=250000
0000)

File mtrand.pyx:1507, in numpy.random.mtrand.RandomState.normal()

File _common.pyx:598, in numpy.random._common.cont()

MemoryError: Unable to allocate 18.6 GiB for an array with shape (250
0000000,) and data type float64
```

Fig. 10.9 When you exceed the free memory, you get a ‘Memory error.’

The use of Dask arrays allows overcoming the problem with minimal changes in our code. As an example, code listing 10.10 uses Dask Arrays, i.e., the parallel mimic of NumPy Arrays to complete, within a Linux Os with 16GB of free ram, the simple operations that were previously impossible using NumPy (i.e., code listing 10.9).

When the problem is the size of the model, i.e., it is growing too much, or it starts being too complex, all computations take too long times. As an example, the grid search I performed in chapter 8 took some hours to complete. Waiting of a few hours is not a big problem, usually. However, simply increasing the dimension of the grid search (e.g., increasing the number of the investigated hyper-parameters and densifying the grid) or the complexity of the decision tree ensemble (e.g., increasing the number of estimators), the execution time will drastically increase, up to days or weeks. Now, if you need to optimize several ML models, the total time required to complete your tasks easily reaches the time span of months or eventually years.

```

In [2]: import dask.array as da
In [4]: my_dataset = da.random.normal(loc=1.0, scale=2.0, size=250000000)
In [7]: da.mean(my_dataset).compute()
Out[7]: 1.0000155698953217
In [8]: da.std(my_dataset).compute()
Out[8]: 2.000041110411836

```

Fig. 10.10 Using Dask to work with medium size data set

The main aim of Dask ML is to provide scalable machine learning in Python for popular machine learning libraries like scikit-Learn(Pedregosa et al., 2011) , XGBoost, and others.

10.3 ‘Eager’ computation Vs. ‘Lazy’ evaluation

Usually, in Python, we deal with the so-called ‘Eager’ computation. It simply means that Python performs each operation, e.g, transformations and calculations, immediately upon being called. As an example, Fig. 10.12 reports the definition of an ‘Eager’ function, i.e. *simple_lithopress()* at line 2, to estimate the lithostatic pressure when assuming both the density and acceleration due to gravity as constants. We disclose the ‘Eager’ nature of the function at lines 3 and 4, since *simple_lithopress()* returns a computed value as soon as we call it in the code workflow, i.e., it performs calculations immediately.

```

[1]: def simple_lithopress(z, ro, g):
      p_MPa = z*g*ro/1e6 # return the pressure in MPa
      return p_MPa
[3]: my_pressure = simple_lithopress(z=2000, ro=2900, g=9.8)
[3]: my_pressure
[3]: 56.84

```

Fig. 10.11 Defining the ‘Eager’ function named *simple_lithopress()*

Similarly, if we perform a Monte Carlo error propagation combining NumPy arrays and the *simple_lithopress()* function, we get an immediate execution lasting less than one second, generating an array made of 10^7 elements and consuming about 76 MB.

```
[4]: import numpy as np

[5]: %%time
z_dist = np.random.normal(loc=2000, scale=200, size= 10000000)
ro_dist = np.random.normal(loc=2900, scale=290, size= 10000000)
g_dist = np.random.normal(loc=9.8, scale=0.1, size= 10000000)
my_pressure_dist = simple_lithopress(z=z_dist, ro = ro_dist, g = g_dist)

CPU times: user 854 ms, sys: 106 ms, total: 960 ms
Wall time: 963 ms

[6]: my_pressure_dist.nbytes / 1024**2 # size in MB

[6]: 76.2939453125
```

Fig. 10.12 Performing a Monte Carlo error propagation using the ‘Eager’ `simple_lithopress()`

To be aware of what we are doing, Fig. 10.13 shows the histogram distribution of the computed pressures, resulting from depth, density, and acceleration due to gravity estimations, taking into account also the error estimates.

```
1 import matplotlib.pyplot as plt
2
3 my_pressure_mean = np.mean(my_pressure_dist)
4 my_pressure_std = np.std(my_pressure_dist)
5
6 fig, ax = plt.subplots()
7 ax.hist(my_pressure_dist, density=True, bins='auto',
8         color='#0F7F8B', label='Pressure estimates')
9 ax.axvline(my_pressure_mean, color='#C82127', label='mean value')
10 ax.axvspan(my_pressure_mean - my_pressure_std,
11            my_pressure_mean + my_pressure_std,
12            color='#F15C61', alpha=0.4,
13            label=r'1$\sigma$ estimate')
14 ax.set_xlabel('Pressure [MPa]')
15 ax.set_ylabel('Probability Density')
16 ax.legend()
17 plt.show()
```

Listing 10.1 Plotting the results of the Monte Carlo error propagation.

The ‘lazy’ evaluation operates differently from the ‘Eager’ computation. For the ‘lazy’ evaluation, Dask prepares a Directed Acyclic Graph (i.e., DAG) for the involved functions, operations, and transformations, but it does not perform any computation. DAGs are mathematical objects that are governed by graph theory. Describing the theory behind DAGs and graph theory is far beyond the scope of the present book. Please refer to specialized references to go deep into details about DAGs (Fiore & Campos, 2013; Maurer, 2013; Xu, 2003).

In this section, we are mainly interested in learning the main benefits of using DAGs for our computations. Among them, one of the most effective is the chance

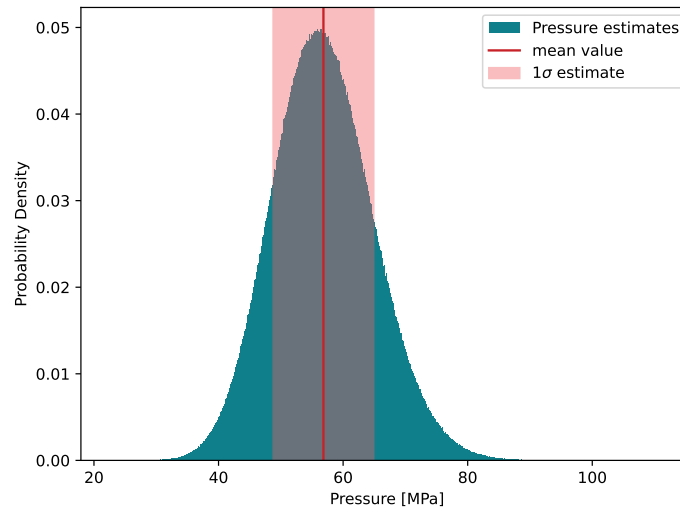


Fig. 10.13 Result of code listing 10.13

```
[1]: import dask.array as da
    from dask import delayed

[2]: def simple_lithopress(z, ro, g):
    p_MPa = z*g*ro/1e6 # return the pressure in MPa
    return p_MPa

    z_da = da.random.normal(loc=2000, scale=200, size= 10000000)
    ro_da = da.random.normal(loc=2900, scale=290, size= 10000000)
    g_da = da.random.normal(loc=9.8, scale=0.1, size= 10000000)

    my_pressure_da = da.map_blocks(simple_lithopress, z_da, g_da, ro_da)

[3]: my_pressure_da = simple_lithopress(z=z_da, ro = ro_da, g = g_da)

[4]: my_pressure_da.visualize(filename='my_DAG.pdf')
```

Fig. 10.14 Result of code listing 10.14

of evaluating and visualizing the structure and the complexity of our computations before running them. This opportunity comes with many advantages. As an example, you can evaluate the complexity and heaviness of the computations, allowing you to decide if performing them in a single machine, a small cluster, or a High Performance (HPC). As example Fig. 10.14 shows how to perform a 'lazy' evaluation of the Monte Carlo error propagation performed in Fig. 10.12. The resulting DAG is displayed in Fig. 10.15. It is a simple structure showing that, after generating three normal distributions for the depth, density, and acceleration due to gravity, the *simple_lithopress()* function uses them as input and generates an output. If we increase the size of the three input arrays from 10^7 to 10^8 , the structure of the DAG changes

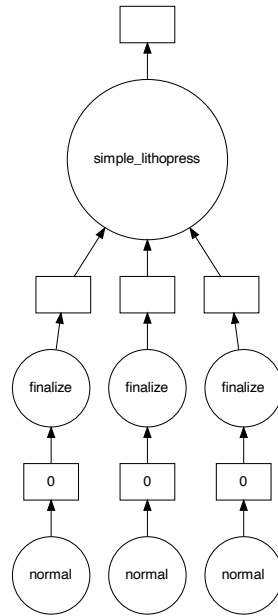


Fig. 10.15 Result of code listing 10.15

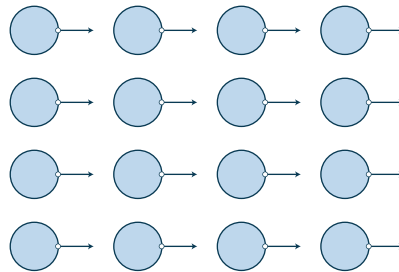


Fig. 10.16 'Embarrassingly parallel' workload

(Fig. 10.17). In detail, we defined a so-called 'embarrassingly parallel' workload. It is the optimal case since the problem results perfectly parallelizable.

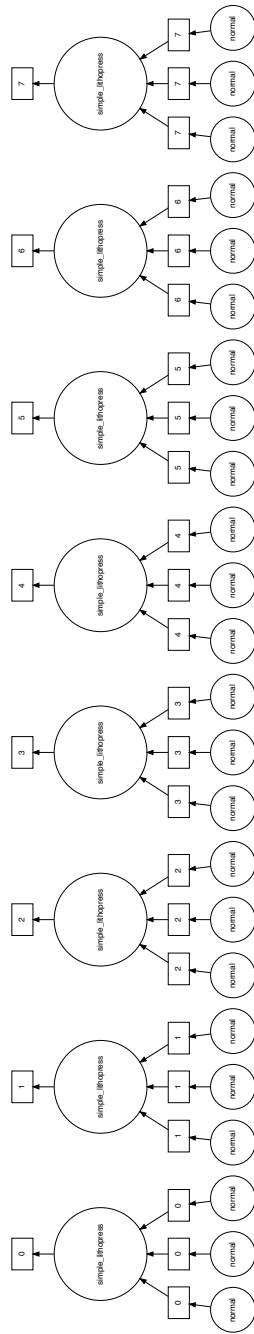


Fig. 10.17 Result of code listing 10.17



Fig. 10.18 Dask Interactive dashboard

10.4 Diagnostic and feedback

The Dask distributed scheduler, i.e., the one utilized to interact with clusters, provides us with an effective interactive dashboard. It consists of a rich ecosystem of monitoring and profiling tools and can be accessed by a web browser (Fig. 10.18). As an example, Fig. 10.18 displays a monitor divided into two portions where the left panel and right panel contain a Jupyter Notebook and the Dask interactive dashboard, respectively. The Jupyter Notebook starts the Dask client and its interactive dashboard at line 2. Then, it defines (lines 3-4), evaluates (line 5), and finally triggers (line 6, in progress and therefore displayed as *) the computations. The right portion of the monitor shows the Dask interactive dashboard during the ongoing process triggered at line 6 of the Jupyter Notebook.

Chapter 11

Scale Your Models in the Cloud

11.1 How to Scale your environment in the Cloud

Generally speaking, the term scalability refers to the ability of a system to manage a growing amount of work. As stated in the previous chapter, the achievement of compute or memory bounds when handling ML models requires scaling. Now moving to a cloud computing facility, the term scaling points to the ability to quickly and efficiently increase (or decrease) the capability of a computational resource to handle a model, that does not fit anymore the current resources (e.g., RAM, CPUs, and storage capabilities). In detail, the computational infrastructure could scale following two main strategies, i.e., scale up or scale out (ref).

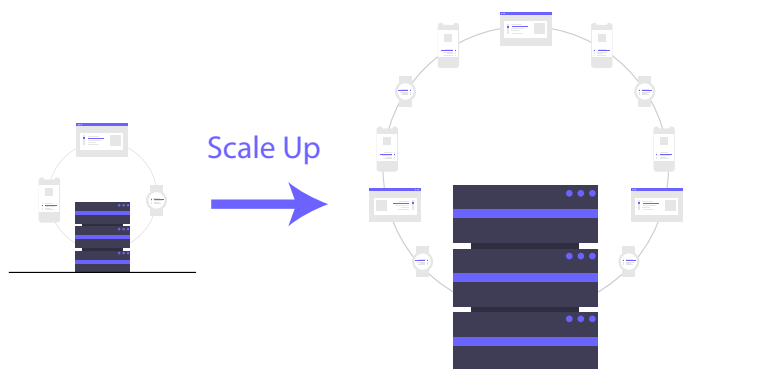


Fig. 11.1 Scaling Up

Scale Up

Scaling up, or vertical scaling, consists of replacing the current computational instance with something more powerful (Fig. 11.1).

As an example, we could increase the number of cores, the amount of memory, and the capability of the storage (Fig. 11.2).

Compute Optimized Instances by Amazon Web Services

Instance Size	vCPU	Memory (GiB)
r6a.large	2	16
r6a.xlarge	4	32
r6a.2xlarge	8	64
r6a.4xlarge	16	128
r6a.8xlarge	32	256
r6a.12xlarge	48	384
r6a.16xlarge	64	512
r6a.24xlarge	96	768
r6a.32xlarge	128	1024
r6a.48xlarge	192	1536
r6a.metal	192	1536

Fig. 11.2 Scaling Up

Scale Out

Scaling out, or horizontal scaling, consists of increasing the computational capability by replicating the instances and allowing them to work in parallel (Fig. 11.3).

11.2 Scaling in the Cloud: the Hard Way

For the scaling, the hard way consists of taking care of all the configurations and technical steps in either Amazon Web Services (AWS), Google Compute Engine (GCE), Microsoft Azure (MA), and other providers. Explaining how to master the scaling procedures in the cloud is far beyond the scope of the present book. Here I

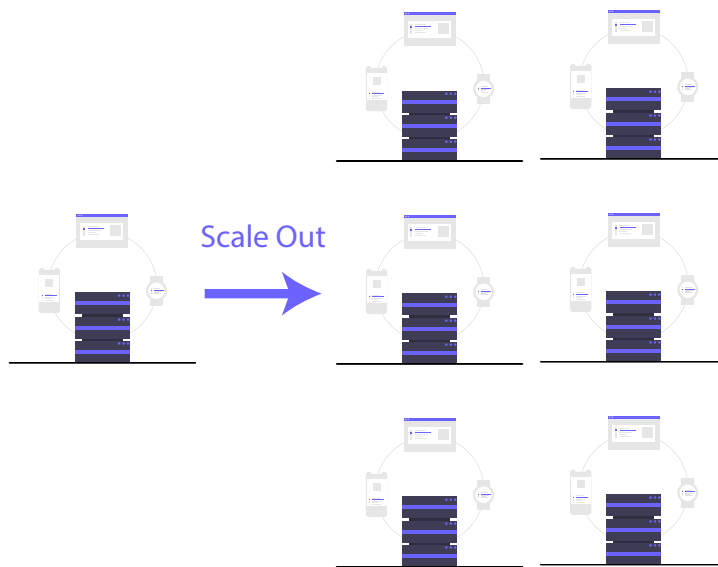


Fig. 11.3 Scaling Out

will briefly report some of the best practices and the reference go in deeper details if interests.

Scaling up is quite easy with all cloud providers. It simply consists of selecting larger or smaller instances to scale up and down (Fig. 11.2), respectively. Also, some providers offer specific services for Auto Scaling, e.g., “AWS Auto Scaling monitors your applications and automatically adjusts capacity to maintain steady, predictable performance at the lowest possible cost. Using AWS Auto Scaling, it’s easy to setup application scaling for multiple resources across multiple services in minutes. The service provides a simple, powerful user interface that lets you build scaling plans for resources including Amazon EC2 instances...”¹

On the contrary, the scaling out is not as straightforward as scaling up. Looking at the Dask documentation, they suggest the use of Kubernetes and Helm solutions. Kubernetes is “a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation.”² Helm is “an open source package manager for Kubernetes. It provides the ability to provide, share, and use software built for Kubernetes.”³ As reported in the Dask documentation, “it is easy to launch a Dask cluster and a Jupyter notebook server on cloud resources using Kubernetes and Helm.”⁴ I could agree, however, the

¹ <https://aws.amazon.com/autoscaling/>

² <https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/>

³ <https://helm.sh/docs/>

⁴ <https://docs.dask.org/en/stable/deploying-kubernetes-helm.html>

instructions reported in the Dask documentation assume that a Kubernetes cluster and Helm have been already installed and ready for use. The setup of a Kubernetes cluster and helm is not straightforward for a novice. You can find detailed instructions for many cloud providers, in the guide “Zero to JupyterHub.”⁵

11.3 Scaling in the Cloud: the Easy Way

Saturn Cloud

Saturn Cloud⁶ is a cloud based platform designed to support data scientists working with Python⁷, R⁸, Julia⁹, and other programming languages. Resources (Fig. 11.4) are the building blocks of the Saturn Cloud platform. In detail, the term resource refers to a complete computational and coding environment. Each resource is independent, so you can split out the different types of activities you’re doing. Saturn Cloud hosted solutions¹⁰ are a pay as you go services. It means that you will pay computational resources per hour. As an example, during the writing of the present book, Medium (2 vCPU and 4 GB of RAM) and V100-16XLarge (64 vCPU, 8 vGPU, and 488 GB of RAM) cost 0.06 \$ and 34.24 \$ per hour, respectively. A free hosted plan also exists with limited resources. In the next sections, we will benefit from the free hosted plan for the first step of scaling up. Then, I will show you the results on a Hosted Pro Plan, also providing you the details about the costs, in case you intend reproduce my findings.

Speed up GridSearchCV on Saturn Cloud

In section 8.3, we performed a GridSearchCV, i.e., an extensive search within the hyper-parameters governing the *Extremely Randomized Trees* (see code listing 8.9 and Tab. 8.1). The aim was finding the combination of hyper-parameters, providing the highest degree of accuracy. The combination of the selected hyper-parameters resulted in a grid of 48 models, each repeated three times through cross validation, for a total of 144 attempts. To note, running the code listing 8.9 in my MacBook pro, equipped with a 2.3 GHz Quad-Core Intel™ i7 and 32GB of RAM, lasted about 8 hours.

In Saturn Cloud, the free hosted plan allows a slight scaling up of the hardware supporting my MacBook pro using the 2X Large instance (i.e., 8 Cores and 64GB of

⁵ <https://zero-to-jupyterhub.readthedocs.io/en/latest/kubernetes/>

⁶ <https://saturncloud.io>

⁷ <https://www.python.org>

⁸ <https://www.r-project.org>

⁹ <https://julialang.org>

¹⁰ <https://saturncloud.io/plans/hosted/>

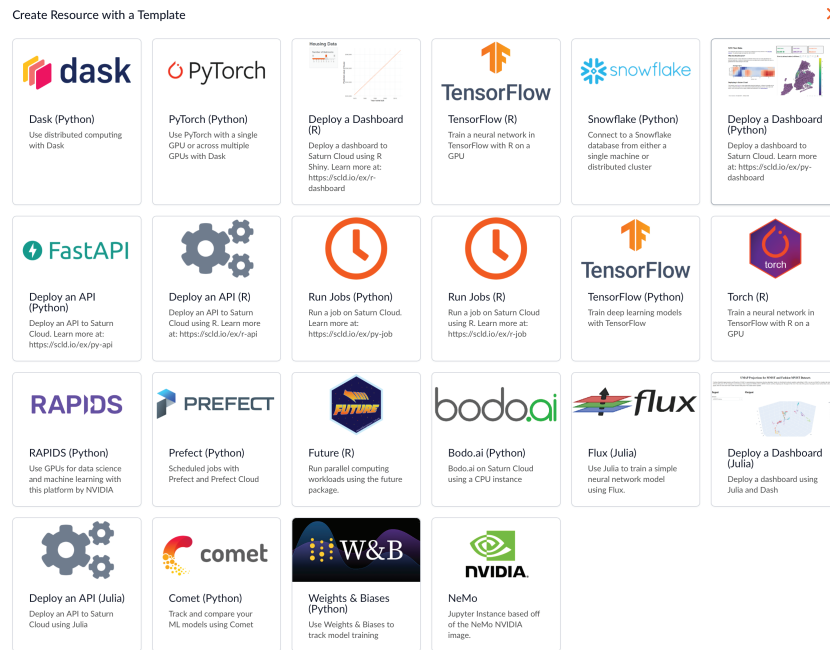


Fig. 11.4 Saturn Cloud computing templates

RAM). So let's try setting up a 2X Large instance and running the code listing 8.9. To start, register to Saturn Cloud and click on the New Python Server button (Fig. 11.5). It will start a guided procedure that allows the configuration of a new instance, ready for basic Python data analysis, machine learning, and, possibly, parallel processing with Dask.

Figures 11.6 and 11.7 show all the steps to configure the new instance. I suggest using a self-explanatory name, e.g., `scale_GridSearchCV_Joblib`, 100Gi of disk space, and the 2Xlarge instance. Also, remember to add `pytables` as extra package, installed using `Conda Install`. Pytables allow the manipulation, i.e., reading and saving, of HDF5 files. Leave all the other options untouched, and click `Create`.

The instance is now ready (Fig. 11.8). The next steps consist of starting the instances, create a new Jupyter Notebook, and upload the HDF5 file named `ml_data.h5` (Fig. 11.9). Finally, we are ready to replicate the code listing 8.9 in a 2Xlarge instance (Fig. 11.10), in Saturn Cloud. Note that the second block of code in (Fig. 11.10), simply reports the outputs on a log file named `data.log`. Figure 11.10 shows that the fitting, i.e., the block number 5, lasted 5 hours and 15 minutes, significantly better than the 8 hours of my MacBook pro.

Then, I activate a Hosted Pro Plan¹¹ and progressively scaled up the code reported in Fig. 11.10 to 8XLarge (i.e., 32 Cores and 256GB of RAM at the cost of 3.30 \$/hour)

¹¹ <https://saturncloud.io/plans/hosted/>

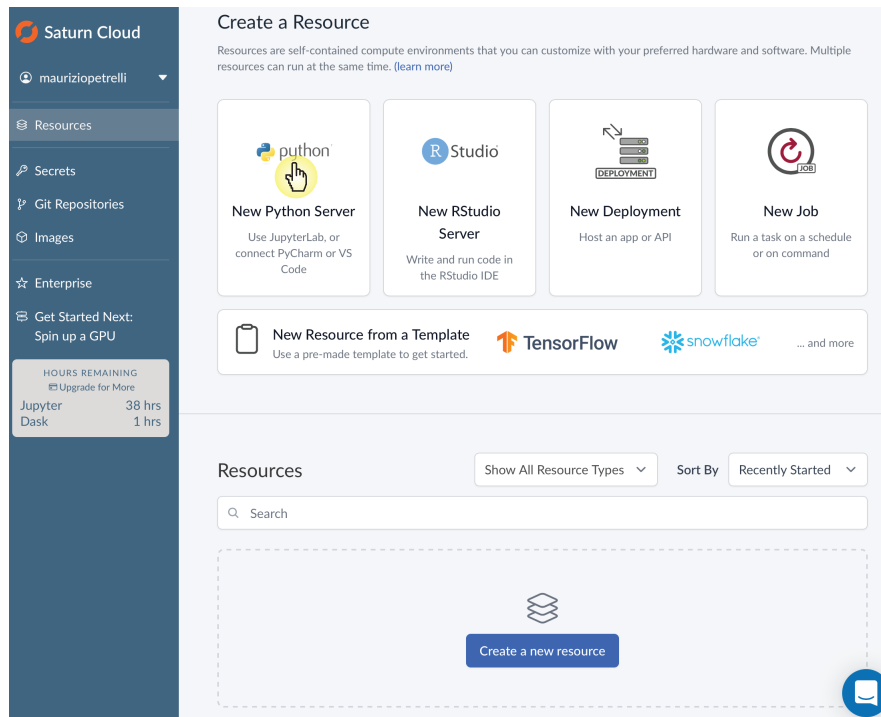


Fig. 11.5 Starting a New Python Server

and 16XLarge (i.e., 64 Cores and 512GB of RAM at the cost of 6.59 \$/hour) instances, improving the performances to ~ 2 hours and ~ 1 hour, respectively.

As a final step, I attempted to scaling out the code reported in Fig. 11.10. To achieve my goal, I created a dask cluster, i.e., by clicking *New Dask Cluster* (Fig. 11.8). It opens the Cluster configuration window (Fig. 11.11). In detail, I opted for a 16XLarge (i.e., 64 Cores and 512GB of RAM) scheduler and 4 8XLarge (i.e., 32 Cores and 256GB of RAM) workers. To run the *GridSearchCV* in the newly created Dask Cluster, the code reported in Fig. 11.10, only required minimal changes, all reported in Fig. 11.12. In detail, I imported the *SaturnCluster* from *dask_saturn* (block 1), allowed *n_jobs=-1* (i.e., nested parallelism) for both *ExtraTreesClassifier* and *GridSearchCV* (Block 4), defined the *SaturnCluster* client (Block 5), and run *Joblib* with *dask* as engine for the fitting (Block 6). In this final case, fitting the *GridSearchCV* lasted less than 30 minutes!

Create a Jupyter Server
✕

Jupyter servers let you interactively write code via JupyterLab, a terminal, or via SSH.

Overview
Show Advanced Options

Owner
Name

mauriziopetrelli / scale_GridSearchCV_Joblib

Hardware
Hide Advanced Options

The hardware your Jupyter server will run on.

Disk Space

100Gi

The Free plan is limited to 100GiB of disk space. Upgrade to Pro for unlimited resources.

Hardware

CPU ✔

An instance with only CPU processors.

GPU

An instance with both CPU and GPU processors.

Size

2XLarge - 8 cores - 64 GB RAM

Disabled options are not supported due to your account limit. To increase the limit, please contact your administrator.

Environment
Show Advanced Options

The software your Jupyter server will use. This includes libraries, packages, environment variables, and other attributes.

Image
Version

saturncloud/saturn
:
2022.04.01

Extra Packages

Extra packages are installed every time the resource starts up - right before the start script. Use spaces to separate packages.

If you find yourself adding the same packages to lots of resources, you may want to permanently add packages to a custom image instead. (?)

● Conda Install
Pip Install
Apt Packages

pytables

Solve with Mamba

If enabled, mamba will be used to solve the conda environment.

The packages together will run the following script:

mamba install pytables

Fig. 11.6 Setting-up the Python Server parameters

Git Repositories
Attach git repositories to connect your resource to your code.

Add a git repository ▼ New Git Repository

Remote URL	Actions
No Git Repositories Added.	

Additional features Show Advanced Options
Optional settings for your Jupyter server.

Allow SSH Connections
Use SSH to directly connect to the server, including through VSCode, PyCharm, and other tools (?)

Shutoff After
1 hour ▼

Disabled options are not supported in the Free plan. Upgrade to Pro for unlimited resources.

Create Cancel 🗨️

Fig. 11.7 Setting-up the Python Server parameters

Resources / mauriziotpetrelli / scale_GridSearchCV_Joblib

JUPYTER SERVER Logs Edit

mauriziotpetrelli / scale_GridSearchCV_Joblib
3019699baab34a2b9ea49291defb97d9

[Overview](#) [Environment](#) [Secrets](#) [Git Repos](#) [Manage](#)

Jupyter Server stopped ▶ Start

2XLarge - 8 cores - 64 GB RAM - 100i Disk

Metrics
Auto Shutoff: 1 hour
Spot Instance: No
SSH URL: (not enabled) (?)
App URL: (not enabled)

Jupyter Lab ▼

New Dask Cluster

Fig. 11.8 Starting the Python Server

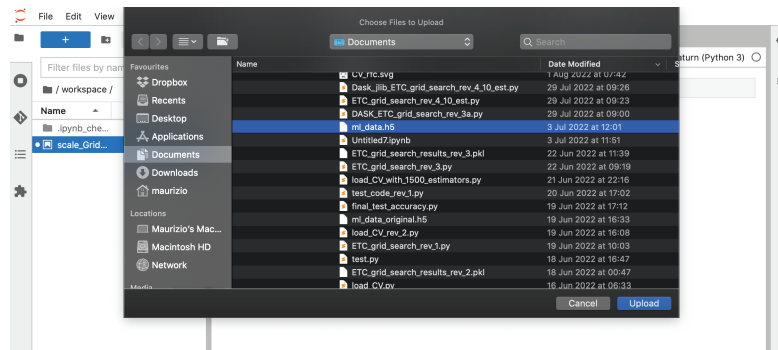


Fig. 11.9 Uploading a hdf5 file

```
[1]: import joblib as jb
import pandas as pd
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
Last executed at 2022-07-05 18:22:35 in 706ms

[2]: import logging
import sys

so = open("data.log", 'w', 10)
sys.stdout.echo = so
sys.stderr.echo = so

get_ipython().log.handlers[0].stream = so
get_ipython().log.setLevel(logging.INFO)
Last executed at 2022-07-05 18:22:36 in 4ms

[3]: X = pd.read_hdf('ml_data.h5', 'train').values
y = pd.read_hdf('ml_data.h5', 'train_target').values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=10, stratify=y)
Last executed at 2022-07-05 18:22:38 in 612ms

[4]: param_grid = {
    'classifier__criterion': ['entropy', 'gini'],
    'classifier__min_samples_split': [2, 5, 8, 10],
    'classifier__max_features': ['sqrt', 'log2', None],
    'classifier__class_weight': ['balanced', None]
}

clf = Pipeline([('scaler', StandardScaler()),
                ('classifier', ExtraTreesClassifier(n_estimators=250, n_jobs=-1))])

CV_rfc = GridSearchCV(estimator=clf, refit=True, param_grid=param_grid, cv=3, verbose=10)
Last executed at 2022-07-05 18:22:39 in 4ms

[5]: CV_rfc.fit(X_train, y_train)
Last executed at 2022-07-05 23:38:10 in 5h 15m 27s
Fitting 3 folds for each of 48 candidates, totalling 144 fits
[CV 1/3; 1/48] START classifier__class_weight=balanced, classifier__criterion=entropy, classifier__max_features=sqrt, classifier__min_samples_split=2
[CV 1/3; 1/48] END classifier__class_weight=balanced, classifier__criterion=entropy, classifier__max_features=sqrt, classifier__min_samples_split=2; score=0.953 total time= 1.5min
[CV 2/3; 1/48] START classifier__class_weight=balanced, classifier__criterion=entropy, classifier__max_features=sqrt, classifier__min_samples_split=2
[CV 2/3; 1/48] END classifier__class_weight=balanced, classifier__criterion=entropy, classifier__max_features=sqrt, classifier__min_samples_split=2; score=0.954 total time= 1.5min
[CV 3/3; 1/48] START classifier__class_weight=balanced, classifier__criterion=entropy, classifier__max_features=sqrt, classifier__min_samples_split=2
[CV 3/3; 1/48] END classifier__class_weight=balanced, classifier__criterion=entropy, classifier__max_features=sqrt.

[6]: jb.dump(CV_rfc, 'ETC_grid_search_results_rev_3_baseline.pkl')
Last executed at 2022-07-05 23:51:53 in 5.98s

[6]: ['ETC_grid_search_results_rev_3_baseline.pkl']

[8]: CV_rfc.best_params_
Last executed at 2022-07-05 23:53:25 in 4ms

[8]: {'classifier__class_weight': 'balanced',
'classifier__criterion': 'entropy',
'classifier__max_features': None,
'classifier__min_samples_split': 2}
```

Fig. 11.10 Scaling Up the *GridSearchCV*

Create Dask Cluster

Attached to Jupyter Server
mauripetre/example-dask

When you create a Dask cluster for a resource (Jupyter server, RStudio server, Deployment, Perfect Cloud flow), that cluster's workers will use the same image and code from the resource.

Scheduler Hardware

CPU
An instance with only CPU processors.

GPU
An instance with both CPU and GPU processors.

Size
16XLarge - 64 cores - 512 GB RAM

Disabled options are not supported due to your account limit. To increase the limit, please contact your administrator.

Worker Hardware

CPU
An instance with only CPU processors.

GPU
An instance with both CPU and GPU processors.

Size
8XLarge - 32 cores - 256 GB RAM

Disabled options are not supported due to your account limit. To increase the limit, please contact your administrator.

Spot Instance
This requests worker of specified size as Spot Instance. Spot Instances are less expensive, but may be shut down at any time (with a two-minute warning). (7)

Number of Workers (n_workers)
4

Disabled options are not supported in the Free plan. Upgrade to Pro for unlimited resources.

Use the optimal number of processes

Number of Worker Processes (nprocs)
1

Number of Worker Threads (nthreads)
32

Subdomain
community.saturnenterprise.io

Set a custom subdomain for the Dask cluster URL. Must be globally unique. If unset, a default will be chosen based on name.

Create Cancel

Fig. 11.11 Setting-up a new Dask Cluster

```
[1]: import joblib as jb
import pandas as pd
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from dask_saturm import SaturnCluster
from dask.distributed import Client
Last executed at 2022-08-06 18:34:28 in 928ms
```

```
[2]: import logging
import sys

so = open("data.log", 'w', 10)
sys.stdout.echo = so
sys.stderr.echo = so

get_ipython().log.handlers[0].stream = so
get_ipython().log.setLevel(logging.INFO)
Last executed at 2022-08-06 18:34:29 in 5ms
```

```
[3]: X = pd.read_hdf('ml_data.h5', 'train').values
y = pd.read_hdf('ml_data.h5', 'train_target').values

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=10, stratify=y)
Last executed at 2022-08-06 18:34:31 in 622ms
```

```
[4]: param_grid = {
    'classifier__criterion': ['entropy', 'gini'],
    'classifier__min_samples_split': [2, 5, 8, 10],
    'classifier__max_features': ['sqrt', 'log2', None],
    'classifier__class_weight': ['balanced', None]
}

clf = Pipeline([['scaler', StandardScaler()],
                ('classifier', ExtraTreesClassifier(n_estimators=250, n_jobs=-1))]
)

CV_rfc = GridSearchCV(estimator=clf, refit=True, param_grid=param_grid, cv=3, verbose=1, n_jobs=-1)
Last executed at 2022-08-06 18:36:47 in 4ms
```

```
[5]: client = Client(SaturnCluster())
Last executed at 2022-08-06 18:39:01 in 309ms

INFO:dask-saturm:Cluster is ready
INFO:dask-saturm:Registering default plugins
INFO:dask-saturm:Success!
```

```
[6]: with jb.parallel_backend("dask"):
    _ = CV_rfc.fit(X_train, y_train)
Last executed at 2022-08-06 19:07:58 in 28m 53.79s

Fitting 3 folds for each of 48 candidates, totalling 144 fits
```

```
[7]: jb.dump(_, 'ETC_grid_search_results_rev_3_dask.pkl')
Last executed at 2022-08-06 19:12:53 in 27.36s
```

```
[7]: ['ETC_grid_search_results_rev_3_dask.pkl']
```

```
[8]: _._best_params_
Last executed at 2022-08-06 19:13:52 in 4ms
```

```
[8]: {'classifier__class_weight': 'balanced',
      'classifier__criterion': 'entropy',
      'classifier__max_features': None,
      'classifier__min_samples_split': 2}
```

Fig. 11.12 Scaling Out the *GridSearchCV*

Part V
Next Step: Deep Learning

Chapter 12

Introduction to Deep Learning

12.1 What does Deep Learning mean?

As we introduced in chapter 1, machine learning algorithms learn knowledge by extracting patterns from data.

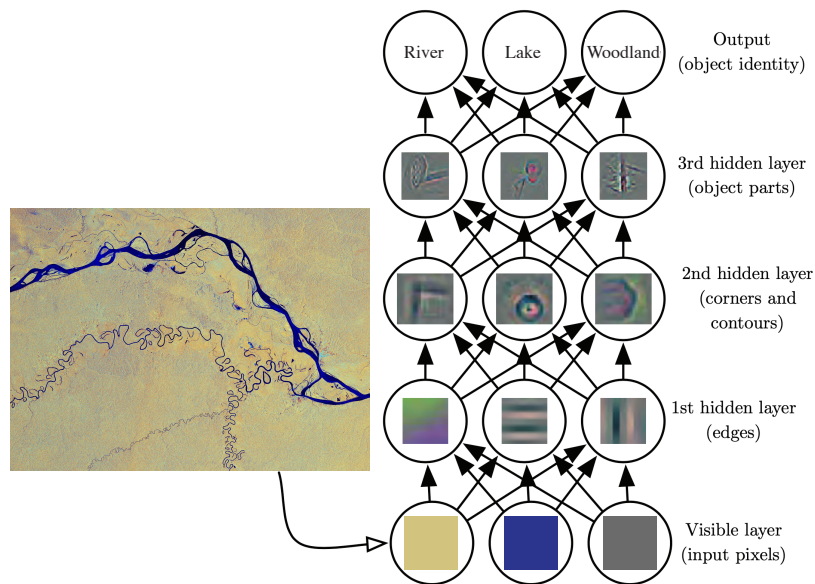


Fig. 12.1 Illustration of a deep learning, multilayer perceptron model. Modified from Goodfellow et al. (2016). The image comes from Copernicus Sentinel-1 mission and shows the Amazon River meandering.²

² https://www.esa.int/ESA_Multimedia/Images/2020/09/Amazon_River

In other words, they try to map the representation provided by the investigated features to produce an output (Goodfellow et al., 2016).

Therefore, features are central in ML since they provide the information to build a representation. However, simply mapping a representation to deliver an output is, often, not enough. Therefore, we need training machine learning systems to discover not only the mapping from representation to output but also the representation itself (Goodfellow et al., 2016). This approach is known as representation learning. In complex problems (e.g., characterized by many features or extremely large data sets), successfully applying the representation learning is not straightforward.

“Deep learning solves this central problem in representation learning by introducing representations that are expressed in terms of other, simpler representations. Deep learning enables the computer to build complex concepts out of simpler concepts” (Goodfellow et al., 2016).

A typical example of deep learning is the multilayer perceptron, i.e. a mathematical function that maps a set of inputs to output values (Goodfellow et al., 2016). The function is formed by composing many simpler functions (Fig. 12.1). In detail, Figure 12.1 depicts how a deep learning method can represent the concept of an image by combining simpler notions, such as corners and contours, which are in turn defined in terms of edges (Goodfellow et al., 2016). In Fig. 12.1, the input feeds the visible layer. Then a series of hidden layers progressively extract and elaborate abstract features from the initial inputs. The final layer provides the output, e.g., the result of mapping the representation that has been developed during the learning process (Goodfellow et al., 2016).

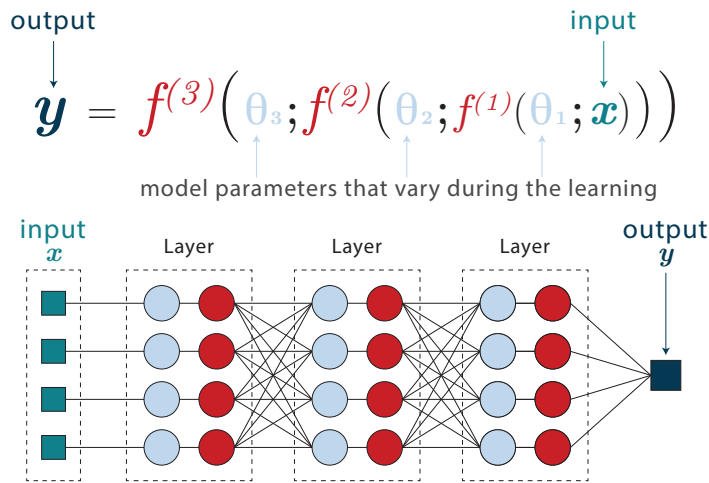


Fig. 12.2 Example of 3 layer feedforward networks or multilayer perceptrons.

From the mathematical point of view, deep feedforward networks (or multilayer perceptrons) aim at approximating some function f^* (Goodfellow et al., 2016). In detail, it defines a mapping $\mathbf{y} = f(\mathbf{x}; \boldsymbol{\theta})$ and learns the value of the parameters $\boldsymbol{\theta}$ that result in the best function approximation (Goodfellow et al., 2016). Why feedforward? Because data flow through the function from the input \mathbf{x} , through the intermediate computations used to define f , and finally to the output \mathbf{y} . Why networks? Because they are typically expressed by combining many different functions. For example, we might combine three functions $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$ in a chain to define $f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$ (Goodfellow et al., 2016). In detail, $f^{(1)}$ is the first layer of the network, $f^{(2)}$ is the second layer, and so on (Goodfellow et al., 2016). The overall length of the chain defines the depth of the model. That's why they are deep. The final layer of a feedforward network provides us with the output. During the training process, we adjust $\boldsymbol{\theta}$ parameters in $f(\mathbf{x}; \boldsymbol{\theta})$ to match $f^*(\mathbf{x})$ (Goodfellow et al., 2016).

12.2 PyTorch

“PyTorch is an optimized tensor library for deep learning using GPUs and CPUs.”³ Tensors, i.e., multidimensional arrays, are at the base of PyTorch. Also, PyTorch hosts the *autograd* engine (i.e., *torch.autograd*). It has the ability to effectively compute derivatives, even providing complex data structures. The other PyTorch modules mainly bases on tensors and on the *autograd* engine. As an example, the *torch.nn* module provides common neural network layers and other architectural components. The *torch.optim* implements *state of the art* optimization strategies for the learning process (Imambi et al., 2021).

12.3 PyTorch Tensors

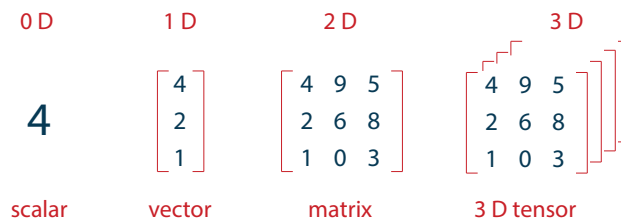


Fig. 12.3 Vectors, matrices, tensors.

³ <https://pytorch.org/docs/stable/index.html>

```
[1]: import torch
Last executed at 2022-08-15 09:51:44 in 7.11s
```

1D tensor → vector

```
[2]: zeros = torch.zeros(6)
print(zeros)
Last executed at 2022-08-15 09:40:55 in 84ms
tensor([0., 0., 0., 0., 0., 0.])
```

2D tensor → matrix

```
[3]: ones = torch.ones(2, 3)
print(ones)
Last executed at 2022-08-15 09:40:56 in 27ms
tensor([[1., 1., 1.],
        [1., 1., 1.]])
```

3D tensor

```
[4]: random1 = torch.randn(2, 3, 3)
print(random1)
Last executed at 2022-08-15 09:40:57 in 8ms
tensor([[[ 0.2335, -1.5447,  1.4459],
         [ 0.9062,  0.9170,  2.1128],
         [-0.8420,  0.0852,  1.4125]],

        [[ 2.5360,  0.0471, -0.7606],
         [ 0.3327,  0.4224,  0.4928],
         [-0.3284, -0.5549,  0.2324]])
```

Tensor from a Python list

```
[5]: my_list = [3, 6, 8, 6, 8, 9]
tensor1 = torch.tensor(my_list, dtype = torch.float)
print(tensor1)
Last executed at 2022-08-15 09:40:58 in 5ms
tensor([3., 6., 8., 6., 8., 9.]])
```

Tensor from NumPy array

```
[6]: import numpy as np

np_array = np.array([3, 4, 5, 7, 9])
tensor2 = torch.from_numpy(np_array)
tensor3 = torch.tensor(np_array)
print(tensor2)
print(tensor3)
Last executed at 2022-08-15 09:41:02 in 4ms
tensor([3, 4, 5, 7, 9])
tensor([3, 4, 5, 7, 9])
```

Fig. 12.4 Vectors, matrices, tensors.

PyTorch tensors are multidimensional arrays (Fig. 12.3). They are not conceptually different from those in NumPy. However, differently than NumPy arrays, they can: a) perform accelerated operations on graphical processing units (GPUs), b) natively works on distributed environments, and c) keep track of the graph of operations that created them (Imambi et al., 2021). The initialization of PyTorch tensors mimics those in NumPy, and Numpy arrays can be easily imported as PyTorch tensors (Fig. 12.4).

```
[1]: import torch
Last executed at 2022-08-17 16:45:10 in 7.27s

Working on the GPU

[2]: torch.cuda.is_available()
Last executed at 2022-08-17 16:45:10 in 34ms

[2]: True

[3]: random_cuda1 = torch.randn(500000000, device='cuda')
random_cuda2 = torch.randn(500000000, device='cuda')
Last executed at 2022-08-17 16:45:17 in 7.31s

[4]: power_cuda = random_cuda1 ** random_cuda2
Last executed at 2022-08-17 16:45:17 in 13ms

[5]: random_cpu1 = torch.randn(500000000, device='cpu')
random_cpu2 = torch.randn(500000000, device='cpu')
Last executed at 2022-08-17 16:45:26 in 8.55s

[6]: power_cpu = random_cpu1 ** random_cpu2
Last executed at 2022-08-17 16:45:29 in 3.10s
```

Fig. 12.5 Vectors, matrices, tensors.

By default, PyTorch tensors live on the CPU. However, they can be easily defined on the GPU, if available (block 2 of Fig. 12.5), by using the *device* parameter (i.e., *device='cuda'*, block 3 of Fig. 12.5). To note, blocks 3 to 6 in Fig. 12.5 simply highlight that the power operation performed on the *'cuda'*, i.e., GPU, device lasts 7 ms only. Much faster than the ~3 seconds required to execute the same operation in the CPU device. Please keep in your mind that cross-GPU operations are not allowed by default, with the exception of *copy_()* and other methods with copy-like functionality.

12.4 Structuring a feedforward network in PyTorch

Figure 12.6 shows how to develop the feedforward neural networks, i.e., a multilayer perceptron, depicted in Fig. 12.2 in PyTorch.

```
[1]: import torch
      from torch import nn
      Last executed at 2022-08-17 16:34:00 in 571ms

[2]: class MultilayerPerceptron(nn.Module):
      ...
      Example of Multilayer Perceptron
      ...
      def __init__(self):
          super().__init__()
          self.layers = nn.Sequential(
              nn.Linear(4, 4),
              nn.ReLU(),
              nn.Linear(4, 4),
              nn.ReLU(),
              nn.Linear(4, 1)
          )

      def forward(self, x):
          return self.layers(x)
      Last executed at 2022-08-17 16:34:00 in 4ms

[3]: device = "cuda" if torch.cuda.is_available() else "cpu"
      print(f"Using {device} device")
      Last executed at 2022-08-17 16:34:00 in 4ms
      Using cpu device

[4]: model = MultilayerPerceptron().to(device)
      print(model)
      Last executed at 2022-08-17 16:34:00 in 32ms
      MultilayerPerceptron(
        (layers): Sequential(
          (0): Linear(in_features=4, out_features=4, bias=True)
          (1): ReLU()
          (2): Linear(in_features=4, out_features=4, bias=True)
          (3): ReLU()
          (4): Linear(in_features=4, out_features=1, bias=True)
        )
      )
```

Fig. 12.6 Developing a multilayer perceptron in PyTorch

In detail, it consists of an input layer (i.e., layer 1) accepting input vectors with four features. ReLu functions processes the input features and forward the results to a hidden layer (i.e., layer 2) characterized by the same number of neurons (i.e., 4) as input vectors and the same activation function (i.e., the ReLu function). Finally, the output layer returns a scalar, i.e., a number, as output.

Programmatically, a neural network in PyTorch is a module with a nested structure. In other words, it consists of a module that contains other modules (i.e., layers). The model can live either in the CPU or in the GPU (Blocks 3 and 4 in Fig. 12.6), if available.

12.5 How to train a feedforward network

12.5.1 The universal approximation theorem

The universal approximation theorem (Hornik et al., 1989; Cybenko, 1989) states that feedforward networks with a linear output layer and at least one hidden layer can approximate any continuous function on a closed and bounded subset of \mathbb{R}^n (Goodfellow et al., 2016). It means that we can state that feedforward networks with hidden layers are universal approximators (Goodfellow et al., 2016). In other words, “the universal approximation theorem means that regardless of what function we are trying to learn, we know that a large MLP will be able to represent this function (Goodfellow et al., 2016).” However, despite what is affirmed by the universal approximation theorem, we are not ensured that the training process will be able to correctly learn the target function (Goodfellow et al., 2016). As an example, the optimization algorithm used for training may not be able to find the correct values for the theta-parameters that describe the desired function. Also, the training process might choose a wrong function because of the occurrence of overfitting (Goodfellow et al., 2016). To avoid these issues, we need to do our best to find: 1) a robust loss function $L(\theta)$; 2) a strategy to compute the gradient with respect to model parameters, i.e., $\Delta_{\theta}L(\theta)$, of $L(\theta)$; 3) an efficient optimization algorithm to descend $\Delta_{\theta}L(\theta)$ and find the minimum of $L(\theta)$.

12.5.2 Loss Functions in PyTorch

A loss function (or cost function) computes a numerical value that the learning process will attempt to minimize. Typically, a loss function compare (e.g., by subtraction) the desired outputs (i.e., the labels) and the current outputs of our model (Stevens et al., 2020). Table 12.1 report the loss function available in PyTorch.

12.5.3 The Back-Propagation and its implementation in PyTorch

In feedforward neural networks, the information starts from the input \mathbf{x} , flows through the hidden layers, and finally produces an output \mathbf{y} (Goodfellow et al., 2016). The name of this process is forward propagation. At the beginning of the training, the

Table 12.1 Loss functions in PyTorch: <https://bit.ly/pyt-loss-functions>.

Loss Function	Description
nn.L1Loss	Loss function based on the mean absolute error (MAE)
nn.MSELoss	Loss function based on the mean squared error (squared L2 norm)
nn.CrossEntropyLoss	It computes the cross entropy loss between input and target
nn.CTCLoss	Connectionist Temporal Classification loss
nn.NLLLoss	The negative log likelihood loss
nn.PoissonNLLoss	Negative log likelihood loss with Poisson distribution of target
nn.GaussianNLLoss	Gaussian negative log likelihood loss
nn.KLDivLoss	The Kullback-Leibler divergence loss
nn.BCELoss	Binary Cross Entropy between the target and the input probabilities
nn.BCEWithLogitsLoss	It combines a Sigmoid layer and the BCELoss in one single class
nn.MarginRankingLoss	It measures the loss given inputs x_1, x_2 , two $1D$ mini-batch or $0D$ Tensors, and a label $1D$ mini-batch or $0D$ Tensor y (containing 1 or -1)
nn.HingeEmbeddingLoss	It measures the loss given an input tensor x and a labels tensor y (containing 1 or -1)
nn.MultiLabelMarginLoss	It optimizes a multi-class multi-classification hinge loss (margin-based loss)
nn.HuberLoss	It creates a criterion that uses a squared term if the absolute element-wise error falls below delta and a delta-scaled L1 term otherwise
nn.SmoothL1Loss	It creates a criterion that uses a squared term if the absolute element-wise error falls below beta and an L1 term otherwise
nn.SoftMarginLoss	It creates a criterion that optimizes a two-class classification logistic loss between input tensor x and target tensor y (containing 1 or -1)
nn.MultiLabelSoftMarginLoss	It optimizes a multi-label one-versus-all loss based on max-entropy, between input x and target y of size (N, C) .
nn.CosineEmbeddingLoss	It measures the loss given input tensors x_1, x_2 and a Tensor label y with values 1 or -1.
nn.MultiMarginLoss	It Creates optimizes a multi-class classification hinge loss (margin-based loss)
nn.TripletMarginLoss	It measures the triplet loss given an input tensors x_1, x_2, x_3 and a margin with a value greater than 0
nn.TripletMarginWithDistanceLoss	It measures the triplet loss given input tensors a, p , and n (representing anchor, positive, and negative examples, respectively), and a nonnegative, real-valued function ('distance function') used to compute the relationship between the anchor and positive example ('positive distance') and the anchor and negative example ('negative distance')

forward propagation produces an output \mathbf{y} and an associated cost function $J(\boldsymbol{\theta})$ that relies on non-optimized θ parameters (Goodfellow et al., 2016).

The back-propagation algorithm allows computing the gradient of $L(\boldsymbol{\theta})$ by propagating the information from the output, i.e., the cost function, backward through the network (Goodfellow et al., 2016). Please note that the back-propagation only allows the definition of the gradient of $L(\boldsymbol{\theta})$. Then we need an optimization algorithm, e.g., the Stochastic Gradient Descent (section 7.5), to perform the learning along this gradient (Goodfellow et al., 2016). Describing in detail the back-propagation algorithm is beyond the scope of the present book. Please refer to Goodfellow et al. (2016) or other specialized books for further details.

The `torch.autograd` is PyTorch's automatic differentiation engine. It defines a directed acyclic graph whose leaves are the input tensors and roots are the output tensors. That way, it allows the computation of gradients using the chain rule.

12.5.4 Optimization

Once defined The `optim` submodule of `torch` (i.e. `torch.optim`) stores the optimization algorithms (Table 12.2).

Table 12.2 Optimization Algorithms in PyTorch: <https://bit.ly/pytorch-optim>.

Optimization Algorithm	Description
Adadelta	Implements Adadelta algorithm
Adagrad	Implements Adagrad algorithm
Adam	Implements Adam algorithm
AdamW	Implements AdamW algorithm
SparseAdam	Implements lazy version of Adam algorithm suitable for sparse tensors
Adamax	Implements Adamax algorithm (a variant of Adam based on infinity norm)
ASGD	Implements Averaged Stochastic Gradient Descent
L-BFGS	Implements L-BFGS algorithm, heavily inspired by minFunc
NAdam	Implements NAdam algorithm
RAdam	Implements RAdam algorithm
RMSprop	Implements RMSprop algorithm
Rprop	Implements the resilient backpropagation algorithm
SGD	Implements stochastic gradient descent (optionally with momentum)

12.5.5 Network Architectures

In this section, I provide a quick overview of some popular neural network architectures.

Multilayer Perceptron

It is the neural network structure depicted in Fig. 12.2. It consists of fully connected layers of perceptrons (i.e., artificial neurons). Selecting the optimal number of hidden layers is not always straightforward. It is commonly driven by background knowledge and experimentation (Hastie et al., 2017). Hastie et al. (2017) report that “it is better to have too many hidden units than too few. With too few hidden units, the model might not have enough flexibility to capture the nonlinearities in the data; with too many hidden units, the extra weights can be shrunk toward zero if appropriate regularization is used”. For common applications, the number of hidden layers typically ranges between 5 and 100 (Hastie et al., 2017). It is notable that most of the machine learning models described in Chapter 7, e.g., support vector machines or the logistic regression, can be simulated with multilayer perceptrons containing one or two layers, only (Aggarwal, 2018).

Radial Basis Function Networks

Radial basis function (RBF) networks consist of shallow, i.e., two layers only, neural networks where the first and the second layers are unsupervised and supervised, respectively (Aggarwal, 2018). In detail, RBF networks are based on Cover’s theorem on the separability of patterns (Cover, 1965), stating that pattern classification problems are more likely to be linearly separable when cast into a high-dimensional space with a nonlinear transformation. The idea behind RBF networks is close to that of nearest-neighbor classifiers with the addition of a supervised step in the second layer (Aggarwal, 2018). Also, they are similar to support vector machines trained with radial basis functions as the kernel. However, RBF networks are more general than kernel support vector machines (Aggarwal, 2018).

Restricted Boltzmann Machines

Restricted Boltzmann machines (RBMs) are unsupervised neural network architectures relying on the concept of energy minimization (Fischer & Igel, 2012). RBMs have been introduced in the 1980s (Aggarwal, 2018). However, the increase in computational power and the development of new learning strategies made RBMs significantly more appealing in recent times than in the 1980s (Fischer & Igel, 2012). RBMs are notably effective for creating generative models (Fischer & Igel, 2012), and they are nearly related to probabilistic graphical models (Koller & Friedman,

2009). Also, RBMs have been proposed as building blocks of the so-called deep belief networks (DBNs, Hinton et al., 2006). The training of an RBM is rather different from that of a feed-forward network since it cannot use backpropagation (Fischer & Igel, 2012). On the contrary, RBMs rely on Monte Carlo sampling to perform the training (Fischer & Igel, 2012).

Recurrent Neural Networks

Recurrent neural networks (RNNs) aim at investigating sequential data like text sentences, time series, and additional discrete sequences (Abraham and Tyagi, 2022). An important point about RNNs is that they account for the potential dependence of subsequent inputs from previous ones, making them well-suitable, for example, time series forecasting or speech recognition (Aggarwal, 2018; Kumar & Abraham, 2022). They use a specific backpropagation algorithm named backpropagation through time (BPTT; Aggarwal, 2018). It accounts for sharing and temporal length when updating the weights during the learning process. As a drawback, the optimization and training processes of RNNs are not straightforward, making them difficult to access, especially for novices (Aggarwal, 2018; Kumar & Abraham, 2022). Specialized variants of the recurrent neural network architecture have also been proposed to solve specific problems, such as the handling of long-term dependencies using long short-term memory (LSTM) networks (Hochreiter & Schmidhuber, 1997).

Convolutional Neural Networks

Convolutional neural networks (CNNs) are biologically inspired networks finding applications in video and speech recognition, recommendation systems, image classification and segmentation, natural language processing, and time series forecasting (e.g., Yamashita et al., 2018). A CNN mimics the visual cortex functionalities of animals (Fukushima, 1980) and aims at “automatically and adaptively learn spatial hierarchies of features through backpropagation by using multiple building blocks, such as convolution layers, pooling layers, and fully connected layers” (Fukushima, 1980).

CNNs are well suited to process grid-shaped data, like RGB images or spectral maps, by using three main categories of layers: convolution, pooling, and fully connected layers (Fukushima, 1980). The first two layers, i.e., convolution and pooling, fulfill feature extraction, and the third, i.e., fully connected layers, map the extracted features into the final output. Convolution layers play a fundamental role in CNNs (Yamashita et al., 2018).

Convolutional layers are the places where the majority of computation occurs. They typically consist of three components: input data, a filter (or kernel), and a feature map (Yamashita et al., 2018). To better understand, please consider the example reported in Fig. 12.7, where the input and the kernel are a 6x6 and 3x3 array, respectively. The output, i.e., a 4x4 array named feature map, activation map,

or convolved feature, derives by the systematic application of the filter, i.e. a dot product, to different portions of the input. After each convolution, the CNN applies an activation function, e.g., a Rectified Linear Unit (ReLU) to the output, and then moves to the next layer (Yamashita et al., 2018).

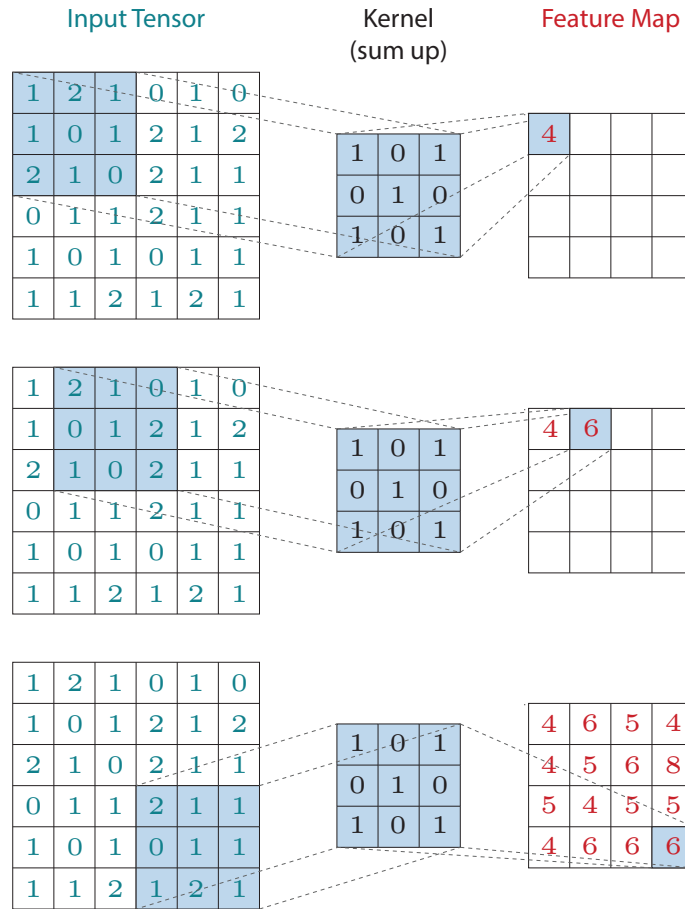


Fig. 12.7 Example of Convolution

Pooling layers perform a dimensionality reduction (or downsampling) step, reducing the number of parameters of their input. They typically consist of a filter that applies an aggregation function, e.g., the max or the average pooling (Fukushima, 1980). The max pooling selects the pixel with the maximum value of the filter and sends it to the output array. Similarly, the average pooling calculates the average value within the filter and sends it to the output array. If you complain that a huge amount of information is lost in pooling layers, you are right. However, they help in reducing the complexity of the model, improve its efficiency, and limit the risk of overfitting

(Fukushima, 1980). Finally, fully connected layers mimic a multilayer perceptron, allowing many ML tasks. As example, CNNs have been largely used in semantic image segmentation Badrinarayanan et al., 2017; Long et al., 2015; Milletari et al., 2016. Semantic image segmentation consists of identifying the areas, i.e., the pixels, of the image occupied by a specific subject, e.g., a person as in the case of Fig. 12.8.

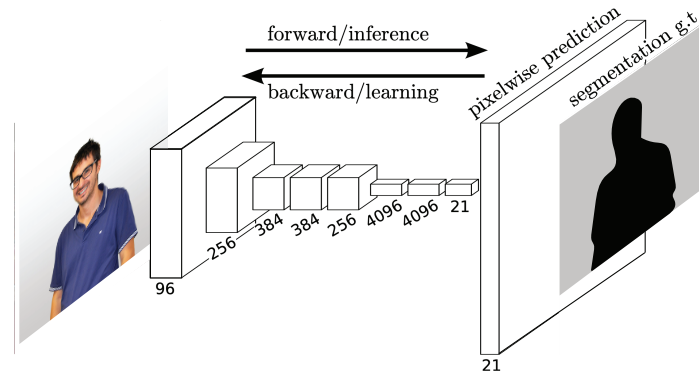


Fig. 12.8 Convolutional Neural Networks for image segmentation. Modified from Long et al., 2015

12.6 Example Application

The problem

As example application of deep learning potentials in the Earth Sciences, I report the training and validation of a CNN to identify building footprints from satellite records.

The problem falls in the specific sub-field of ML classification named semantic image segmentation (e.g., Fig. 12.8). In this specific case, we aim at identifying the areas, i.e., the pixels, of the image occupied by buildings in the Aerial Image Labeling data set (Maggiori et al., 2017) (e.g., Fig. 12.9 - left panel). In detail, Fig. 12.9 (right panel) shows the solution to the problem as mask where the white and black colors define building and non-building areas, respectively. The question is: Can we train a CNN to achieve the solution reported in Fig. 12.9 (right panel)? To attempt a simplified solution, I will train the U-Net CNN (Ronneberger et al., 2015) using PyTorch.



Fig. 12.9 The Aerial Image Labeling data set (Maggiori et al., 2017)

The Data set and Pre-Processing

As a starting point, I downloaded the Aerial Image Labeling data set (Maggiori et al., 2017). It consists of 360 orthorectified RGB images, linked to official cadastral records (Maggiori et al., 2017). The whole data set covers several areas, for example, Austin (USA), Chicago (USA), Vienna (Austria), East and West Tyrol (Austria), San Francisco (USA), and Innsbruck (Austria). The lateral resolution is 0.3 m, and each tile is 5000x5000 pixels (Maggiori et al., 2017). For 180 tiles, a mask containing two semantic classes, i.e., building and not-building, is also provided (Maggiori et al., 2017). For the case study provided in the present section, I selected 10 tiles from Austin. For each tile, I also collected the associated masks to train and validate the model. From each tile, I extracted 25 images, i.e., 1000x1000 pixels each, dividing it with a 5x5 grid. I executed the same operation for each mask. The resulting data set consists of 245 images and 245 masks. Then, I split the data set into two parts used for the training (220) and the validation (25), respectively.

The U-Net Architecture

The U-Net is a “fully convolutional network” (FCN; Long et al., 2015). The main concept behind FCNs is to take an input of arbitrary size and produce correspondingly-sized output with efficient inference and learning (Long et al., 2015).

Fig. 12.10 depicts the U-Net architecture. It consists of a contracting network (left side), followed by an expansive path (right side; Ronneberger et al., 2015). The contracting path applies a sequence of two 3x3 convolutions, each followed by a rectified linear unit (ReLU) and a 2x2 max pooling (Ronneberger et al., 2015). Then, in the expansive path, the U-Net performs an upsampling of the feature map, followed by a 2x2 convolution (‘up-convolution’), and two 3x3 convolutions, each followed by a ReLU (Ronneberger et al., 2015). The final layer applies a 1x1 convolution to map

each 64-component feature vector to the desired number of classes (Ronneberger et al., 2015). The code listing 12.1 shows a PyTorch implementation of the U-Net.

```

1 """ Full assembly of the parts to form the complete network """
2
3 from .UNET_parts import *
4
5
6 class UNet(nn.Module):
7     def __init__(self, n_channels, n_classes, bilinear=False):
8         super(UNet, self).__init__()
9         self.n_channels = n_channels
10        self.n_classes = n_classes
11        self.bilinear = bilinear
12
13        self.inc = DoubleConv(n_channels, 64)
14        self.down1 = Down(64, 128)
15        self.down2 = Down(128, 256)
16        self.down3 = Down(256, 512)
17        factor = 2 if bilinear else 1
18        self.down4 = Down(512, 1024 // factor)
19        self.up1 = Up(1024, 512 // factor, bilinear)
20        self.up2 = Up(512, 256 // factor, bilinear)
21        self.up3 = Up(256, 128 // factor, bilinear)
22        self.up4 = Up(128, 64, bilinear)
23        self.outc = OutConv(64, n_classes)
24
25    def forward(self, x):
26        x1 = self.inc(x)
27        x2 = self.down1(x1)
28        x3 = self.down2(x2)
29        x4 = self.down3(x3)
30        x5 = self.down4(x4)
31        x = self.up1(x5, x4)
32        x = self.up2(x, x3)
33        x = self.up3(x, x2)
34        x = self.up4(x, x1)
35        logits = self.outc(x)
36        return logits

```

Listing 12.1 Plotting the results of the Monte Carlo simulation.

Results

Figure 12.11 shows the application of the trained model (1260 epochs) to one of the 25 validation images extracted from the original data set. In detail, the top-right panel of Fig. 12.11 shows the original image, i.e., the input RGB matrix. Also, the top-left panel reports the building/non-building mask. Please take in mind that we

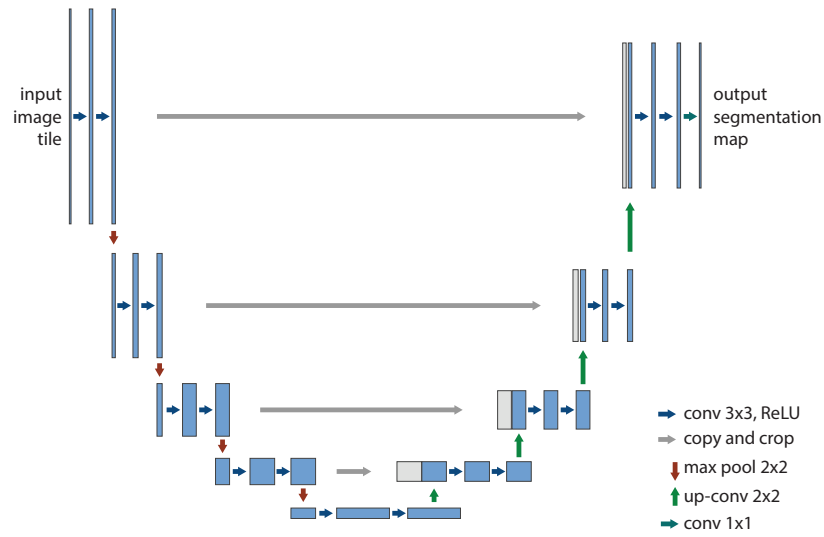


Fig. 12.10 Architecture of the U-Net convolutional neural network (modified from Ronneberger et al., 2015)

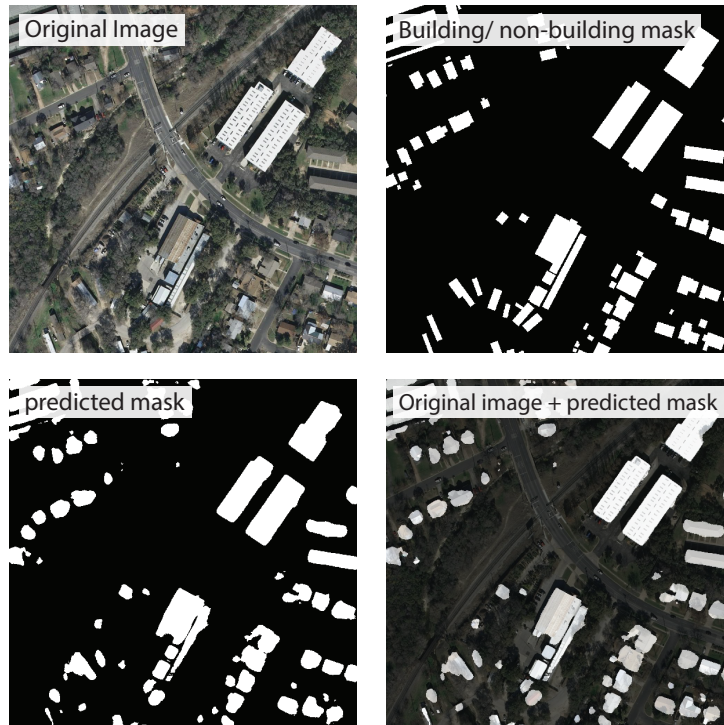


Fig. 12.11 Semantic image segmentation using the U-Net (Ronneberger et al., 2015)

use building/non-building masks to educate the model during the training and as quality control during the validation. The bottom-right panel of Fig. 12.11 reports the predicted mask. Finally, in the bottom-left panel, I over-imposed the predicted mask to the original image to highlight the goodness of the results.

Going in deeper details of the application of semantic image segmentation to Earth Sciences is far behind the scopes of the present book. If interested in improving your knowledge on this topic, I strongly suggest to start looking at the TorchGeo⁴ library (Stewart et al., 2021).

⁴ <https://pytorch.org/blog/geospatial-deep-learning-with-torchgeo/>

Further Readings

Part I: Basic Concepts of Machine Learning in Python for Earth Scientist

- Aitchison, J. (1982). The Statistical Analysis of Compositional Data. *Journal of the Royal Statistical Society. Series B (Methodological)*, 44(2), 139–177.
- Aitchison, J. (1984). The statistical analysis of geochemical compositions. *Math. Geol.*, 16(6), 531–564.
- Aitchison, J., & Egozcue, J. J. (2005). Compositional Data Analysis: Where Are We and Where Should We Be Heading? *Mathematical Geology* 37:7, 37(7), 829–850. <https://doi.org/10.1007/S11004-005-7383-7>
- Bestagini, P., Lipari, V., & Tubaro, S. (2017). A Machine Learning Approach to Facies Classification Using Well Logs. *SEG Technical Program Expanded Abstracts*, 2137–2142. <https://doi.org/10.1190/SEGAM2017-17729805.1>
- Bharath, R., & Reza Bosagh, Z. (2018). *TensorFlow for Deep Learning [Book]*. O'Reilly.
- Bishop, C. (2007). *Pattern recognition and machine learning*. Springer Verlag.
- Boujibar, A., Howell, S., Zhang, S., Hystad, G., Prabhu, A., Liu, N., Stephan, T., Narkar, S., Eleish, A., Morrison, S. M., Hazen, R. M., & Nittler, L. R. (2021). Cluster Analysis of Presolar Silicon Carbide Grains: Evaluation of Their Classification and Astrophysical Implications. *The astrophysical journal. Letters*, 907(2), L39. <https://doi.org/10.3847/2041-8213/ABD102>
- Caricchi, L., Petrelli, M., Bali, E., Sheldrake, T., Pioli, L., & Simpson, G. (2020). A Data Driven Approach to Investigate the Chemical Variability of Clinopyroxenes From the 2014–2015 Holuhraun–Bárdarbunga Eruption (Iceland). *Frontiers in Earth Science*, 8. <https://doi.org/10.3389/feart.2020.00018>
- Chollet, F. (2021). *Deep Learning with Python* (Second Edi). Manning.
- Corlett, W. J., Aitchison, J., & Brown, J. A. C. (1957). The Lognormal Distribution, With Special Reference to Its Uses in Economics. *Applied Statistics*, 6(3), 228. <https://doi.org/10.2307/2985613>
- De Mauro, A., Greco, M., & Grimaldi, M. (2016). A formal definition of Big Data based on its essential features. *Library Review*, 65(3), 122–135. <https://doi.org/10.1108/LR-06-2015-0061/FULL/XML>
- Egozcue, J. J., & Pawlowsky-Glahn, V. (2005). Groups of Parts and Their Balances in Compositional Data Analysis. *Mathematical Geology* 37:7, 37(7), 795–828. <https://doi.org/10.1007/S11004-005-7381-9>
- Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc.
- Hastie, T., Tibshirani, R., & Friedman, J. (2017). *The Elements of Statistical Learning* (Second Edi). Springer.

- Jordan, M., & Mitchell, T. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245), 255–260. <https://doi.org/10.1126/science.aaa8415>
- Lee, W.-M. (2019). *Python Machine Learning*. John Wiley & Sons Inc.
- Limpert, E., Stahel, W. A., & Abbt, M. (2001). Log-normal distributions across the sciences: Keys and clues. [https://doi.org/10.1641/0006-3568\(2001\)051\[0341:LNDATS\]2.0.CO;2](https://doi.org/10.1641/0006-3568(2001)051[0341:LNDATS]2.0.CO;2)
- Lowe, D. J. (2011). Tephrochronology and its application: A review. *Quaternary Geochronology*, 6(2), 107–153. <https://doi.org/10.1016/j.quageo.2010.08.003>
- Maharana, K., Mondal, S., & Nemade, B. (2022). A Review: Data Pre-Processing and Data Augmentation Techniques. *Global Transitions Proceedings*. <https://doi.org/10.1016/J.GLTP.2022.04.020>
- Mitchell, T. M. (1997). *Machine Learning*. McGraw-Hill.
- Morrison, S., Liu, C., Eleish, A., Prabhu, A., Li, C., Ralph, J., Downs, R., Golden, J., Fox, P., Hummer, D., Meyer, M., & Hazen, R. (2017). Network analysis of mineralogical systems. *American Mineralogist*, 102(8), 1588–1596. <https://doi.org/10.2138/am-2017-6104CCBYNCND>
- Murphy, K. P. (2012). *Machine Learning: A Probabilistic Perspective*. The MIT Press.
- Negus, C. (2015). *Linux Bible* (9th Edition, Vol. 112). John Wiley & Sons, Inc.
- Panda, D. K., Lu, X. (o. c. s., & Shankar, D. (2022). *High-performance big data computing*. MIT Press.
- Papa, J. (2021). *PyTorch Pocket Reference*. O'Reilly Media, Inc.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., . . . Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32.
- Pedregosa, F., Varoquaux, G. G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Petrelli, M., Caricchi, L., & Perugini, D. (2020). Machine Learning Thermo-Barometry: Application to Clinopyroxene-Bearing Magmas. *Journal of Geophysical Research: Solid Earth*, 125(9). <https://doi.org/10.1029/2020JB020130>
- Petrelli, M. (2021). *Introduction to Python in Earth Science Data Analysis*. Springer International Publishing. <https://doi.org/10.1007/978-3-030-78055-5>
- Pietsch, W. (2021). *Big Data*. Cambridge University Press. <https://doi.org/10.1017/9781108588676>
- Razum, I., Ilijanić, N., Petrelli, M., Pawlowsky-Glahn, V., Miko, S., Moska, P., & Giaccio, B. (2023). Statistically coherent approach involving log-ratio

- transformation of geochemical data enabled tephra correlations of two late Pleistocene tephra from the eastern Adriatic shelf. *Quaternary Geochronology*, 74, 101416. <https://doi.org/10.1016/J.QUAGEO.2022.101416>
- Samuel, A. L. (1959). Some Studies in Machine Learning Using the Game of Checkers. *IBM J. Res. Dev.*, 3, 210–229.
- Shai, S.-S., & Shai, B.-D. (2014). *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.
- Stephan, T., Bose, M., Boujibar, A., Davis, A. M., Gyngard, F., Hoppe, P., Hynes, K. M., Liu, N., Nittler, L. R., Ogliore, R. C., & Trappitsch, R. (2021). The Presolar Grain Database for silicon carbide—grain type assignments (abstract). *Lunar Planet. Sci*, 52, 2358.
- Tenenbaum, J. B., De Silva, V., & Langford, J. C. (2000). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500), 2319–2323. <https://doi.org/10.1126/SCIENCE.290.5500.2319>
- Trugman, D., & Shearer, P. (2017). GrowClust: A Hierarchical clustering algorithm for relative earthquake relocation, with application to the Spanish Springs and Sheldon, Nevada, earthquake sequences. *Seismological Research Letters*, 88(2), 379–391. <https://doi.org/10.1785/0220160188>
- van den Boogaart, K. G., & Tolosana-Delgado, R. (2013). *Analyzing compositional data with R*. Springer Berlin Heidelberg. <https://doi.org/10.1007/978-3-642-36809-7/COVER>
- Wang, Q., Zhang, F., & Li, X. (2018). Optimal Clustering Framework for Hyperspectral Band Selection. *IEEE Transactions on Geoscience and Remote Sensing*, 56(10), 5910–5922. <https://doi.org/10.1109/TGRS.2018.2828161>
- Ward, B. (2021). *How Linux Works, 3rd Edition: What Every Superuser Should Know*. No Starch Press, Inc.
- Zhang, Z. (2016). Missing data imputation: focusing on single imputation. *Annals of Translational Medicine*, 4(1), 9. <https://doi.org/10.3978/J.ISSN.2305-5839.2015.12.38>
- Zhu, X., & Goldberg, A. B. (2009). *Introduction to Semi-Supervised Learning*. Morgan; Claypool Publishers.

Part II: Unsupervised Learning

- Aitchison, J. (1982). The Statistical Analysis of Compositional Data. *Journal of the Royal Statistical Society. Series B (Methodological)*, 44(2), 139–177.
- Aitchison, J. (1984). The statistical analysis of geochemical compositions. *Math. Geol.*, 16(6), 531–564.
- Aitchison, J., & Egozcue, J. J. (2005). Compositional Data Analysis: Where Are We and Where Should We Be Heading? *Mathematical Geology* 2005 37:7, 37(7), 829–850. <https://doi.org/10.1007/S11004-005-7383-7>
- Andronico, D., & Corsaro, R. A. (2011). Lava fountains during the episodic eruption of South-East Crater (Mt. Etna), 2000: Insights into magma-gas dy-

- namics within the shallow volcano plumbing system. *Bulletin of Volcanology*, 73(9), 1165–1178. <https://doi.org/10.1007/S00445-011-0467-Y/FIGURES/8>
- Arthur, D., & Vassilvitskii, S. (2007). K-Means++: The Advantages of Careful Seeding. *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1027–1035.
- Belkin, M., & Niyogi, P. (2003). Laplacian eigenmaps for dimensionality reduction and data representation. *Neural computation*, 15(6), 1373–1396.
- Blei, D. M., & Jordan, M. I. (2006). Variational inference for Dirichlet process mixtures. *Bayesian Analysis*, 1(1), 121–143. <https://doi.org/10.1214/06-BA104>
- Bonaccorso, A., Carleo, L., Currenti, G., & Sicali, A. (2021). Magma Migration at Shallower Levels and Lava Fountains Sequence as Revealed by Borehole Dilatometers on Etna Volcano. *Frontiers in Earth Science*, 9, 800. <https://doi.org/10.3389/FEART.2021.740505/BIBTEX>
- Boschetti, F. O., Ferguson, D. J., Cortés, J. A., Morgado, E., Ebmeier, S. K., Morgan, D. J., Romero, J. E., & Silva Parejas, C. (2022). Insights Into Magma Storage Beneath a Frequently Erupting Arc Volcano (Villarrica, Chile) From Unsupervised Machine Learning Analysis of Mineral Compositions. *Geochemistry, Geophysics, Geosystems*, 23(4), e2022GC010333. <https://doi.org/10.1029/2022GC010333>
- Branca, S., & Del Carlo, P. (2004). Eruptions of Mt. Etna During the Past 3,200 Years: a Revised Compilation Integrating the Historical and Stratigraphic Records. *Geophysical Monograph Series*, 143, 1–27. <https://doi.org/10.1029/143GM02>
- Cappello, A., Bilotta, G., Neri, M., & Negro, C. D. (2013). Probabilistic modeling of future volcanic eruptions at Mount Etna. *Journal of Geophysical Research: Solid Earth*, 118(5), 1925–1935. <https://doi.org/10.1002/JGRB.50190>
- Caricchi, L., Petrelli, M., Bali, E., Sheldrake, T., Pioli, L., & Simpson, G. (2020). A Data Driven Approach to Investigate the Chemical Variability of Clinopyroxenes From the 2014–2015 Holuhraun–Bárdarbunga Eruption (Iceland). *Frontiers in Earth Science*, 8. <https://doi.org/10.3389/feart.2020.00018>
- Comaniciu, D., & Meer, P. (2002). Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(5), 603–619. <https://doi.org/10.1109/34.1000236>
- Corsaro, R. A., & Miraglia, L. (2022). Near Real-Time Petrologic Monitoring on Volcanic Glass to Infer Magmatic Processes During the February–April 2021 Paroxysms of the South-East Crater, Etna. *Frontiers in Earth Science*, 10, 222. <https://doi.org/10.3389/FEART.2022.828026/BIBTEX>
- Costa, F., Shea, T., & Ubide, T. (2020). Diffusion chronometry and the timescales of magmatic processes. *Nature Reviews Earth and Environment*, 1(4), 201–214. <https://doi.org/10.1038/s43017-020-0038-x>
- Dempster, A. P., Laird, N. M., & Rubin, D. B. (1977). Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical*

- Society: Series B (Methodological)*, 39(1), 1–22. <https://doi.org/10.1111/J.2517-6161.1977.TB01600.X>
- Derpanis, K. G. (2005). Mean shift clustering. *Lecture Notes*, 32.
- Di Renzo, V., Corsaro, R. A., Miraglia, L., Pompilio, M., & Civetta, L. (2019). Long and short-term magma differentiation at Mt. Etna as revealed by Sr-Nd isotopes and geochemical data. *Earth-Science Reviews*, 190, 112–130. <https://doi.org/10.1016/J.EARSCIREV.2018.12.008>
- Donoho, D. L., & Grimes, C. (2003). Hessian eigenmaps: Locally linear embedding techniques for high-dimensional data. *Proceedings of the National Academy of Sciences*, 100(10), 5591–5596.
- Ester, M., Kriegel, H.-P., Sander, J., Xu, X., et al. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *kdd*, 96(34), 226–231.
- Ferlito, C., Coltorti, M., Lanzafame, G., & Giacomoni, P. P. (2014). The volatile flushing triggers eruptions at open conduit volcanoes: Evidence from Mount Etna volcano (Italy). *Lithos*, 184–187, 447–455. <https://doi.org/10.1016/J.LITHOS.2013.10.030>
- Ge, W., Cheng, Q., Jing, L., Wang, F., Zhao, M., & Ding, H. (2020). Assessment of the Capability of Sentinel-2 Imagery for Iron-Bearing Minerals Mapping: A Case Study in the Cuprite Area, Nevada. *Remote Sensing 2020, Vol. 12, Page 3028*, 12(18), 3028. <https://doi.org/10.3390/RS12183028>
- Hastie, T., Tibshirani, R., & Friedman, J. (2017). *The Elements of Statistical Learning* (Second Edi). Springer.
- Higgins, O., Sheldrake, T., & Caricchi, L. (2021). Machine learning thermobarometry and chemometry using amphibole and clinopyroxene: a window into the roots of an arc volcano (Mount Liamuiga, Saint Kitts). *Contributions to Mineralogy and Petrology 2021 177:1*, 177(1), 1–22. <https://doi.org/10.1007/S00410-021-01874-6>
- Immitzer, M., Vuolo, F., Atzberger, C., Sarathi Roy, P., & Thenkabail, P. S. (2016). First Experience with Sentinel-2 Data for Crop and Tree Species Classifications in Central Europe. *Remote Sensing 2016, Vol. 8, Page 166*, 8(3), 166. <https://doi.org/10.3390/RS8030166>
- Johnson, S. C. (1967). Hierarchical clustering schemes. *Psychometrika*, 32(3), 241–254.
- Jolliffe, I. T., & Cadima, J. (2016). Principal component analysis: a review and recent developments. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 374(2065). <https://doi.org/10.1098/RSTA.2015.0202>
- Jolliffe, I. T. (2002). *Principal Component Analysis*. Springer-Verlag. <https://doi.org/10.1007/B98835>
- Jorgenson, C., Higgins, O., Petrelli, M., Bégué, F., & Caricchi, L. (2022). A Machine Learning-Based Approach to Clinopyroxene Thermobarometry: Model Optimization and Distribution for Use in Earth Sciences. *Journal of Geophysical Research: Solid Earth*, 127(4), e2021JB022904. <https://doi.org/10.1029/2021JB022904>

- Lloyd, S. (1982). Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2), 129–137. <https://doi.org/10.1109/TIT.1982.1056489>
- Majidi Nezhad, M., Heydari, A., Pirshayan, E., Groppi, D., & Astiaso Garcia, D. (2021). A novel forecasting model for wind speed assessment using sentinel family satellites images and machine learning method. *Renewable Energy*, 179, 2198–2211. <https://doi.org/10.1016/J.RENENE.2021.08.013>
- Marchese, F., Filizzola, C., Lacava, T., Falconieri, A., Faruolo, M., Genzano, N., Mazzeo, G., Pietrapertosa, C., Pergola, N., Tramutoli, V., & Neri, M. (2021). Mt. Etna Paroxysms of February–April 2021 Monitored and Quantified through a Multi-Platform Satellite Observing System. *Remote Sensing 2021, Vol. 13, Page 3074*, 13(16), 3074. <https://doi.org/10.3390/RS13163074>
- McLachlan, G. J., & Peel, D. (2000). *Finite mixture models*. Wiley.
- Musu, A., Corsaro, R. A., Higgins, O., Jorgenson, C., Petrelli, M., & Caricchi, L. (2022). The magmatic evolution of South-East Crater (Mt. Etna) during the February-March 2021 sequence of lava fountains from a mineral chemistry perspective. *Bulletin of Volcanology*.
- Petrelli, M., Caricchi, L., & Perugini, D. (2020). Machine Learning Thermo-Barometry: Application to Clinopyroxene-Bearing Magmas. *Journal of Geophysical Research: Solid Earth*, 125(9). <https://doi.org/10.1029/2020JB020130>
- Petrelli, M., & Zellmer, G. (2020). *Rates and Timescales of Magma Transfer, Storage, Emplacement, and Eruption*. <https://doi.org/10.1002/9781119521143.ch1>
- Petrelli, M. (2021). *Introduction to Python in Earth Science Data Analysis*. Springer International Publishing. <https://doi.org/10.1007/978-3-030-78055-5>
- Putirka, K. (2008). Thermometers and barometers for volcanic systems. <https://doi.org/10.2138/rmg.2008.69.3>
- Roweis, S. T., & Saul, L. K. (2000). Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500), 2323–2326. <https://doi.org/10.1126/SCIENCE.290.5500.2323>
- Schaepman-Strub, G., Schaepman, M. E., Painter, T. H., Dangel, S., & Martonchik, J. V. (2006). Reflectance quantities in optical remote sensing—definitions and case studies. *Remote Sensing of Environment*, 103(1), 27–42. <https://doi.org/10.1016/J.RSE.2006.03.002>
- Scrucca, L., Fop, M., Murphy, T. B., & Raftery, A. E. (2016). mclust 5: Clustering, Classification and Density Estimation Using Gaussian Finite Mixture Models. *The R Journal*, 8(1), 289–317. <https://doi.org/10.32614/RJ-2016-021>
- Sovdat, B., Kadunc, M., Batič, M., & Milčinski, G. (2019). Natural color representation of Sentinel-2 data. *Remote Sensing of Environment*, 225, 392–402. <https://doi.org/10.1016/J.RSE.2019.01.036>
- Sugiyama, M. (2015). *Introduction to Statistical Machine Learning*. Elsevier Inc. <https://doi.org/10.1016/C2014-0-01992-2>
- Tenenbaum, J. B., De Silva, V., & Langford, J. C. (2000). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500), 2319–2323. <https://doi.org/10.1126/SCIENCE.290.5500.2319>

- Ubide, T., & Kamber, B. (2018). Volcanic crystals as time capsules of eruption history. *Nature Communications*, 9(1). <https://doi.org/10.1038/s41467-017-02274-w>
- Ubide, T., Neave, D., Petrelli, M., & Longpré, M.-A. (2021). Editorial: Crystal Archives of Magmatic Processes. *Frontiers in Earth Science*, 9. <https://doi.org/10.3389/feart.2021.749100>
- Von Luxburg, U. (2007). A tutorial on spectral clustering. *Statistics and computing*, 17(4), 395–416.
- Zheng, N., & Xue, J. (2009). Manifold Learning. In *Statistical learning and pattern analysis for image and video processing* (pp. 87–119). Springer, London. https://doi.org/10.1007/978-1-84882-312-9{_}4

Part III: Supervised Learning

- Aitchison, J. (1982). The Statistical Analysis of Compositional Data. *Journal of the Royal Statistical Society. Series B (Methodological)*, 44(2), 139–177.
- Bentley, J. L. (1975). Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9), 509–517. <https://doi.org/10.1145/361002.361007>
- Bestagini, P., Lipari, V., & Tubaro, S. (2017). A Machine Learning Approach to Facies Classification Using Well Logs. *SEG Technical Program Expanded Abstracts*, 2137–2142. <https://doi.org/10.1190/SEGAM2017-17729805.1>
- Bottou, L. (2012). Stochastic Gradient Descent Tricks. In G. Montavon, G. B. Orr, & K.-R. Müller (Eds.), *Neural networks: Tricks of the trade: Second edition* (pp. 421–436). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-35289-8{_}25
- Breiman, L. (2001). Random Forests. *Machine Learning 2001 45:1*, 45(1), 5–32. <https://doi.org/10.1023/A:1010933404324>
- Breiman, L., Friedman, J. H. (H.), Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Chapman; Hall/CRC.
- Cortes, C., & Vapnik, V. (1995). Support-vector networks. *Machine Learning*, 20(3), 273–297. <https://doi.org/10.1007/BF00994018>
- Costa, F., Shea, T., & Ubide, T. (2020). Diffusion chronometry and the timescales of magmatic processes. *Nature Reviews Earth and Environment*, 1(4), 201–214. <https://doi.org/10.1038/s43017-020-0038-x>
- Friedman, J., Hastie, T., & Tibshirani, R. (2010). Regularization Paths for Generalized Linear Models via Coordinate Descent. *Journal of statistical software*, 33(1), 1. <https://doi.org/10.18637/jss.v033.i01>
- Geurts, P., Ernst, D., & Wehenkel, L. (2006). Extremely randomized trees. *Machine Learning 2006 63:1*, 63(1), 3–42. <https://doi.org/10.1007/S10994-006-6226-1>
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* (Vol. 29). MIT Press.

- Hall, B. (2016). Facies classification using machine learning. *Leading Edge*, 35(10), 906–909. https://doi.org/10.1190/TLE35100906.1/ASSET/IMAGES/LARGE/TLE35100906.1{_}FIG2.JPEG
- Hall, M., & Hall, B. (2017). Distributed collaborative prediction: Results of the machine learning contest. *The Leading Edge*, 36(3), 267–269. <https://doi.org/10.1190/TLE36030267.1>
- Hastie, T., Tibshirani, R., & Friedman, J. (2017). *The Elements of Statistical Learning* (Second Edi). Springer.
- Hernandez-Martinez, E., Perez-Muñoz, T., Velasco-Hernandez, J. X., Altamira-Areyan, A., & Velasquillo-Martinez, L. (2013). Facies Recognition Using Multifractal Hurst Analysis: Applications to Well-Log Data. *Mathematical Geosciences*, 45(4), 471–486. <https://doi.org/10.1007/S11004-013-9445-6/FIGURES/9>
- Hirschmann, M., Ghiorso, M., Davis, F., Gordon, S., Mukherjee, S., Grove, T., Krawczynski, M., Medard, E., & Till, C. (2008). Library of Experimental Phase Relations (LEPR): A database and Web portal for experimental magmatic phase equilibria data. *Geochemistry, Geophysics, Geosystems*, 9(3). <https://doi.org/10.1029/2007GC001894>
- Jorgenson, C., Higgins, O., Petrelli, M., Bégué, F., & Caricchi, L. (2022). A Machine Learning-Based Approach to Clinopyroxene Thermobarometry: Model Optimization and Distribution for Use in Earth Sciences. *Journal of Geophysical Research: Solid Earth*, 127(4), e2021JB022904. <https://doi.org/10.1029/2021JB022904>
- Kubat, M. (2017). *An Introduction to Machine Learning*. Springer International Publishing. <https://doi.org/10.1007/978-3-319-63913-0>
- Lemaître, G., Nogueira, F., & Aridas, C. K. (2017). Imbalanced-learn: A Python Toolbox to Tackle the Curse of Imbalanced Datasets in Machine Learning. *Journal of Machine Learning Research*, 18(17), 1–5.
- Petrelli, M., Caricchi, L., & Perugini, D. (2020). Machine Learning Thermobarometry: Application to Clinopyroxene-Bearing Magmas. *Journal of Geophysical Research: Solid Earth*, 125(9). <https://doi.org/10.1029/2020JB020130>
- Petrelli, M., El Omari, K., Spina, L., Le Guer, Y., La Spina, G., & Perugini, D. (2018). Timescales of water accumulation in magmas and implications for short warning times of explosive eruptions. *Nature Communications*, 9(1). <https://doi.org/10.1038/s41467-018-02987-6>
- Petrelli, M. (2021). *Introduction to Python in Earth Science Data Analysis*. Springer International Publishing. <https://doi.org/10.1007/978-3-030-78055-5>
- Putirka, K. (2008). Thermometers and barometers for volcanic systems. <https://doi.org/10.2138/rmg.2008.69.3>
- Song, Y. Y., & Lu, Y. (2015). Decision tree methods: applications for classification and prediction. *Shanghai Archives of Psychiatry*, 27(2), 130. <https://doi.org/10.11919/J.ISSN.1002-0829.215044>
- Sugiyama, M. (2015). *Introduction to Statistical Machine Learning*. Elsevier Inc. <https://doi.org/10.1016/C2014-0-01992-2>

- Tibshirani, R. (1996). Regression Shrinkage and Selection Via the Lasso. *Journal of the Royal Statistical Society: Series B (Methodological)*, 58(1), 267–288. <https://doi.org/10.1111/J.2517-6161.1996.TB02080.X>
- Tolosana-Delgado, R., Talebi, H., Khodadadzadeh, M., & Boogaart, K. (2019). On machine learning algorithms and compositional data.
- Ubide, T., & Kamber, B. (2018). Volcanic crystals as time capsules of eruption history. *Nature Communications*, 9(1). <https://doi.org/10.1038/s41467-017-02274-w>
- Ubide, T., Neave, D., Petrelli, M., & Longpré, M.-A. (2021). Editorial: Crystal Archives of Magmatic Processes. *Frontiers in Earth Science*, 9. <https://doi.org/10.3389/feart.2021.749100>
- Wood, D. A. (2021). Enhancing lithofacies machine learning predictions with gamma-ray attributes for boreholes with limited diversity of recorded well logs. *Artificial Intelligence in Geosciences*, 2, 148–164. <https://doi.org/10.1016/J.AIIG.2022.02.007>
- Zhang, H. (2004). The Optimality of Naive Bayes. *The Florida AI Research Society*.
- Zou, H., & Hastie, T. (2005). Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2), 301–320. <https://doi.org/10.1111/J.1467-9868.2005.00503.X>
- Zou, Q., Ni, L., Zhang, T., & Wang, Q. (2015). Deep Learning Based Feature Selection for Remote Sensing Scene Classification. *IEEE Geoscience and Remote Sensing Letters*, 12(11), 2321–2325. <https://doi.org/10.1109/LGRS.2015.2475299>

Part IV: Scaling your Machine Learning Models

- Caesar Wu, R. B. (2015). *Cloud Data Centers and Cost Modeling: A Complete Guide To Planning, Designing and Building a Cloud Data Center* (1st ed.). Morgan Kaufmann.
- Daniel, J. C. (2019). *Data Science with Python DASK*. Manning Publications Co.
- Fiore, M., & Campos, M. D. (2013). The algebra of directed acyclic graphs. In *Lecture notes in computer science (including subseries lecture notes in artificial intelligence and lecture notes in bioinformatics)* (pp. 37–51). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-38164-5_{_}4/COVER/
- Maurer, S. B. (2013). *Directed Acyclic Graphs*. Routledge Handbooks Online. <https://doi.org/10.1201/B16132-10>
- Padua David. (2011). *Encyclopedia of Parallel Computing*. Springer US. <https://doi.org/10.1007/978-0-387-09766-4>
- Pedregosa, F., Varoquaux, G. G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É.

- (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
- Peter Pacheco, M. M. (2020). *An Introduction to Parallel Programming* (2nd ed.). Morgan Kaufmann.
- Xu, J. (2003). *Theory and Application of Graphs* (Vol. 10). Springer US. <https://doi.org/10.1007/978-1-4419-8698-6>

Part V: Next Step: Deep Learning

- Aggarwal, C. C. (2018). *Neural Networks and Deep Learning*. Springer International Publishing. <https://doi.org/10.1007/978-3-319-94463-0>
- Badrinarayanan, V., Kendall, A., & Cipolla, R. (2017). SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12), 2481–2495. <https://doi.org/10.1109/TPAMI.2016.2644615>
- Cover, T. M. (1965). Geometrical and Statistical Properties of Systems of Linear Inequalities with Applications in Pattern Recognition. *IEEE Transactions on Electronic Computers*, EC-14(3), 326–334. <https://doi.org/10.1109/PGEC.1965.264137>
- Fischer, A., & Igel, C. (2012). An introduction to restricted Boltzmann machines. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 7441 LNCS, 14–36. https://doi.org/10.1007/978-3-642-33275-3_{_}2/COVER
- Fukushima, K. (1980). Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position. *Biological Cybernetics* 1980 36:4, 36(4), 193–202. <https://doi.org/10.1007/BF00344251>
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning* (Vol. 29). MIT Press.
- Hastie, T., Tibshirani, R., & Friedman, J. (2017). *The Elements of Statistical Learning* (Second Edi). Springer.
- Hinton, G. E., Osindero, S., & Teh, Y. W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18(7), 1527–1554. <https://doi.org/10.1162/NECO.2006.18.7.1527>
- Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8), 1735–1780. <https://doi.org/10.1162/NECO.1997.9.8.1735>
- Imambi, S., Prakash, K. B., & Kanagachidambaresan, G. R. (2021). *Deep learning with PyTorch*. Manning.
- koller, D., & Friedman, N. (2009). *Probabilistic Graphical Models*. MIT Press.
- Kumar, T. A., & Abraham, A. (2022). *Recurrent neural networks : concepts and applications*. CRC Press.
- Long, J., Shelhamer, E., & Darrell, T. (2015). Fully convolutional networks for semantic segmentation. *2015 IEEE Conference on Computer Vision and*

- Pattern Recognition (CVPR)*, 3431–3440. <https://doi.org/10.1109/CVPR.2015.7298965>
- Maggiori, E., Tarabalka, Y., Charpiat, G., & Alliez, P. (2017). Can semantic labeling methods generalize to any city? the inria aerial image labeling benchmark. *International Geoscience and Remote Sensing Symposium (IGARSS)*, 3226–3229. <https://doi.org/10.1109/IGARSS.2017.8127684>
- Milletari, F., Navab, N., & Ahmadi, S. A. (2016). V-Net: Fully convolutional neural networks for volumetric medical image segmentation. *Proceedings - 2016 4th International Conference on 3D Vision, 3DV 2016*, 565–571. <https://doi.org/10.1109/3DV.2016.79>
- Ronneberger, O., Fischer, P., & Brox, T. (2015). U-net: Convolutional networks for biomedical image segmentation. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 9351, 234–241. https://doi.org/10.1007/978-3-319-24574-4_{_}28/COVER
- Stevens, E., Antiga, L., & Viehmann, T. (2020). *Deep learning with PyTorch*. Manning.
- Stewart, A. J., Robinson, C., Corley, I. A., Ortiz, A., Ferres, J. M. L., & Banerjee, A. (2021). TorchGeo: Deep Learning With Geospatial Data. <https://doi.org/10.48550/arxiv.2111.08872>
- Yamashita, R., Nishio, M., Do, R. K. G., & Togashi, K. (2018). Convolutional neural networks: an overview and application in radiology. *Insights into Imaging*, 9(4), 611–629. <https://doi.org/10.1007/S13244-018-0639-9/FIGURES/15>