Machine Learning with Time Series

A Taxonomy of Learning Tasks, Development of a Unified Framework, and Comparative Benchmarking of Algorithms

Markus Löning

A dissertation submitted in partial fulfilment of the requirements for the degree of

Doctor of Philosophy

of

University College London (UCL)

19th September 2021

Declaration

I, Markus Löning, confirm that the work presented in this thesis is my own. Where information has been derived from other sources, I confirm that this has been indicated in the work.

To my brother and father

Abstract

Time series data is ubiquitous in real-world applications. Such data gives rise to distinct but closely related learning tasks (e.g. time series classification, regression or forecasting). In contrast to the more traditional cross-sectional setting, these tasks are often not fully formalized. As a result, different tasks can become conflated under the same name, algorithms are often applied to the wrong task, and performance estimates are are potentially unreliable. In practice, software frameworks such as `scikit-learn` have become essential tools for data science. However, most existing frameworks focus on cross-sectional data. To our knowledge, no comparable frameworks exist for temporal data. Moreover, despite the importance of these framework, their design principles have never been fully understood. Instead, discussions often concentrate on the usage and features, while almost completely ignoring the design.

To address these issues, we develop in this thesis (i) a formal taxonomy of learning tasks, (ii) novel design principles for ML toolboxes and (iii) a new unified framework for ML with time series. The framework has been implemented in an open-source Python package called `sktime`. The design principles are derived from existing state-of-the-art toolboxes and classical software design practices, using a domain-driven approach and a novel scientific type system. We show that these principles cannot just explain key aspects of existing frameworks, but also guide the development of new ones like `sktime`. Finally, we use `sktime` to reproduce and extend the M4 competition, one of the major comparative benchmarking studies for forecasting. Reproducing the competition allows us to verify the published results and illustrate `sktime`'s effectiveness. Extending the competition enables

us to explore the potential of previously unstudied ML models. We find that, on a subset of the M4 data, simple ML models implemented in `sktime` can match the state-of-the-art performance of the hand-crafted M4 winner models.

Impact statement

Time series data is ubiquitous in scientific, commercial and industrial applications. The analysis of such data is extremely important. It guides decision making in applications and ultimately can improve outcomes in the real world.

The main output of the research presented in this thesis is `sktime`, a unified software framework for time series analysis. `sktime` is available as an open-source Python library on GitHub.[1] The impact of `sktime` has already been demonstrated. Since its first release in 2019, `sktime` has become one of the most popular libraries for time series analysis. It has grown from the initial academic collaboration into a community-driven project with contributors from around the world. At the time of writing, `sktime` has more than 90 contributors, 550 forks, 150 dependent projects, and 4K stars on GitHub. `sktime` is being used in methodological and applied research as well as commercial and industrial applications. But the impact of this research goes beyond `sktime`.

The development of `sktime` has both required and facilitated broader research. Developing `sktime` would not have been possible without prior work on ML methodology and software design. We developed clear definitions of different learning problems in the time series domain and novel principles for designing ML toolboxes more generally. With this, we have laid the groundwork for future research in time series methodology and ML software design.

`sktime` has also facilitated other research, primarily through its principled and modular user interface which offers key benefits for ML research and applications. For practitioners, `sktime` enables rapid prototyping and experimentation based

[1]`https://github.com/alan-turing-institute/sktime`

on pre-defined templates for common modeling and validation workflows, allowing practitioners to easily evaluate and compare algorithms in a systematic and reproducible way. In addition, `sktime` makes research code more readable, maintainable and transparent, enabling practitioners to easily verify the scientific validity of implemented workflows. For developers, `sktime` allows to implement new methods as part of an existing open-source framework using available templates, thereby ensuring that new methods are easily accessible and interoperable with existing functionality. Various research publications on algorithm development and benchmarking have already made use of `sktime`. Moreover, `sktime` facilitates the exchange of knowledge between developers and practitioners who work with time series data by providing a platform for collaborative, open-source research on time series analysis.

The research presented in this thesis has also had a direct impact on researchers, practitioners and students through outreach activities. These included software development sprints at The Alan Turing Institute in 2019 and the PyData Global conference in 2020, an online tutorial at the PyData Amsterdam conference in 2020, which has been viewed over 60K times,[2] as well as mentorship programs such as Major League Hacking,[3] Outreachy,[4] Google Summer of Code,[5] Nuffield Future Researchers,[6] Open Life Science[7] and `sktime`'s own mentorship program.[8]

[2]`https://www.youtube.com/watch?v=Wf2naBHRo8Q`

[3]`https://fellowship.mlh.io`

[4]`https://outreachy.org`

[5]`https://summerofcode.withgoogle.com`

[6]`https://www.nuffieldfoundation.org/students-teachers/nuffield-research-placements`

[7]`https://openlifesci.org`

[8]`https://www.sktime.org/en/latest/mentoring.html`

Acknowledgments

My foremost gratitude goes to my supervisor, Dr Franz Király, for his great advice and support throughout my PhD. I am particularly grateful to Franz for his close and insightful supervision, his guidance on a vast number of topics, and personal integrity. I thank him dearly for everything he taught me during the PhD.

I would also like to thank my other supervisors, Prof Paul Longley and Prof James Cheshire, for their guidance and support during my PhD journey. Special thanks go to Paul for his openness and trust in allowing me to pursue my chosen research topic and helping me navigate around several obstacles throughout my PhD.

Research is a team effort and I was fortunate to work with many collaborators. I am deeply grateful to Prof Anthony Bagnall who has been a close collaborator for more than two years now. I would like to express my gratitude to Tony for all the support, advice and opportunities he has given me. I also thank Anthony Blaom, Ahmed Guecioueur, Viktor Kazakov, Istvan Papp, Raphael Sonabend for sharing lessons from their software projects, including `mlaut` [140], `MLJ` [31], `mlr3proba` [237], `pysf` [98] and `skpref` [231], as well as all my colleagues who have helped me in my research at The Alan Turing Institute, the Departments of Statistics, Computer Science and Geography at the University College London (UCL) and the Department of Computer Science at the University of East Anglia (UEA), with special thanks to Paolo Barucca, Balamurgan Soundararaj, Terje Trasberg and Mariflor Vega Carrasco.

I would also like to thank all the people that took the time to read this thesis and gave me feedback, including David Löning, Raphael Sonabend, Simon Weiß, Harold Stone and Karoline Pelikan.

I am deeply grateful to my mother, my brother and friends who have supported me through my academic journey. A special gratitude goes to Karoline whose unwavering optimism and understanding got me through the most challenging times of my PhD.

Finally, I want to thank all contributors who helped build `sktime` and the tools that my research and `sktime` itself has relied upon, with special thanks to Aaron Bostrom, Michal Chromcak, Aadesh Deshmukh, Sajaysurya Ganesh, Viktor Kazakov, Ryan Kuhns, James Large, Jason Lines, Matthew Middlehurst, George Oastler, Omar Onaha, Satya Pattnaik, Patrick Rockenschaub, Patrick Schäfer, Ayushmaan Seth, Martina Vilas, Sebastian Vollmer, Martin Walter, Kirstie Whitaker and Hongyi Yang.

This research was funded by the UK Economic and Social Research Council (ESRC) (ESRC grant: ES-P000592-1), the Consumer Data Research Centre (CDRC) (ESRC grant: ES-L011840-1), my industry partner and The Alan Turing Institute (EPSRC grant: EP-N510129-1). The work on `sktime` was also partly funded by a grant from the 2020 JROST Rapid Response Fund, a community effort of Invest in Open Infrastructure and Code for Science and Society.

Contributions

Publications

Journal articles

- Franz J. Király et al. 'Designing Machine Learning Toolboxes: Concepts, Principles and Patterns'. In: *arXiv preprint* (2021)
- Markus Löning and Franz J. Király. 'Forecasting with sktime: Designing sktime's New Forecasting API and Applying It to Replicate and Extend the M4 Study'. In: *arXiv preprint* (2020)
- Markus Löning et al. 'sktime: A Unified Interface for Machine Learning with Time Series'. In: *Workshop on Systems for ML at NeurIPS 2019* (2019)

Technical reports

- *Tackling Hidden Hunger through Soils.* The Alan Turing Institute & Rothamsted Research. 2020. URL: https://doi.org/10.5281/zenodo.3775489
- *Augmenting Clinical Decision-Making in Intensive Care.* The Alan Turing Institute & Great Ormond Street Hospital. 2020. URL: https://doi.org/10.5281/zenodo.3670726
- *Machine Learning for Enhanced Understanding of Cell Culture Bioprocess Development.* The Alan Turing Institute & AstraZeneca. 2019. URL:

`https://doi.org/10.5281/zenodo.3367412`

- Anthony Bagnall et al. ‘A tale of two toolkits, report the first: benchmarking time series classification algorithms for correctness and efficiency’. In: *arXiv preprint arXiv:1909.05738* (2019)

Software

- Markus Löning et al. *alan-turing-institute/sktime*. Oct. 2020. URL: `https://doi.org/10.5281/zenodo.3749000`

Contents

Chapter 1

Introduction

1.1 Time series analysis and machine learning

Time series data is ubiquitous in scientific, commercial and industrial applications. A time series consists of an indexed sequence of values, typically from observing some phenomenon under study repeatedly over time. Time series appear in many applications. Examples include price movements on financial markets, sensor readings in industrial processes (e.g. the temperature or pressure in a chemical reactor), patients' medical records (e.g. the blood pressure and heart rate), and customers' shopping histories.

An intrinsic characteristic of time series is that, typically, observations are statistically dependent on previous observations. Intuitively, having observed certain values in the past makes it more probable to observe certain values in the future. Time series analysis is a set of techniques that is concerned with the analysis of this dependence. Analyzing time series is extremely important in real-world applications. It enables us to better understand the underlying processes that give rise to the observed data and to make predictions about them. Ultimately, time series analysis can guide our decision making and improve outcomes in real-world applications.

Machine learning offers a closely related set of techniques for data analysis and prediction.[1] Although machine learning has traditionally focused on non-temporal, cross-sectional data, in recent years many techniques have been developed for making predictions for time series data. We will refer to these techniques throughout the thesis as "machine learning", or "ML" for short, and when applied to time series as "ML with time series".

ML with time series is a highly interdisciplinary field. Techniques are used and developed in various, often overlapping, disciplines, including econometrics, finance, medicine, engineering and the natural and social sciences, among others. Because of this, similar problems and techniques are often presented in different contexts. In contrast to the more traditional, cross-sectional ML setting, there appear to be no established "consensus" definitions of common time series problems, what we call "learning tasks". At the same time, there are various closely related but distinct tasks that can emerge in a temporal data context. For example, forecasting, one of the most common tasks, refers, in general, to the problem of making a temporal forward prediction based on past data. But, depending on context, forecasting may take on different meanings: it may involve a single or multiple time series, series may be related or independent, and data may or may not be available for the future time periods we want to predict. These are subtle but important differences for real-world applications. While forecasting alone gives rise to ambiguous problems formulations, many more learning tasks can emerge in a temporal data context, including time series classification, time series regression and annotation, among others. As a consequence, it is not always obvious how different problems relate to each other or how solutions for one problem can be applied to another. Understanding the different forms such problems may take is crucial for understanding how we can use ML techniques to solve them. What are common forms of data that arise in temporal settings? What are common kinds of learning problems? How are they related? And how can we define these problems in a mathematically precise manner?

The first goal of this thesis is to address these questions. We do this by

[1]For an introduction to machine learning, see e.g. James et al. [132].

developing a formal description and taxonomy of time series data and key learning learning tasks. The taxonomy concentrates on predictive time series tasks, specifically (deterministic) point-prediction tasks, including forecasting as well as time series classification and regression, leaving non-predictive tasks (e.g. time series clustering) and probabilistic tasks (e.g. distributional forecasting) for future work. Note that throughout the thesis, we focus on learning associations rather than identifying causal relationships.

1.2 A unified framework for time series analysis

A second goal of this thesis is to design a unified software framework for multiple time series learning tasks. In practice, ML applications typically involve a number of steps: practitioners first specify, train and select an appropriate model and then validate and deploy it. To evaluate such workflows, practitioners write software code, often incorporating functionality from existing software packages. These software packages, called “toolboxes”, provide prefabricated pieces of code that make it faster to write application code. Instead of constructing every piece of software from scratch, practitioners can simply put together prefabricated pieces of code. A “framework”, on the other hand, is a special kind of toolbox. Frameworks not only offer reusable functionality, but also provide overall structure. They capture common software design decisions within a given application domain and distill them into templates that practitioners only need to copy and fill in. This reduces the number of decisions practitioners must take and allows them to focus on application specifics. Not only can practitioners write software faster as a result, but applications will have a similar structure. They will be more consistent, more reusable and easier to maintain.

In recent years, frameworks have become essential infrastructure of modern data science. They largely determine what is possible in practice. They have become the principal tool for practitioners and central components in scientific, commercial and industrial applications. Popular examples include `scikit-learn`

[205] in Python, `Weka` [101] in Java, `MLJ` [31] in Julia, and `mlr3` [151] or `caret` [148] in R. However, many existing frameworks, including all of the above, focus on cross-sectional data. Cross-sectional data consists of observations on multiple independent instances from different kinds of measurements at a single point in time (e.g. the medical diagnosis for different patients at hospital admission). The fundamental assumption of cross-sectional data, that observations represent independent samples, is typically violated by time series data. This is why cross-sectional frameworks tend to consider time series out of scope (see e.g. Buitinck et al. [46]). Note that while one can still apply cross-sectional techniques to time series, this usually adds considerable complications to standard data science workflows and requires extra care to avoid errors. Despite the ubiquity of time series data, to our knowledge, no framework comparable to the cross-sectional ones exists for ML with time series. The second goal of this thesis is to develop such a framework.

As we have seen, time series data can give rise to many learning tasks, including forecasting, classification and annotation. These tasks describe distinct learning problems, but they are also closely related. This relation can be understood in terms of "reduction" [24]. Reduction is a technique for leveraging an algorithm for one task to solve another task. As we will see, reduction is central to time series analysis. Many reduction approaches are possible in the time series domain and many existing state-of-the-art solutions make use of reduction.

Many popular reduction approaches reframe a time series task into a related cross-sectional task, so that one can use any of the more established cross-sectional algorithms to solve the original time series task. For example, a forecasting task can be solved via cross-sectional regression with the help of a prior sliding-window transformation of the data [34]. Similarly, a time series classification task can be reframed into a cross-sectional classification task by first extracting features from each time series [87].

However, despite the importance of reduction, there is no unified framework for multiple learning tasks which would enable practitioners to easily apply algorithms for one task to another. Much to the contrary, the current software ecosystem for time series analysis is relatively fragmented. While there are various specialized toolboxes that provide rich interfaces to particular model families or learning tasks, most toolboxes are incompatible with each other and lack integration with more foundational cross-sectional frameworks. For practitioners, this makes it hard to combine functionality from different toolboxes and to fully leverage reduction relations. For developers, this makes it hard to build and integrate new methods without having to re-implement substantial parts of functionality that already exists elsewhere. Overall, toolbox capabilities for time series analysis therefore remain limited. To address these issues, we propose to develop a *unified* framework, supporting multiple learning tasks and reduction approaches between them.

The proposed framework has been implemented in `sktime`, a free and open-source software package.[2] To the best of our knowledge, `sktime` is the first unified framework for ML with time series. The aim of `sktime` is to build a well-established framework that makes the current ecosystem more usable and interoperable as a whole. While our target audience is capable of basic programming, the ambition is to provide a practical and consistent ML framework in Python for the purpose of specifying, training and validating time series algorithms within a programming environment that is accessible to non-ML experts and reusable in various scientific applications. The focus is, therefore, on providing a modular and principled object-oriented application programming interface (API). We make use of the enhanced, interactive Python interpreter [206], specifically designed for scientific computing, rather than expending efforts into creating a command-line interface, let alone a graphical user interface (GUI). We concentrate on medium-sized data that fits into the memory of a single machine. For larger data sets, we intend to integrate existing tools for more scalable, distributed computing across multiple machines (e.g. `Dask` [219]).

The current version of `sktime` focuses on common predictive tasks such as time series classification, regression and forecasting. The design, however, is

[2]`sktime` is hosted on GitHub at `https://github.com/alan-turing-institute/sktime`.

easily extensible to other tasks. In future work, we want to add support for non-predictive tasks, such as time series clustering and annotation.

1.3 Software design for ML toolboxes

It will be helpful to distinguish, from the above, two fundamental problems in ML applications and research. We call them the "practitioner's problem" and the "developer's problem". The practitioner's problem is to solve a particular ML problem at hand. For example, forecasting the temperature of a chemical process or predicting the type of disease from a patient's heart rate. To solve these problems, practitioners write application code. The developer's problem, on the other hand, is to develop toolboxes that help practitioners solve their respective problem more effectively. Much of existing ML research has focused on algorithm development and finding better solutions to specific practitioner's problems. Much of this thesis, by contrast, is concerned with finding better solutions to the developer's problem.

The effectiveness and applicability of a toolbox crucially depends on its design. Toolbox design – much like any software design – is hard. One has to identify key objects in practitioners' workflows, find abstractions for them at the right level of granularity, translate them into classes and functions with well-defined interfaces, specify clear hierarchies and relations among them, and implement them in a reusable software package. For example, the key objects in an ML context are learning algorithms and data containers. Toolbox design is then about finding abstractions for these objects, so that practitioners can reuse algorithms on different data sets without having to change much code.

While the importance of the developer's problem has been recognized (see e.g. Sonnenburg et al. [238]), there is still only little research to date that addresses its challenges. In particular, few papers exist that study the principles of toolbox design. Instead, discussion often concentrates on communicating the "what" of designs (e.g. software features or usage) while almost completely ignoring the "why". For example, toolbox developers often subscribe to a set of design principles when presenting their work, however these principles typically remain too vague to explain concrete design decisions (see e.g. Buitinck et al. [46]). While the actual software tends to contain a wealth of design ideas, we are not aware of any literature in which generalizable design principles for ML frameworks are described. To address these issues, a third goal of this thesis is therefore to derive key software design principles for ML toolboxes.

We believe that analyzing the "why" is crucial for generalizing successful designs from one domain to new ones (e.g. from the domain of cross-sectional data to time series). Our principles, as we will see, cannot just explain key aspects of existing toolboxes, but also guide the development of new frameworks – including `sktime`, our proposed framework for ML with time series. Ultimately, we hope that our research will inspire other research on the underlying design principles of ML toolboxes.

Our approach to software design largely falls under "domain-driven design" [78]. The central idea of domain-driven design is that the structure and language of software should closely correspond to key concepts in the domain of interest. Every software relates to some activity or interest of its intended users. That subject area to which the user applies the software is the domain. The first step in toolbox design is therefore to develop a conceptual model of our domain of interest, namely ML theory or methodology and specifically for ML with time series.

The second step is to map the conceptual model onto software. The corpus of classical software design offers a natural source of relevant ideas, useful formalism and best practices for translating concepts into software. While much is directly transferable from existing software design practice, there is a substantial aspect in which ML departs from classical domains: algorithms, interfaces and workflows are closely intertwined with mathematical and statistical theory – to an extent that mathematical objects are not just at the methodological core, but also a key element in their representation, workflow specification and user interaction. We believe that progress in toolbox design necessitates progress in ML theory, both in terms of conceptual analysis and mathematical formalism. Yet, ML theory remains largely absent from scientific discussions on software design. This situation poses unique challenges: How can one identify, describe and motivate design solutions for ML toolboxes? How can we find incisive abstractions in the domain of ML? How can we formalize these abstractions in a way that is linked to the underlying mathematical concepts and at the same time implementable in software? What generalizable design principles can be derived from existing toolboxes to guide the design of new ones? While these questions address ML toolbox design more generally, there are also questions specific to the time series domain that need to be answered: What are the different types of algorithms in the domain of ML with time series? What should the interface for these algorithms look like? And how do different algorithms interact and relate to each other?

We attempt to address these questions by a combination of conceptual modeling, formal mathematical statistics, novel design principles and adapted state-of-art design patterns. We will start by reviewing key concepts in software design, with a focus on object-oriented programming, the predominant paradigm for ML software. We then propose a simple but powerful idea called "scientific types" – a new type system which captures the data scientific purpose of key ML concepts. In brief, a scientific type is a structured data type together with key mathematical or statistical properties that all elements of that type must satisfy. Scientific types will enable us to describe key concepts in a way that is both mathematically precise and easily implementable in software. For example, we might say that a "pipeline" consists of a "feature extractor" and a "supervised learner", put together in a certain form. We believe that the types in question can be made precise so that they can be leveraged for design considerations, rather than just being vague metaphors without tangible content. With scientific types, we are able to derive a novel set of ML-specific software design principles. As we will see, these principles cannot only explain central aspects of existing toolboxes, but also guide the development of new ones like `sktime`.

1.4 Comparative benchmarking of algorithms

Having developed a unified framework for ML with time series, the fourth and last goal of this thesis is to illustrate its effectiveness by reproducing and extending one of the major comparative benchmarking studies for forecasting, called the M4 competition [182].

Comparative benchmarking studies are essential for ML research, as they allow us to systematically evaluate new algorithms and compare them against existing baselines and state-of-the-art solutions. The reproducibility of these studies is therefore crucial for scientific progress [40, 128, 178]. Toolboxes, like `sktime`, with a principled and modular interface, enable us to easily replicate results from existing algorithms and experiment with new ones.

In particular, reproducing the M4 competition will allow us to validate the published results, test our framework against reference implementations, and illustrate its effectiveness and applicability. Extending the M4 competition will allow us to implement and evaluate previously unstudied algorithms and investigate whether simple, reduction-based ML algorithms can match the state-of-the-art performance of the custom-built algorithms that won the M4 competition. To this end, we use `sktime` to re-implement key algorithms included in the competition and add reduction-based models that are easily implementable in `sktime`. To our knowledge, this is the first complete reproduction of the M4 competition that is independent of the published code. As we will see, we are able to not just re-implement the competition within the single framework provided by `sktime`, but also to build simple ML models that perform on par with the winning models on important subsets of the M4 data set.

1.5 Research questions

In summary, the research questions addressed in this thesis can be grouped into three sets of questions, according to their conceptual, methodological and applied nature.

The first set of questions is of conceptual nature: What are the different kinds of data forms and learning problems that arise in a time series data context? How can we formalize these problems in a mathematically precise way? How are they related? These questions are addressed in part I of this thesis by developing a formal taxonomy of learning tasks.

The second set of questions is of methodological nature: How can we identify, describe and motivate design solutions for ML toolboxes? How can we find incisive abstractions in the domain of ML? How can we formalize these abstractions in a way that is linked to the underlying mathematical concepts and at the same time easily implementable in software? What generalizable and reusable design principles and patterns can be derived from existing toolboxes to guide the design of new ones? These questions are addressed in part II. Answers to these questions largely require novel research on ML toolbox design. We hope to provide some answers in this thesis by deriving key design principles from existing state-of-the-art toolboxes and best practices from classical software design. While these questions concern toolbox design more generally, this thesis also addresses the specific design of a unified framework for ML with time series: What are the different types of algorithms in this domain? What should the interface for these algorithm types look like? How do different algorithm types interact and relate to each other? In addition, we discuss questions relating to the rationale for creating a new unified framework for ML with time series: What related software does already exist? What are the limitations of the existing software ecosystem? What are the reasons for developing a unified framework for ML with time series? While several toolboxes exist for ML with time series, to our knowledge, we are the first to provide a unified framework supported by our taxonomy of learning tasks and design principles.

The last set of questions is of applied nature: How can we use the unified framework to specify new ML algorithms? Given a unified framework for ML with time series, can we find simple ML algorithms that match the performance of state-of-the-art algorithms for forecasting? These questions are addressed in part III by reproducing and extending the M4 competition, one of the key comparative benchmarking studies in forecasting research.

1.6 Contributions

The research contributions of this thesis can be summarized as follows:

1. *Formalization of time series learning problems* as learning tasks and development of a formal *taxonomy for time series learning tasks*, with a focus on common (deterministic) point-predictive tasks such as time series classification, regression and forecasting, as well as the reduction relations between them.

2. *Development, formalization and motivation of a novel set of general, reusable design principles for ML toolboxes* based on the idea of a *scientific type system* that links the software implementation with the underlying mathematical and statistical concepts, inspired by existing state-of-the-art toolboxes and best practices from classical software design.

3. *Review of existing time series analysis software* and a discussion of the limitations of current toolbox capabilities, with a focus on the open-source ecosystem in Python.

4. *Design and implementation of the first unified framework for ML with time series*, with the aim to provide a principled and modular object-oriented application programming interface (API) in Python for specifying, training and validating time series algorithms for medium-sized data. The unified framework has been implemented in an open-source project called `sktime`.[3]

5. *Reproduction and extension of the M4 forecasting competition*, one of the major comparative benchmarking studies for predictive performance of

[3] `https://github.com/alan-turing-institute/sktime`

forecasting algorithms, with a focus on using `sktime` to evaluate and compare simple reduction-based ML algorithms. To our knowledge, this is the first complete reproduction of the M4 competition within a single framework independent of the published code.

1.7 Structure

This thesis has a three-part structure, with a conceptual, a methodological and an applied part.

In the first part, we develop the conceptual model for the domain of ML with time series, including the taxonomy of time series learning task. We start in Chapter 2 by reviewing key ML concepts and the traditional cross-sectional supervised learning setting which will serve us as a reference for comparison throughout the thesis. The formal taxonomy of time series learning tasks is then developed in Chapter 3.

The second part is methodological. We first derive general software design principles for ML toolboxes, and then use them to develop the specific design of the unified framework for ML with time series implemented in `sktime`. We begin in Chapter 4 by introducing fundamental concepts from software design in the context of ML toolbox design. Chapter 5 introduces the idea of scientific types and derives generalizable design principles. Chapter 7 reviews the open-source Python time series ecosystem and discusses the current limitations and rationale for a unified framework for time series analysis. Chapter 8 motivates and describes the design and implementation of `sktime`, combining the conceptual model from part I and software design principles from part II.

The third and final part is applied. In Chapter 9, we use `sktime` to evaluate and compare simple ML algorithms for forecasting by reproducing and extending the M4 forecasting competition, validating published results, testing the correctness of the implemented functionality against reference implementations and illustrating `sktime`'s effectiveness and applicability.

Chapter 10 concludes by discussing limitations of this thesis and directions for future research.

Part I

A Taxonomy of Learning Tasks

Chapter 2

Background: Machine Learning

2.1 Introduction

The questions addressed in this thesis are at the intersection of time series analysis, ML and software design. This part of the thesis focuses on time series analysis and ML. The next part focuses on software design.

We begin by developing a conceptual model of the time series domain. In line with domain-driven design [78], the conceptual model is a crucial first step for developing the proposed unified framework for ML with time series. In this chapter, we review key concepts from ML and time series analysis. In the next chapter, we introduce the taxonomy of learning tasks.

Throughout this thesis, we will refer to both "machine learning", or "ML" for short, and "time series analysis", or jointly "ML with time series" as a set of techniques, or a methodology, that is being used and developed in various fields. ML is highly interdisciplinary. Similarly, time series analysis draws upon ideas from multiple related and often overlapping fields, including statistics [36, 41, 59, 93, 105, 132, 161], artificial intelligence [222], data science [75], data mining [77, 79, 83, 103], econometrics [16, 97, 175, 208, 267], finance [51, 66] as well as signal processing and engineering [239], among others.

While ML techniques for time series are being used and developed in all of these fields, the way they are typically presented differs from field to field. As a result, it is not always obvious how techniques from different fields relate to each other or could be applied to applications in other fields. The aim of the taxonomy of learning tasks developed in the next chapter is to clearly describe these tasks and clarify their relationship.

The rest of this chapter is organized as follows: We start in Section 2.2 by giving a motivating example for ML with time series. Based on the example, we will describe the concepts a learning task in Section 2.3. In Section 2.5, we illustrate how learning tasks help us formalize learning problems by reviewing the the traditional supervised learning tasks, an example that should be familiar to most readers. This example will serve us as a reference for comparison when we introduce the taxonomy of time series tasks in the next chapter.

2.2 A motivating example: Chemical process engineering

To motivate our study of ML with time series, we begin with an example inspired by Löning et al. [168]. The purpose of the example is to provide an intuitive introduction to the topic. We will later formalize some of the key concepts.

Suppose that we are data scientists hired by a pharmaceutical company. Our job is to provide advice on how to improve a chemical production process to increase the quality of the drugs produced. The production process is carried out in a chemical reactor, i.e. an enclosed vessel in which chemical reactions take place in a controlled environment. This allows us to both control the process by adjusting certain parameters and to collect data about the process.

Suppose we run the same kind of process multiple times. For each run, we collect repeated measurements, say at one-minute intervals, of two key variables, say temperature and pressure inside the reactor. Suppose the total duration of the process is one hour, so we observe a total of $T = 60$ time points. Combining data from all runs gives us what we call “multivariate panel data”, consisting of observations on multiple independent runs for different kinds of measurements at multiple time points. At the end of each process, we additionally measure the quality of the drug produced. Suppose that product quality is simply recorded as a binary variable indicating "high" or "low" quality.

In order to improve drug quality, we need to simplify our problem into more tractable tasks which can be solved by ML algorithms.[1]

Time-series classification

For example, one of these tasks may be to make an accurate prediction of the drug quality from a partially observed process, say a process observed only up to 30 minutes. In its general form, this task is called "time series classification". Predictions about the process outcome from a partially observed process may be crucial in practice since they can guide our decision making and save costs. For example, it can signal adjustments to process control parameters or early termination of a process that is predicted to yield a low-quality drugs.

In this setting, the time series data for temperature and pressure are input variables, while drug quality is the output variable. Denote the input variables by Z, with a subscript to distinguish the variable and time point. So Z_{1t} might be temperature at some time point $t \leq T$ and Z_{2t} might be pressure. The output variable is usually denoted by Y. We assume that there is some relationship between Y and Z which can be written in the general form:

$$Y \approx f(Z). \tag{2.1}$$

Here f is some fixed but unknown function of Z. In this formulation, f represents the systematic information that Z provides about Y. The task is then to find an estimate of the unknown function f so that we can predict Y as $\hat{Y} = \hat{f}(Z)$, where

[1]Note that an adequate answer to the question of how to improve the chemical process control requires causal modeling which is beyond the scope of this thesis. Throughout the thesis, we focus on learning associations rather than causal relationships. For an introduction to causal inference, see e.g. Pearl, Glymour and Jewell [204].

$\hat{f}$ denotes the estimated function and $\hat{Y}$ the predicted outcome.

To estimate $\hat{f}$, we typically use a learning algorithm. In this setting, the learning algorithm takes in some training data returns a fitted prediction function $\hat{f}$. In our case, the training data consists of a subset of the runs. For each run, we have a pair of input-output data. Various time series classification algorithms have been proposed in the ML literature (e.g. Time Series Forest [71] or ROCKET [68]). For an overview, see Bagnall et al. [14]. We will later return to time series classification.

Forecasting

Another task that may help us improve drug quality is to make accurate temporal forward predictions of one of the process variables itself based on a partially observed process. In its general form, this task is called "forecasting". Being able to make forecasts is important in practice, as it enables us to anticipate future changes to process and can indicate process adjustments.

In this setting, the time series variable that we want to predict, say temperature, is called the "target" variable and denoted by Z. We assume there is some relationship over time which can be written in the general form:

$$Z_t \approx f(t). \tag{2.2}$$

Here, f is some fixed but unknown function of time t that we want to estimate so that we can predict $\hat{Z}_t = \hat{f}(t)$ at some future time point t.

As before, to estimate $\hat{f}$ we use a learning algorithm that takes in some training data and returns a fitted prediction function $\hat{f}$. In this setting, training data typically consists of past data of a single run observed up to the point in time at which we want to make a prediction. Various algorithms have been put forward in the literature (e.g. exponential smoothing [113, 114, 263] or ARIMA [39]). For an overview, see Makridakis, Spiliotis and Assimakopoulos [182].

So far we have given an introduction to time series classification and forecasting.

We will define these two tasks more formally in Chapter 3. Time series classification and forecasting are two examples of time series learning tasks. Many more other tasks may be encountered in practice. We give a stylized overview of key time series tasks in Table 3.1.

We make a few additional remarks below to further motivate our study of ML with time series and limit the scope of this thesis.

Why estimate f? – predictive and explanatory modeling

There are two fundamental reasons why practitioners may wish to estimate f [229]:

- *Predictive modeling.* In many situations, a set of inputs is readily available, but the output cannot be easily observed. So practitioners find an estimate $\hat{f}$ to generate predictions from the input. In this setting, $\hat{f}$ is often treated as a "black box" in the sense that we are not typically concerned with the exact form of $\hat{f}$ provided that it yields reliable predictions. Predicting the drug quality from input series or forecasting temperature from past observations are two examples of modeling for prediction.

- *Explanatory modeling.* In other situations, practitioners are interested in explaining the relationship between data points. In this setting, one wishes to estimate f but the goal is not necessarily to make predictions, but rather to understand the relationship between the input and output variables. Now $\hat{f}$ cannot be treated as a black box because practitioners want to know its exact form. Trying to understand the relationship between temperature and drug quality in terms of how much a given change in temperature at some point during the process is associated with a change in drug quality is an example of modeling for explanation.

In this thesis, we are primarily concerned with predictive modeling, leaving explanatory modeling (e.g. time series clustering [2]) for future work.

Deterministic and probabilistic predictions

In many situation, it is enough to make a point prediction giving a single value for the data point of interest. In others situation, one wishes to quantify the uncertainty associated with a point prediction or make predictions for an outcome which is inherently uncertain. In these situations, it is not enough to make point predictions. Instead, practitioners need to make probabilistic predictions giving a probability distribution over the values that the output variable could take. In this setting, a point prediction can be understood as an estimate of a central value of the predicted distribution (e.g. the mean). Common probabilistic time series tasks include:

- Survival analysis, modeling the risk of a single event under censoring [60, 139, 146],
- Point process analysis, modeling the risk of multiple recurring events [64],
- Probabilistic variants of the predictive tasks discussed above (e.g. distributional forecasting or probabilistic time series classification).

In this thesis, we focus on deterministic, point-predictive tasks, leaving probabilistic modeling for future work. But note that much of the conceptualization and design ideas presented in this thesis extend to non-predictive and probabilistic modeling. We will now start to further define the concepts of a learning algorithm and a learning task.

2.3 Learning tasks – no solution without a problem

A learning task is a formal, mathematical definition of the learning problem one wants to solve. Common examples of learning tasks include cross-sectional classification and regression, which we will at the end of this chapter. Common examples of time series learning tasks include time series classification and forecasting, as described in the chemical process engineering example above.

We argue that to fully define a learning tasks, one has to specify the following three parts:

- **Data specification.** This includes a formal description of the assumed *data-generative process* that gives rise to the observed data, including the statistical dependency structure between data points, as well as description of the *relational structure* (in database parlance) of the data in some (mathematical) data objects, for example a two-dimensional matrix with descriptions for the meaning of rows and columns.

- **Learning process specification.** The learning process specification answers the question: at what point does the learning algorithm take in what data, and what does it return? It includes a combination of an *algorithmic interface* description specifying the input and output objects of the learning algorithm, and a *statistical model* specifying the relationship and dependency between data and fittable model parameters.

- **Success specification.** This includes *evaluation criteria* for the learning algorithm specifying quantitatively what it means for such an algorithm to be a good solution. This needs to be operationalizable. Whether the algorithm has been successful should be clearly defined and empirically verifiable. In a predictive setting, this involves a loss function which, by comparing predicted values against actually observed values, quantifies the predictive accuracy of the algorithm. Other evaluation criteria are usually taken into account too, such as interpretability [95], fairness [187, 191] or computational efficiency of the algorithm.

The three parts of the learning task are crucial. For a clear problem specification we need to know: given what, what are we doing, and how do we know we have done it well. We will use this three-part specification as a template, both for the traditional supervised learning task below and the taxonomy of time series learning tasks in the next chapter.

We make a few additional remarks:

- A learning task describes a *type* of problem. It describes the general structure and assumptions of a learning problem, without reference to specifics about data sets, algorithms or loss functions. The problems that practitioners might work on in practice can be seen as a particular instance of a given type of task. A instance of task will contain the necessary information about the problem at hand in order to fully specify it. For example, in supervised learning, the task would contain at least a reference to the target variable, so that one knows which variables are features and which one is the target to predict. In forecasting, it would contain at least a reference to the target variable and forecasting horizon, that is, the time points for which one wants to generate predictions.
- To find a solution, a clear task specification is crucial. Without a clear problem description, it is not even clear what solutions could look like. On the other hand, once we arrive at a clear task specification, the type of solution is also defined. As we will see, a task not only describes what we want to solve, but also the type of learning algorithm required to solve it. A task can be understood to be the counterpart of a learning algorithm. The algorithm reflects the type of task that it is meant to solve. More specifically, the algorithm can be understood as the counterpart to the task's learning process specification, which defines the steps that the algorithm is required to perform in order to solve the task. We will make use of this insight later when discussing ML toolbox design in Chapter 5.
- The concept of a learning task is not uncommon in ML methodology, but typically implicit.[2] It is rarely made explicit as a concept on its own, possibly

[2]For example, see the taxonomy of different cross-sectional predictive tasks put forward by Dietterich [72].

since it is not a necessary concept in typical methodological arguments where it appears as part of the framing. Instead, it is often implicit in the choice of algorithm type, scope of the software, or application domain. For example, when we choose a regressor from `scikit-learn` [205], the implied task is that of cross-sectional supervised regression. As a partial inspiration of our conceptual model, the `mlr` toolbox ecosystem [28, 151] and `openML` API [252] has software formalism for (concrete) learning tasks, which is similar to our (generic) tasks on the conceptual level.

- Correctly identifying the learning task from a real-world problem is not always straightforward, especially in a time series settings where several closely related but distinct tasks are possible. Misdiagnosing the learning task, if unnoticed, can often make developed ML models inapplicable to the actual real-world problem, or worse, lead to over-optimistic performance estimates and bad outcomes when the model is deployed.

Before we look at the example tasks of cross-sectional supervised learning, we will introduce some terminology.

2.4 Terminology: Instances, variables and time points

Here and throughout the thesis, we will use the following terminology to describe different forms of data:

- An *instance*, sometimes also called "example" or "sample", refers to an experimental unit, that is, a member of the set of entities being studied about which an ML practitioner wishes to generalize (e.g. patients or chemical process runs) [61, 193]. The crucial statistical assumption is that instances represent identically distributed and independent (i.i.d.) realizations of the same assumed generative process. Throughout the thesis, I will use "instance" rather than "sample" to disambiguate the term from the individual "samples", or observations, of a time series.

- A *variable* refers to a *kind of measurement.* Variables may be cross-sectional (sometimes also called "static" or "offline"), representing a time-invariant measurement (e.g. a patient's place of birth or the drug quality measured at the end of a chemical process) or time series (sometimes called "dynamic" or "online"), representing a time-varying measurement (e.g. a patient's blood pressure over time or repeated measurements of the temperature in a chemical process).
- A *time point* refers to a point in time at which we make an observation. A time point may represent an exact point in time (a timestamp), a time period (e.g. minutes, hours or days), or simply an index indicating the position of an observation in the sequence of values. We discuss time series data in more detail in Chapter 3.

2.5 Example: Cross-sectional supervised learning

In this section, we review the traditional cross-sectional classification and regression tasks, jointly known as "supervised learning". These tasks should be familiar to most readers and will serve us as a reference for comparison when developing the taxonomy of time series learning tasks in the next chapter.

In supervised learning, we are interested in predicting the value of a target variable based on patterns between feature data and the target variable. Our exposition below follows the "consensus" formulation as, for example, found in James et al. [132].[3]

[3]Similar formulations can be found in Hastie, Tibshirani and Friedman [105] and Shalev-Shwartz and Ben-David [228] among others. For a more fundamental approach to statistical learning theory, see e.g. Vapnik [253–255].

Data specification

We assume cross-sectional data. Cross-sectional data consists of observations on *multiple independent instances* for different *variables* at a *single time point* – i.e. a cross-section in time. Formally, we can define the assumed data generative process as follows. Let

$$(X^{(1)}, Y^{(1)}) \dots (X^{(N)}, Y^{(N)}) \overset{i.i.d.}{\sim} (X, Y) \tag{2.3}$$

be random variables which are jointly i.i.d., with X denoting feature variables, Y the target variable, and $N \in \mathbb{N}^+$ the number of instances.[4] Note that X and Y are not assumed to be be independent of each other. Otherwise labels would be independent of, and hence completely unrelated to features.

The features X take values in $\mathcal{X}$, where usually $\mathcal{X} \subseteq \mathbb{R}^K$ with the number of features $K \in \mathbb{N}^+$. Features, sometimes also called "attributes", "independent variables" or "predictors", represent the kind of measurements from which we want to predict the target variable. Note that all features are assumed to be static, or cross-sectional variables.

The target variable Y takes values in $\mathcal{Y}$. The target, also called "label" or "dependent variable", represents the outcome to be predicted. If $\mathcal{Y} = \mathbb{R}$, we call the learning task "cross-sectional regression". If $\mathcal{Y} = \mathcal{C}$ where $\mathcal{C}$ is a finite set of given class values, we call the learning task "cross-sectional classification". For example, for binary classification, one has $\mathcal{C} = \{0, 1\}$.

In practice, we observe realizations of these random variables as feature-target pairs $(x^{(i)}, y^{(i)})$, for instances $i = 1 \dots N$. In terms of the relational data structure, we usually collect these realizations in a $N \times K$ dimensional matrix for the feature data, with rows representing instances and columns representing variables, and a

[4]As is standard, for random variable $X : \Omega \rightarrow \mathcal{X}$, we write "$X$ takes values in $\mathcal{X}$". This is done without specifying the probability space Ω which will implicitly be assumed to exist and to be fixed, as usual to the finest such spaces of all appearing variables. Measurability of X is implicitly assumed.

$N \times 1$ dimensional vector for the target values, given by:

$$\mathbf{X} = \begin{bmatrix} x_1^{(1)} & x_2^{(1)} & \dots & x_K^{(1)} \\ x_1^{(2)} & x_2^{(2)} & \dots & x_K^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ x_1^{(N)} & x_2^{(N)} & \dots & x_K^{(N)} \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(N)} \end{bmatrix}$$

where the superscript denotes the instance and the subscript the feature variable.

Because of this representation, cross-sectional data is sometimes called “tabular data”. Note that the term “tabular data” refers to the shape of the data, i.e. a table, whereas “cross-sectional data” refers to the generative setting. Since other data can also be represented in a tabular format without necessarily being of cross-sectional nature, we prefer “cross-sectional data” throughout this thesis.

Also note that even though this setting assumes cross-sectional data with no repeated measurements over time, it can still be used to solve time series problem. These approaches will be discussed under the idea of “reduction” in Section 3.4.

Learning process specification

The assumed relationship we want to estimate is $Y \approx f(X)$ where f is an unknown function of X. So, a good learning algorithm should yield a prediction functional $\hat{f}$ that on average provides a good approximation to the putative relationship $Y \approx f(X)$. The estimation process, also called “fitting”, “learning” or “training”, tries to find the best possible approximation to the data used for training. However, the algorithm should not only perform well on the training data, but more importantly, should generalize well to new data.

To measure the generalization performance on new data, the data is usually split into two disjoint subsets: a set of training instances used for training the algorithm and a set of test instances reserved for evaluating the performance on unseen data.[5] Denote the training data as $D = \{(X^{(1)}, Y^{(1)}) \dots (X^{(M}, Y^{(M)})\}$

[5]Some references call the test set as the “hold-out set” because these data are “held out” of the data used for training. We will use “training set” and “test set” throughout this thesis.

where $M \in \mathbb{N}^+, M < N$ is the number of instances in the training set.

For a given training set, the learning process is to use algorithm A to fit a prediction functional $\hat{f} := A(D)$, where A is defined as follows:

$$A : (\mathcal{X} \times \mathcal{Y})^M \rightarrow [\mathcal{X} \rightarrow \mathcal{Y}]. \tag{2.4}$$

In words, algorithm A is a function that takes in data of type $(\mathcal{X} \times \mathcal{Y})^M$ and returns a function that maps input of type $\mathcal{X}$ to outputs of type $\mathcal{Y}$. That is, the output is the prediction functional $\hat{f} : \mathcal{X} \rightarrow \mathcal{Y}$. The *direction of information transfer* goes from the features to the target variable.

Trainable algorithms such as A are sometimes also called "learning machines", "estimators", "strategies" or "predictors", among others. For classification, we call A a "cross-sectional classifier". For regression, we call A a "cross-sectional regressor". This task is referred to as "supervised learning" to indicate that for every training instance $x^{(i)}$ the learning algorithm is provided with a teaching signal $y^{(i)}$.

It is important to distinguish prediction functionals such as $\hat{f}$ from learning algorithms such as A. Formally, $\hat{f} := A(D)$ is not a function, but a function-valued random variable which mediates the dependence of $\hat{f}$ on the data used to train it. While $\hat{f}$ may depend on D, it is assumed to be independent of anything else. Also note that algorithm A may also involve some randomness. For simplicity, we assume a deterministic dependence of $\hat{f}$ on D.

Success specification

To estimate the generalization performance, it is important to evaluate an algorithm on new data. In the case of cross-sectional data, new data refers to i.i.d. instances that have not been used during training. Having trained the algorithm on the training set, we can use the test set, denoted by (X^*, Y^*), for evaluation.

To quantify performance, we define a loss function $L : \mathcal{Y} \times \mathcal{Y} \rightarrow \mathbb{R}$ comparing the predicted value with the true value from the test set. For example, for regression, we can use the squared loss $L_{sq}(\hat{y}^*, y^*) = (\hat{y}^* - y^*)^2$. For classification, we can use the misclassification loss $L_c(\hat{y}^*, y^*) = \mathbb{1}_{\hat{y}^* \neq y^*}$, where $\mathbb{1}$ denotes the indicator function equal to one if $\hat{y}^* \neq y^*$ and zero otherwise.

For a given training set D, the learning task can then be understood as the optimization problem of finding the prediction functional which minimizes the generalization error defined as:

$$GE(\hat{f}|D) = \mathbb{E}[L(\hat{f}(X^*), Y^*)|D], \tag{2.5}$$

where expectations are taken over (X^*, Y^*), and $\hat{f}$ is fixed for the given training set D.

When the goal is evaluate different learning algorithms, it is important to make comparisons on a more general basis, accounting for the dependence of $\hat{f}$ on the training data D. In this case, we are interested in minimizing the expected generalization error given by:

$$EGE(A, D, X^*, Y^*) = \mathbb{E}[GE] = \mathbb{E}[L(A(D)(X^*), Y^*)]. \tag{2.6}$$

where expectations are now taken over everything that involves randomness, including the training set to account for the dependence of $\hat{f}$ on D.

Note that GE gives a conditional generalization error, since it depends on the training data used to train $\hat{f}$. As such, GE is a success criterion for a given prediction functional $\hat{f}$. EGE, on the other hand, gives an unconditional generalization error and is a success criterion for the learning algorithm A.

Both GE and EGE are defined in terms of random variables. They are theoretical quantities. In practice, they have to be estimated. Additional algorithms are needed for their estimation and comparison. These algorithm may take different forms and may be sensible to varying degrees according to statistical argument.

A final remark: The supervised learning tasks highlight that we can define tasks at different levels of generalization. A task may be defined as a sub-task of a more general task. For example, we can define classification and regression as sub-tasks of the more general supervised learning tasks, where the domain of the target variable is further restricted in otherwise identical task descriptions, i.e. for classification we have $\mathcal{Y} = \mathcal{C}$, for regression we have $\mathcal{Y} = \mathbb{R}$.

2.6 Concluding remarks

In this chapter, we gave an overview of the domain of ML with time series, with an example from chemical process engineering and focusing on point prediction tasks. We introduced some terminology to describe different forms of data which we will use throughout the thesis. We also introduced the concept of a learning task, which we will use as a template when we develop the taxonomy of time series tasks throughout the next chapter. To illustrate the concept of the learning task, we reviewed the traditional supervised learning tasks of cross-sectional classification and regression. These tasks will serves us as a familiar reference for comparison when we formalize time series tasks in the next chapter.

Chapter 3

Machine Learning with Time Series

This chapter is partly based on:

- Markus Löning et al. 'sktime: A Unified Interface for Machine Learning with Time Series'. In: *Workshop on Systems for ML at NeurIPS 2019* (2019)

3.1 Introduction

In the previous chapter, we have introduced the concept of a learning task and discussed cross-sectional supervised learning as an example. In this chapter, we will develop a taxonomy of time series learning tasks. The taxonomy is the first research contribution of this thesis as described in Section 1.6. It addresses the first set of research questions laid out in Section 1.5. The taxonomy will formalize key time series learning tasks. As such, it will form a key part of the conceptual model of the time series domain which underlies the unified software framework implemented in `sktime`. As we will see in Chapter 8, the structure and language of `sktime` closely corresponds to the taxonomy of tasks.

To recall, a time series consists of an indexed sequence of values measured repeatedly over time. Time series data are ubiquitous in real-world applications. Examples include price movements on financial markets, sensor readings in industrial processes (e.g. the temperature or pressure in a chemical reactor), patients' medical records (e.g. the blood pressure and heart rate), and customers' shopping histories. An inherent characteristic of time series data is that observations are statistically dependent on previous observations. Time series analysis, or ML with time series, is a set of techniques that is concerned with the analysis of this dependence. Analyzing time series is extremely important in real-world applications. It enables us to better understand the underlying process that give rise to the data we observe and make predictions about them. Ultimately, it guides our decision making and can improve outcomes. As discussed in Chapter 2, this thesis will focus on predictive learning tasks, leaving non-predictive tasks for future work.

As we have seen in the previous chapter, there are well-established "consensus" formulations for the learning task that arise in the more traditional cross-sectional setting. Many time series tasks, by contrast, are rarely fully formalized in the literature. For example, task descriptions tend to be incomplete. Often they only include the data and success specification, leaving the learning process unspecified (see e.g. [182]). As a result, learning problems become ambiguous. What should be distinct tasks become conflated under the same name. The lack of clear task specifications may potentially lead to unfair comparisons between algorithms, pairing of the wrong algorithm with the problem at hand, or over-optimistic performance estimates.

For example, consider the M4 forecasting competition [182]. The M4 competition is one of the major benchmarking studies for forecasting. It provides a systematic, reproducible and comprehensive comparison of forecasting algorithms. The competition guide provides a clear descriptions of the data and success criteria. However, it lacks a clear specification of the learning process. When looking at the accompanying code, we can see that some models, especially the "statistical" ones, are trained on a single series. The majority of the "ML" models, on the other hand, are trained on multiple series and hence can exploit consistent patters across series (e.g. the winner [234] and runner-up [192]). Comparing models trained on a single series with models trained on multiple series, without making the distinction clear, is an unfair and potentially misleading comparison of models. Technically, the learning task appears to be panel forecasting, sometimes also referred to as "cross-learning" [227], rather than univariate forecasting. We describe these learning tasks in Table 3.1. Note that even though many of the univariate models are originally designed to be trained on a single series (or require multivariate data to also be available during prediction), they can in principle still make use of multiple series in training, for example by optimizing hyper-parameters across series rather than through temporal cross-validation. We believe that a clearer task specification, including a description of the learning process, would have made these differences more explicit and avoided any confusion.

Moreover, since some models have been trained on all available series model performance estimates and statistical comparisons between these models may become unreliable. This is because, without a clear separation of series into a training and test sets, the assumption that each forecast represents an independent sample no longer holds. As a consequence, performance difference between models that appear statistically significant may in fact be insignificant after all. We do not intend to single out the M4 competition; other examples of implicit or incomplete task descriptions can be found in the literature. The aim of this chapter is to provide a formal definition of key time series tasks, with the hope to avoid ambiguity and reduce confusion in future research.

As we have seen, tasks describe distinct learning problems. At the same time, many time series tasks are closely related. One way to understand their relation is the idea of reduction. Reduction allows us to transfer knowledge between tasks and to apply an algorithm for one task to help solve another. Reduction is central to time series analysis. Many reduction approaches are possible and various time series algorithms rely on reduction. However, not only are time series tasks rarely fully formalized in the literature, they are also rarely discussed in relation to each other. To close this gap, we will discuss reduction relations between tasks at the end of this chapter. Clearly separating these learning tasks, while at the same time understanding their relations, will be crucial for finding useful abstractions and a modular design for the unified software framework developed in Chapter 8.

The rest of this chapter is organized as follows: we will start by formalizing key temporal data generative settings in Section 3.2. We will then develop the taxonomy of time series learning tasks in Section 3.3, focusing on three tasks, namely forecasting, time series classification/regression and supervised forecasting. Having clarified the differences between these tasks, we will discuss how they are related through reduction in Section 3.4.

3.2 Time series data

As we have seen, the cross-sectional supervised learning task is characterized by a single data generative setting, i.e. cross-sectional data consisting of observations on multiple instances of one or more variables at a single time point. Time series data, by contrast, can arise in different generative settings. Understanding the different settings that such data may come from is important for understanding the learning tasks that arise in these settings.

We will use the terminology introduced in Section 2.5. In some situations, we may observe a single time series, consisting of observations on a *single instance* (experimental unit) and a *single variable* (kind of measurement) at *multiple time points*. We call this “univariate” data. In other situations, we may observe multiple series. This can happen in two fundamentally different ways. We may observe multiple series representing observations on a *single instance* of *multiple variables*, called “multivariate” data. Or we may observe multiple series representing observations on *multiple instances* of one or more variables, called “panel” data.

We will now formalize these three forms of time series data and the assumed generative processes that can give rise to such data. Afterwards, we will develop the taxonomy of time series tasks which will refer to these forms of data.

3.2.1 Univariate time series

For the discussion of univariate and multivariate series in this and the next section, we partly follow the exposition in Box et al. [39].

A time series consists of an indexed sequence of values observed sequentially over time. We consider a time series to consist of both:

- the *values* we observe,
- the *time points* or *index* at which we observe the values.

While the values tell us what we observed, the index tells us when we observed it.

Formally, we denote a time series by $(z(t_1), z(t_2), \ldots, z(t_T))$, where $z(t)$ represents an observation at time point t. T is the total number of observations, with $T \in \mathbb{N}^+$. We may also represent a univariate time series can be represented in terms of a $T \times 1$ dimensional vector given by:

$$\mathbf{z} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_T \end{bmatrix}$$

Note that to keep our notation simple, we do not explicitly carry the time index. Technically, in our notation, a time series is only a sequence of values without time indices. We assume that the associated information of the time index is available whenever needed.

Data generative setting

If the values of a time series are exactly determined by some mathematical function, for example, $z(t) = f(t)$ for some function f and time point t, the time series is said to be deterministic. If, on the other hand, values can only be described in terms of a probability distribution, the time series is said to be non-deterministic or stochastic. Throughout this thesis, we are concerned with stochastic time series.

A statistical phenomenon that evolves over time according to a probability distribution is called a "stochastic process". A time series that we observe can be thought of as a particular realization produced by a stochastic process. Formally, we can define a the assumed generative process as follows.

Let with Z_t be a random variable taking values in the domain $\mathcal{Z}$ with some probability distribution. We discuss the value domain, $\mathcal{Z}$, in more detail below. It follows that we can regard the observation $z(t)$ as a realization of Z_t at time point t. In general, a time series $(z(t_1), z(t_2), \dots, z(t_T))$ consisting of multiple observations can be regarded as a realization of a multi-dimensional random variable, $Z = (Z_{t_1}, Z_{t_2}, \dots, Z_{t_T})$, with a joint probability distribution. The joint probability distribution captures the inherent characteristic of time series that realizations depend on adjacent realizations. Note that the time points and the number of time points are assumed to be fixed. Only the values observed at those time points are regarded as random quantities.

Value domain

For the value domain of a time series, one usually has $\mathcal{Z} \subseteq \mathbb{R}$, where $\mathbb{R}$ represents the set of real numbers. For example, the random variable Z_t may represent the temperature of a chemical process with temperature being a continuous variable. In this case, the time series is called "value-continuous" and takes values from a continuous domain (e.g. the real numbers).

In other cases, a time series may take on values from different domains, depending on the kind of measurement that the random variable represents. For example, one may have $\mathcal{Z} \subseteq \mathbb{N}^+$ (e.g. count values) or one may have $\mathcal{Z} \subseteq \mathcal{C}$, where $\mathcal{C}$ represents a finite set of some discrete values (e.g. categories). In these cases, the time series is called "value-discrete".

Time domain

The time index usually represents a point in time. We denote the ordered sequence of time points at which we make observations as $\mathbf{t} = (t_i \in \mathcal{T}$ for all $i = 1, \dots, T$; $t_i < t_j$ for all $i < j$ with $i, j \in \mathbb{N}^+)$. $T \in \mathbb{N}^+$ denotes the total number of observations. We can then define the domain of the random variable Z

as series($\mathcal{Z}, \mathbf{t}$), where we define:

$$\text{series}(\mathcal{Z}, \mathbf{t}) \coloneqq \{(z(t_i) : z(t_i) \in \mathcal{Z}; t_i \in \mathbf{t})\} \tag{3.1}$$

as the set of tuples of values ordered in time for given time points in $\mathbf{t}$. We will often further abbreviate this to series($\mathcal{Z}$), assuming the time points are given.

For the time domain, $\mathcal{T}$, one usually has $\mathcal{T} \subseteq \mathbb{R}^+$, meaning that time points lie on the positive real half-line. However, in some situation, it may also represent other scales, as discussed in remark 1.

Remark 1 (Non-temporal sequential data)**.** Time series data, as defined here, can also arise in non-temporal settings, in the sense that the data may not come from a process which obeys the direction or flow of time. Other data falls under the definition too as long as it consists of sequentially ordered values. Examples of non-temporal but sequential data include wave length data (e.g. from spectroscopy) or image outlines unrolled into a sequence with an arbitrary starting point using the distance from the image center to points on the image contour as values.[1]

Regardless of its domain, the index always denotes the position of its associated value in the series. As such, it carries the orderedness information of the data. Depending on the domain of the index, it may contain additional information. For example, an index representing points in absolute time may carry additional calendar information such as the day of the week or whether it is a holiday.

If the index, or set of time points, is continuous, the time series is said to be "time-continuous". If the set of time points is discrete, the time series is said to be "time-discrete". Throughout this thesis, we are only concerned with time-discrete series. Time-discrete series may arise in two ways:

[1]For an explanation of this transformation from image outlines to sequences, sometimes called "radial scanning", see `https://izbicki.me/blog/converting-images-into-timeseries-for-data-mining.html`. For papers that use data from radial scanning, see e.g. Keogh et al. [144] and Tak and Hwang [242]. Note that this transformation can be understood as a reduction from image learning task to a time series learning task, with hyper-parameters such as the starting point, the direction of unrolling, the choice of center point and the number of time points. Reduction is discussed in more detail in Section 3.4.

- By *sampling* a time-continuous time series at a number of time points (e.g. temperature readings during a chemical process at one minute intervals), or
- By *accumulating* a time-continuous time series over a period of time (e.g. measuring accumulated rain fall over a day).

A common simplifying assumption is that intervals between time points are regular, that is, fixed and of equal length. Formally, we can assume that the T successive values of a series are made at equidistant time points $t_0 + d$, $t_0 + 2d$, ..., $t_0 + Td$ where d is a fixed interval between the time points. This allows us to write the time series $z(t_1), z(t_2), ..., z(t_T)$ simply as z_1, z_2, ..., z_T. For many purposes, the values of the t_0 and d are unimportant, but if the observation times need to be defined exactly, these two values can be specified. If we adopt t_0 as the origin and d as the unit of time, we can regard z_t as the observation at time t. We discuss deviations from this assumption in remark 2.

Remark 2 (Random event times)**.** In some situations, observations may not be recorded as regular time series with fixed and equal time intervals. For example, we may observe a sequence of events with exact time points representing random event times. Events may, for example, represent machine failures, shopping transactions or hospital admissions, and we observe the exact time point of each event. Formally, we can define such time points on the positive half-line $[0, \tau]$ with $\tau \in \mathbb{R}^+$, such that $(0 \leq t_1 < t_2 < t_3 < ... < t_{N(\tau)} \leq \tau)$, where $N : [0, \tau] \to \mathbb{N}^+$ is a non-decreasing, piece-wise constant counting function which makes unit steps at the event times. In contrast to regular time series with fixed, equal-length intervals, there is not just additional information in the order of observations, but also in the length of the observed interval between successive observations, sometimes called the "wait time". In order to capture the inherent uncertainty in these processes, a common approach is to use point process models (see e.g. Daley and Vere-Jones [64]).

So far we have discussed single or univariate time series. In many situations, we observe multiple series. It is crucial to distinguish two fundamentally different ways in which this may happen: multivariate time series and panel data. We will consider multivariate data first and then turn to panel data in the following section.

3.2.2 Multivariate time series

A multivariate time series consists of multiple, related univariate series. Like univariate series, multivariate series consists of observations on a *single instance*. Unlike univariate series, multivariate series consist of observations on *multiple variables*. As before, instances represent experimental units and variables different kinds of measurements. For example, we may observe the temperature and pressure from a single chemical process.

Formally, we can define a multivariate stochastic process as multi-dimensional random variable taking values in the domain, series($\mathcal{Z}$), with a joint probability distribution over time points and variables. One now has $\mathcal{Z} \subseteq \mathbb{R}^L$. L is the number of variables, with $L \in \mathbb{N}^+$. Note that we here make the simplifying assumption that all univariate series take values in the same domain. We discuss deviation from this assumption in remark 4.

Denote the observation z_{jt} as a realization of the random variable Z_{jt} for variable j at time point t, with $j = 1 \ldots L$. If we assume that the set of time points $\mathbf{t}$ is shared across variables, we can represent multivariate time series as a $T \times L$ dimensional matrix given by:

$$\mathbf{Z} = \begin{bmatrix} z_{11} & z_{21} & \cdots & z_{L1} \\ z_{12} & z_{22} & \cdots & z_{L2} \\ \vdots & \vdots & \vdots & \vdots \\ z_{1T} & z_{2T} & \cdots & z_{LT} \end{bmatrix}$$

with rows representing time points and columns representing variables.

The joint probability distribution captures the inherent characteristic of multivariate time series that observations are not just statistically dependent on past observations within the same univariate series, but also across series, both contemporaneously and across time points. Some series may lead other series, or there may exist feedback relationships between the series. Hence, it is implausible to assume that the different univariate component series are statistically independent. This is the key difference with regard to panel data discussed below.

Remark 3 (Different time indices across variables)**.** A complication in some situations is that observed time indices vary across univariate component series. Series may not share a common index. Specifically, series may exhibit one or both of the following:

- *varying number of time points* (sometimes also called "unequal-length" or "variable-length" data),
- *fixed time intervals within series but varying intervals between series* (sometimes called "unequally spaced" data).

Of course, time points may also be completely random, as discussed in remark 2. This is an important consideration for the design of a suitable time series data container discussed in Chapter 8.

Remark 4 (Different value domains across variables)**.** Another frequent complication is that observed univariate component series may have different value domains, $\mathscr{Z}$. Again this is relevant for the design of a suitable time series data container discussed in Chapter 8.

Multivariate time series is only one of the ways of observing multiple time series. The other way is panel data which we will discuss now.

3.2.3 Panel data

Panel data, sometimes also called "longitudinal data" [74], consists of observation on *multiple instances* of the *same kind(s) of measurements* at *multiple points in time.* Like cross-sectional data, panel data consists of observations on multiple instances. Unlike cross-sectional data, panel data consists of repeated observations over time. Put differently, panel data consists of multiple cross-sections taken repeatedly over time.

Formally, we can represent the underlying data-generating process by N jointly i.i.d. random variables given by:

$$(Z^{(1)}, \ldots, Z^{(N)}) \overset{i.i.d.}{\sim} Z, \tag{3.2}$$

where the random variable Z takes values in series$(\mathcal{Z})$ and $N \in \mathbb{N}^+$. In contrast to multivariate time series, in panel data, it is plausible to assume that each instance is an i.i.d. realization of the same stochastic process. Note that the i.i.d. assumption is only plausible when applied to the different instances. It is implausible when applied to the time series observations within a given instance, as they may still depend on past observations. Of course, we may also observe panel data with multivariate time series. In this case, the different instances are still i.i.d., but the univariate time series variables within an instance are not, as discussed in Section 3.2.2.

To ease notation, we will make the simplifying assumption of observing what we call "time-homogeneous" data. Time-homogeneous data contains observations which share a common time index across both instances and variables. For example, in time-homogeneous multivariate time series, all univariate component series share the same time index. In time-homogeneous panel data, all variables of all instances share the same time index.

A simple case of panel data is where we observe N instances of a univariate time series, denoted as $\mathbf{z}^{(i)} = (z_1^{(i)}, z_2^{(i)}, ..., z_T^{(i)})$, $i = 1 \ldots N$, for T time points.

This can be represented in a $N \times T$ dimensional matrix given by:

$$\mathbf{Z} = \begin{bmatrix} z_1^{(1)} & z_2^{(1)} & \dots & z_T^{(1)} \\ z_1^{(2)} & z_2^{(2)} & \dots & z_T^{(2)} \\ \vdots & \vdots & \vdots & \vdots \\ z_1^{(N)} & z_2^{(N)} & \dots & z_T^{(N)} \end{bmatrix}$$

Here, rows represent i.i.d. instances and columns represent time points. Note that multivariate panel data does not fit into the tabular format, at least not without making further assumptions. Typically, multivariate panel data requires three-dimensional matrices or hierarchical data structures.

An important case for our purpose is mixed panel data. Mixed panel data consists of instances with observations from both time series and cross-sectional variables. For example, data on patients for who we observe cross-sectional variables (e.g. the date of birth) as well as time series variables (e.g. the heart rate). In this case, we represent the underlying data-generating process as follows:

$$((Z^{(1)}, X^{(1)}), \dots, (Z^{(N)}, X^{(N)})) \overset{i.i.d.}{\sim} (Z, X) \tag{3.3}$$

Here, (Z, X) are random variables, with $Z = (Z_1, Z_2, ..., Z_T)$, which take values in the domain series$(\mathcal{Z}) \times \mathcal{X}$. As in the cross-sectional setting, discussed in Section 2.5, one usually has $\mathcal{X} \subseteq \mathbb{R}^K$, where K denotes the number of cross-sectional features. Note that a cross-sectional variable may be regarded as the target variable to be predicted from the available time series and any other time constant variables. We return to this case in the context of time series classification and time series regression in Section 3.3.1.

Remark 5 (Different time indices across instances)**.** In panel data, time indices may not only vary across variables, as discussed in remark 3, but also across instances. We may observe (i) different number of time points for different instances and/or (ii) different intervals between time points. In this sense, panel data is sometimes described as being "asynchronous", in contrast to "synchronous" data with shared time indices. Of course, time points may also be completely random as discussed in remark 2.

Remark 6 (Different variables across instances)**.** An additional but less common complication is that instances may have *varying numbers of variables*, e.g. some variables may only be observed for some instances.

Until now, we have reviewed three different time series data forms including univariate, multivariate and panel data. We highlighted differences between the generative settings in which these data forms can arise and compared them to the cross-sectional setting from the previous chapter. Based on these data descriptions, we will formalize key time series learning tasks in the next section.

3.3 A taxonomy of time series learning tasks

The variety of generative scenarios is mirrored in a variety of learning tasks that arise from such data. We give a stylized overview of key time series learning tasks in Table 3.1. In this section, we will formalize three of these tasks, using the template defined in Section 2.3, namely time series classification, time series regression and forecasting. These are three fundamental time series tasks. The formalization for these tasks will serve as a foundation for the development of a unified software framework for ML with time series. The formalization of other tasks is left for future work. Throughout the discussion of learning tasks below we refer to the different data settings defined in Section 3.2 above.

3.3.1 Time series classification/regression

Two of the most fundamental time series learning tasks are time series classification and time series regression.[2] In these tasks, we are interested in predicting a cross-sectional target variable from one or more input time series based on patterns

[2] For an overview of time series classification algorithms, see e.g. Bagnall et al. [14] and Ismail Fawaz et al. [131]. For an overview of time series regression, see e.g. Tan et al. [243].

Table 3.1: Stylized overview of key time series learning tasks

Name	Data	Description
Forecasting	Time series	For a given forecasting horizon, make a temporal forward prediction of a univariate time series based on patterns found across time points in past data [39].
Multivariate forecasting	Time series	For a given forecasting horizon, make a temporal forward prediction of a multivariate series based on patterns found across individual series and time points in past data [175].
Panel forecasting	Panel data	For a given instance and forecasting horizon, make a temporal forward prediction based on patterns found across instances and time points in past data where all instances are partially observed up to a given point in time. Instances are typically observed on a common time scale (e.g. countries observed over the same time period) [16].
Supervised forecasting	Panel data	For a given instance and forecasting horizon, make a temporal forward prediction based on patterns found across instances and time points where some instances are fully observed up to the end of the forecasting horizon. Instances are typically observed on individual time scales (e.g. chemical process runs as described in the example in Section 2.2).
Classification	Panel data with a target variable	For a given instance, predict the value of the categorical cross-sectional target variable with predefined class labels based on patterns found across instances between the time series and the target variable [14].
Regression	Panel data with a target variable	For a given instance, predict the value of the continuous cross-sectional target variable based on patterns found across instances between time series and the target variable [243].
Clustering	Panel data	Find groups of time series based on the similarity between instances [2].
Annotation	Time series	Annotate time points of a series where tasks varies by the value domain and interpretation of the annotations, including: (i) outlier detection [110], (ii) anomaly detection [52], (iii) change-point detection [5], (iv) segmentation [143].
Supervised annotation	Panel data with annotations	For a given instance, annotate time points based on patterns found across instances and time points between values and annotations [73].

Notes: This table gives a stylized, informal description of key time series learning tasks. We will formalize some of these tasks in Section 3.3. All of these tasks can also arise in an online learning setting where new data becomes available incrementally and predictions have to be generated repeatedly as time moves on.

found across instances between the input series and the target variable. These tasks are direct extensions of cross-sectional classification and regression to panel data.

We distinguish cross-sectional and time series variants of these learning task because they assume different data generative settings. It may not be obvious at first sight how these tasks are different. The key difference is that features are now series, rather than scalars (e.g. numbers, categories or strings). For this reason, we will sometimes call these tasks "series-as-features" tasks.

Data specification

We assume mixed panel data as defined in Section 3.2.3. Let

$$(Z^{(1)}, Y^{(1)}) \dots (Z^{(N)}, Y^{(N)}) \overset{i.i.d.}{\sim} (Z, Y) \tag{3.4}$$

be jointly i.i.d. random variables with the number of instances $N \in \mathbb{N}^+$. Z represents time series feature variables. Y represents the cross-sectional target variable. The random variables Z and Y are not assumed to be independent of each other. Otherwise target values would be independent of, and hence completely unrelated to features.

The time series features Z take values in series($\mathcal{Z}$) for a given set of time points $\mathbf{t}$. Z may be univariate or multivariate. As in the cross-sectional setting, the random variable Y takes values in $\mathcal{Y}$. If $\mathcal{Y} = \mathbb{R}$, we call the learning task "time series regression". If $\mathcal{Y} = \mathcal{C}$ where $\mathcal{C}$ is a finite set of given class values, we call the learning task "time series classification".

In practice, we observe realizations of $(Z^{(i)}, Y^{(i)})$ as feature-target pairs $(z^{(i)}, y^{(i)})$, $i = 1 \dots N$, where $z^{(i)} = (z_t^{(i)} : t \in \mathbf{t})$ is a time series for given time points t in $\mathbf{t}$. For simplicity, we assume time-homogeneous data, as defined in Section 3.2.3.

Remark 7 (Additional time-invariant features)**.** There may be additional time-invariant features X taking values in $\mathcal{X} \subseteq \mathbb{R}^K$ as in the traditional setting. For simplicity, we ignore them here, but the description can easily be generalized to account for extra time-invariant features.

Learning process specification

The assumed relationship we want to estimate is $Y \approx f(Z)$ where f is a fixed but unknown function of time series Z . The learning process is similar to the cross-sectional setting, with the only difference that the feature variables are now series rather than scalars. We split the data into set of training instances $D = \{(Z^{(1)}, Y^{(1)}) \dots (Z^{(M)}, Y^{(M)})\}$ used for training the algorithm and a set of test instances (Z^*, Y^*) reserved for evaluating the performance, where the number of training instances $M \in \mathbb{N}^+, M < N$.

For a given training set, the learning process is to use an algorithm A to fit a prediction functional $\hat{f} \coloneqq A(D)$ where A is given by:

$$A : (\text{series}(\mathcal{Z}) \times \mathcal{Y})^M \rightarrow [\text{series}(\mathcal{Z}) \rightarrow \mathcal{Y}] \tag{3.5}$$

with prediction functional $\hat{f} : \text{series}(\mathcal{Z}) \rightarrow \mathcal{Y}$. The direction of information transfer goes from the time series variables to the target variable. For classification, we call A a "time series classifier". For regression, we call A a "time series regressor".

Algorithm A may involve some randomness. For simplicity, we assume a deterministic dependence of $\hat{f}$ on D. As before, this task is referred to as "supervised learning" because for every training instance $x^{(i)}$ the learning algorithm is provided with a teaching signal $y^{(i)}$.

We can now further clarify the difference between cross-sectional and time series supervised learning algorithms. Cross-sectional algorithms take in cross-sectional feature data, whereas time series algorithms take in panel feature data. Time series algorithms make use of the information contained in the ordering of the time series observations. By contrast, shuffling the order of the features in the cross-sectional setting will not have an effect on the fitting of cross-sectional supervised learning algorithms (barring negligible changes due to any randomness involved in these algorithms). On the other hand, shuffling the order of time series observations may have an effect on the fitting of time series algorithms.

Nevertheless, cross-sectional algorithms can still be used to solve time series tasks. We discuss these approaches under the idea of reduction in Section 3.4.

Success specification

Evaluation for time series classification and time series regression is analogous to the cross-sectional setting discussed in Section 2.5. To recall, we evaluate performance on new data, where new data refers to independent instances not used for training. Having trained the algorithm on the training set, we can use the test instances (Z^*, Y^*) for evaluation.

To quantify performance, we define a loss function $L : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$. For a given training set D, we want to minimize the generalization error:

$$GE(\hat{f}, Z^*, Y^* | D) = \mathbb{E}[L(\hat{f}(Z^*), Y^*) | D] \tag{3.6}$$

To account for the dependence of $\hat{f}$ on the training data D, we want to minimize the expected generalization error:

$$EGE(A, D, Z^*, Y^*) = \mathbb{E}[GE] = \mathbb{E}[L(A(D)(Z^*), Y^*)]. \tag{3.7}$$

Both GE and EGE are theoretical quantities and have to be estimated in practice.

3.3.2 Forecasting

Besides time series classification and regression, one of the most fundamental time series learning tasks is forecasting.[3] In forecasting, we are interested in making a temporal forward prediction of a univariate series based on patterns found across time points in past data.

[3]For a textbook overview of forecasting, see e.g. Box et al. [39], Brockwell and Davis [43] and De Gooijer and Hyndman [65] or Hyndman and Athanasopoulos [122].

The forecasting task comes in different variations. Our definition below focuses on the simple univariate case without exogenous variables. Other variations are described in Table 3.1.

Data specification

We assume univariate time series defined in Section 3.2.1. Let Z be a random variable for given time points $\mathbf{t} = (t_1, t_2, \dots t_T)$, where $Z = (Z_t : t \in \mathbf{t})$, taking values in series($\mathcal{Z}$), with $\mathcal{Z} \subseteq \mathbb{R}^L$, $L = 1$ meaning that Z is univariate.

In practice, we observe realization $z = (z_t : t \in \mathbf{t})$ for given time points t in $\mathbf{t}$. Importantly, we observe only a single instance of the time series. No assumption of independence is made.

Learning process specification

The assumed relationship we want to estimate is $Z_t \approx f(t)$ where f is a fixed but unknown function of time t. As before, a good learning algorithm gives a good approximation to the assumed relationship so that we can predict $\hat{z}_t = \hat{f}(t)$ at some time point t. The estimation process aims to find the best possible approximation to the training data. However, the algorithm should not just perform well on the training data, but more importantly generalize well to new data. Consequently, it is important to evaluate algorithms on genuine forecasts. For this reason, we split the available data into two disjoint subsets, called a training and test set, where the training set is used for training the algorithm and the test set is used for evaluating its performance. Because the test set is not used in training the algorithms and hence determining the forecasts, it should provide a reliable indication of how well the model is likely to perform on new data. The training set is also sometimes called "in-sample data" and the test set "out-of-sample data". We prefer "training set" and "test set" as the more common terms in the ML literature.

Formally, we split the observed time points into a set of training time points $\mathbf{t}_\tau = (t_i : t_i \in \mathbf{t}; t_i \leq \tau)$ used for training the algorithm and a set of test time points t_*, $t_T \geq t_* > \tau$ reserved for evaluating the performance. The cut-off point τ is defined by $\tau \in \mathcal{T}$, $\tau < t_T$, where $\mathcal{T}$ denotes the time domain as defined in Section 3.2. We call τ the "cutoff point" because it cuts off the test set from the training set. Some references call it the "origin". It represents the point in time at which we want to generate predictions. Usually, the cutoff point is the last time point in the training set. We denote the training data as $D = (Z_t : t \in \mathbf{t}_\tau)$, and the test observation at time point t_* as z_{t_*}, or simply z_*.

For a given training set, the learning process is to use an algorithm A to fit a prediction functional $\hat{f} := A(D)$. A can be defined as follows:

$$A : \text{series}(\mathcal{Z}) \rightarrow [\mathcal{T} \rightarrow \text{series}(\mathcal{Z})]. \tag{3.8}$$

In words, algorithm A is a function that takes in data of type series($\mathcal{Z}$) and returns a function that maps input of type $\mathcal{T}$ to outputs of type series($\mathcal{Z}$). That is, the output is the prediction functional of type $\hat{f} : \mathcal{T} \rightarrow \text{series}(\mathcal{Z})$. The prediction functional takes in a new time point, called the "forecasting horizon", and returns a prediction for that time point. The direction of information transfer goes from past to future observations. We call A a "forecaster".

Algorithm A may involve some randomness, but as before we assume a deterministic dependence of $\hat{f}$ on D. Forecasting may also be understood as a "supervised" learning task. For most observations in the training series z_t, the learning algorithm is provided with the next observation z_{t+1} as a teaching signal. However, the algorithm is not presented with different independent instances of the same stochastic process.

We make a few additional remarks:

- In the definition above, the forecasting horizon consists of a single time point. In practice, we may want to predict values for multiple time points at the same time. Many prediction functionals require predictions for previous time points in order to generate predictions for time points multiple steps ahead. For simplicity, we assume that these predictions are available to the prediction functional.

- The fitted prediction functional usually requires not only a time point, but also some past data to generate predictions. So we could define $\hat{f}$ instead as $\hat{f} : \text{series}(\mathcal{Z}) \times \mathcal{T} \to \text{series}(\mathcal{Z})$. Algorithm A would then be given by:

  $$A : \text{series}(\mathcal{Z}) \to [\text{series}(\mathcal{Z}) \times \mathcal{T} \to \text{series}(\mathcal{Z})]$$

  For simplicity, we assume that the required past data is available to the fitted prediction functional and continue with algorithm A as defined in Equation 3.8.

- Some algorithms require the forecasting horizon already during training. This includes all those algorithms that fit separate parameters for each time point in the forecasting horizon (e.g. the "direct" reduction strategy [34]). While these considerations are important for the design of `sktime` in Section 8.3.3, we continue with the simpler notation in Equation 3.8.

Success specification

In general, the way in which we evaluate models should simulate the way in which we deploy the model in the real-world application in question. We here focus on the deployment situation of a single, fixed cutoff point in which we generate predictions for a given forecasting horizon. We leave aside the alternative setting of a moving cutoff point, sometimes also called "dynamic" or "online" learning. To estimate the generalization performance of an algorithm, it is important to evaluate it on new data. The size of the residuals from the training set is therefore not a reliable indication of how large true forecast errors are likely to be. The accuracy of forecasts can only be determined by considering how well a model performs compared to new data that was not used for training the model. New data here refers to observations at future time points not used during training. Having used the time points $\mathbf{t}_\tau = (t_i : t_i \in \mathbf{t}; t_i \leq \tau)$ for training, we can use test time points t_*, $t_T \geq t_* > \tau$ for evaluation.

To quantify performance, we define loss function $L : \mathcal{Z} \times \mathcal{Z} \rightarrow \mathbb{R}$, comparing the predicted value with the true value at a given time point. For a given training set D, the learning task can then be understood as the optimization problem of minimizing the generalization error at the test time point t_* given by:

$$GE(\hat{f}, t_*, z_*) = \mathbb{E}[L(\hat{f}(t_*), z(t_*))|D], \tag{3.9}$$

where expectations are taken over $z(t_*)$.

To account for the dependence of $\hat{f}$ on the training data D, we can estimate the expected generalization error:

$$EGE(A, D, t_*, z_*) = \mathbb{E}[GE] = \mathbb{E}[L(A(D)(t_*), z(t_*))]. \tag{3.10}$$

where we now take expectation over everything that involves randomness, including the training set.

We make a few additional remarks:

- Performance estimates are only reliable to the extent that the test set is representative of the data-generative process. In order to estimate the expected performance on new data based on past data, we therefore need to assume that the underlying data-generative process follows some statistical equilibrium, typically expressed in an assumption of stationarity [38, chap. 2]. Without such an assumption, performance estimates based on the training data may not generalize to new data.

- As before, both GE and EGE are defined in terms of random variables. They are theoretical quantities. They have to be estimated in practice.

- GE gives a conditional generalization error, since it depends on the training data used to train $\hat{f}$. As such, GE is a success criterion for a given prediction functional $\hat{f}$. EGE, on the other hand, gives an unconditional generalization error and is a success criterion for the learning algorithm A.

- GE and EGE are defined for a single time point t_*, but we may be interested in aggregate errors over a range of test time points. If we want to evaluate the performance over multiple time points, we can aggregate individual loss values in different ways. A simple strategy is to take the mean. Common examples of loss functions used in forecasting are the symmetric mean absolute percentage error (sMAPE) and mean absolute scaled error (MASE) [125].
- Note that in contrast to the cross-sectional and series-as-features setting with independent instances, here the test set is not independent of the training set. The test set comes from the same time series as the training set and hence they are likely to be statistically dependent on each other. As a consequence, when estimating the uncertainty of our performance estimates, we have to take into account the dependency structure of the data.

3.4 Reduction

The above definitions of learning tasks have emphasized the differences between tasks. In this section, we will highlight the relations between them. One way to understand how tasks are related is the idea of "reduction". In the context of ML, reduction often refers to "data reduction" or "dimensionality reduction". Instead, reduction here refers to a technique for converting one task into another Beygelzimer et al. [24]. As such, reduction allows us to apply an any algorithm for the original task to solve the new one.

While reductions are not new, they are rarely discussed in the time series analysis literature as a concept that relates different tasks. Reductions are central to ML with time series. Figure 3.1 gives an overview of the most important reduction relations in the time series domain. Reduction typically works by decomposing a complex tasks into simpler tasks so that solutions to the simpler tasks can be recomposed to give a solution to the complex task. A classical example in supervised learning is one-vs-all classification, reducing k-way multi-category

Figure 3.1: Stylized overview of time series reduction relations

$$\begin{tikzcd}
\text{Forecasting} \arrow[r, shift left, "b"] \arrow[d, shift right, "a"'] & \text{Panel forecasting} \arrow[l, shift left, "c"] \arrow[r, "d"] \arrow[d, shift right, "a"'] & \text{Time series regression} \arrow[r, "g"] \arrow[d, "f"'] & \text{Cross-sectional regression} \arrow[d, "f"'] \\
\text{Annotation} \arrow[r, shift left, "b"] \arrow[u, shift right, "h"'] & \text{Supervised annotation} \arrow[l, shift left, "c"] \arrow[r, "e"] \arrow[u, shift right, "h"'] & \text{Time series classification} \arrow[r, "g"] & \text{Cross-sectional classification} \\
\underbrace{\hphantom{\text{Annotation}}}_{\text{Time series}} & \underbrace{\hphantom{\text{Supervised annotation} \quad \text{Time series classification}}}_{\text{Panel data}} & & \underbrace{\hphantom{\text{Cross-sectional}}}_{\text{Cross-sectional data}}
\end{tikzcd}$$

Notes: The figure shows learning tasks connected by arrows. Table 3.1 gives a description of the time series learning tasks. Section 2.5 describes the cross-sectional tasks. Arrows indicate reduction relations, including: (a) annotate time series with future values, (b) sliding-window transformation to convert single series into panel data with multiple output time periods [34], (c) ignore training set (e.g. fit forecaster on test set only) or use training set for model selection, (d) iterate over output periods, optionally time binning/aggregation of output periods [34], (e) sliding-window transformation to convert single series into panel data with single output period [73], (f) discretize output into one or more bins, (g) feature extraction [55, 88] or time binning/aggregation of input time points, (h) annotate based on forecasts (e.g. within or without predicted interval forecasts).

classification to k binary classification tasks [24, 26]. With time series, many reduction approaches are possible. Reductions are widely used in time series analysis, both in applications and as components in specialized time series algorithms. For example, cross-sectional regressors are often used to solve forecasting problems (see e.g. the M4 competition [182]). Similarly, cross-sectional classifiers are often used as components inside specialized time series classification algorithms (see e.g. the review by Bagnall et al. [14]).

Reductions can be understood as mappings of an algorithm of one type to an algorithm of another type. For example, the reduction from forecasting to cross-sectional regression can be understood as a mapping operation that takes in an algorithm of the cross-sectional regressor type as input and returns an algorithm of the forecaster type. As we will see, reduction plays a key role in the design of the proposed unified software framework for ML with time series. We discuss reduction more formally in Chapter 6 and 8. In the following, we first describe two examples in more detail and then highlight key properties of reduction relations.

Example 1 (Reducing time series classification to cross-sectional classification)**.** A common example is reduction from time series classification to cross-sectional classification. This reduction allows us to apply any of the more established cross-sectional classification algorithms to a time series classification task. In Figure 3.1, this reduction relation is indicated by the (g) arrow in the bottom row. There are various strategies to perform this reduction. All of them involve feature extraction as a way to transform the time series features into scalar features as the required input format by cross-sectional classification algorithms. A trivial strategy is to treat each time point as a separate feature, ignoring any information contained in the order of the time series data. A popular strategy is to extract scalar features from the time series. Characteristics range from simple ones such as the mean to more advanced ones such as Fourier transform coefficients or the time reversal asymmetry statistic [55, 56].

Example 2 (Reducing forecasting to cross-sectional regression)**.** Another common example is reduction from forecasting to a cross-sectional regression. This reduction allows us to apply any cross-sectional regression algorithm to a forecasting task. In Figure 3.1, this reduction relation is indicated by the top-row arrows from (b) through (d) to (g). There are various strategies to perform this reduction. A common strategy, called the "recursive" strategy [34], is as follows: We first split the training time series into fixed-length windows using a sliding window, and then stack all windows on top of each other. This gives us the feature data in the required tabular format, with each row representing one of the windows and each column a time point. To obtain the target variable, we simply take the next value after each window. For a univariate time series $(z_1, z_2, \ldots, z_T)$ and window length w, the resultant $T - w \times w$ dimensional feature matrix and $T - w \times 1$ dimensional target vector are given by:

$$\mathbf{X} = \begin{bmatrix} z_1 & z_2 & \dots & z_w \\ z_2 & z_3 & \dots & z_{w+1} \\ \vdots & \vdots & \vdots & \vdots \\ z_{T-w} & z_{T-w+1} & \dots & z_{T-1} \end{bmatrix} \qquad \mathbf{y} = \begin{bmatrix} z_{w+1} \\ z_{w+2} \\ \vdots \\ z_T \end{bmatrix}$$

where T denotes the total number of time points in the time series, with T, $w \in \mathbb{N}^+$ and $T > w$. Since we regress the target variable on its lagged values, this approach is sometimes also called "'auto-regression" or "lagged-variable regression". Having transformed the series into the required tabular format, we can then fit any cross-sectional regression algorithm to the data.

Once we have a fitted algorithm, we use the last available window $[z_{T-w+1}, z_{T-w+2}, \dots, z_T]$ as input to the fitted algorithm to generate the first-step-ahead prediction. For multi-step-ahead predictions, we recursively (i) update the last window using previously predicted values and (ii) generate new predictions for the next step ahead. Other strategies than the recursive one are possible (for an overview see Bontempi, Taieb and Le Borgne [34]).

Note that in cross-sectional regression, we assume to have observations on multiple i.i.d. instances. However, in forecasting, we only observe a single instance. Even though the transformed data has a tabular format, it is implausible to assume that the rows of the matrix represent independent instances, as they are likely to depend on previous rows. As a consequence, typical model evaluation approaches based on random shuffling of instances, such as using a hold-out test set or resampling strategies such as cross-validation, are likely to lead to over-optimistic performance estimates. So, while we can use cross-sectional regression algorithms to solve forecasting tasks, it is important to evaluate their performance in the context of the original forecasting task and not the reduced regression setting.

Reduction relations have some key properties which we summarize below:

- **Adaptive.** Reductions are adaptive. They adapt an original task to a new task. They enable us to apply an algorithm for the new task to solve the original task. However, note that even though we reframe the task, we still wish solve the original task. Ultimately, we need to be careful to evaluate our algorithm in the context of the original task, not the new task. Performance estimates of the algorithm obtained from the new task, in general, do not hold for the original task. The reason is that key assumptions associated with the new task may not hold in the original task. For example, when reducing a forecasting task to cross-sectional regression using a sliding window transformation, as described in example 2, the transformed data can be represented in a tabular form, but rows do not represent i.i.d. instances. So, while the relational data structure is the same, the assumed data generative setting is not.

- **Modular.** Reductions convert any algorithm for a particular task into an algorithm for a new task. Applying some reduction approach to N base algorithms gives N new algorithms for the new task. Any progress on the base algorithm immediately transfers to the new task, saving both research and software development effort [25, 26].

- **Composable.** Reductions are composable. They can be composed to solve more complicated problems [25, 26]. For example, we can first reduce forecasting to time series regression which we can then reduce to cross-sectional regression as indicated in Figure 3.1 by the top-row arrows (d) and (g).

- **Tunable.** Most reductions require practitioners to make additional modeling choices. For example, when reducing a forecasting task to cross-sectional regression, as described in example 2, practitioners need to select the window length parameter or strategy for generating forecasts – be it the "recursive" strategy or any other [34, 240]. These choices can be understood as hyper-parameters of the reduction approach. In practice, we may wish to optimize, or tune, these choices using appropriate model selection techniques.[4]

We return to these properties in the context of the API design for the unified framework for ML with time series in Section 8.4.2.

3.5 Concluding remarks

In this chapter, we gave an overview of different data generative settings that give rise to time series data. We discussed the key formats such data can take, including univariate, multivariate and panel data. From the overview of the different data settings, we then developed the taxonomy of learning tasks. The taxonomy forms the first research contribution of this thesis as laid out in Section 1.6. We focused on three fundamental tasks, namely time series classification, time series regression and forecasting. While the taxonomy highlighted the differences between these tasks, we discussed reduction as a key concept to understand the relation between tasks. The insights from this chapter will form a central part of our conceptual model for the domain of ML with time series, based on which we will design the unified software framework in Chapter 8.

[4]For an introduction to hyper-parameter optimization and model selection in a setting with i.i.d. data, see e.g. Guyon [99]. For model selection in a temporal data setting, see e.g. Bergmeir and Benítez [23].

Part II

Development of a Unified Framework

Chapter 4

Background: Toolbox Design

This chapter is partly based on:

- Franz J. Király et al. 'Designing Machine Learning Toolboxes: Concepts, Principles and Patterns'. In: *arXiv preprint* (2021)

4.1 Introduction

In the previous part, we have developed a conceptual model for the domain of ML with time series, with the taxonomy of learning tasks at its core. In this part, we address the second set of methodological research questions on the topic of software design for ML toolboxes as discussed in Section 1.5. The answers we develop in this part will form the second research contribution of this thesis as laid out in Section 1.6, namely the development, formalization and motivation of a novel set of general, reusable design principles for ML toolboxes. In the next part, we will apply these design principles to develop a unified framework for ML with time series and its implementation in `sktime`.

This part is concerned with what we have called in Chapter 1 the "developer's problem". While practitioners try to solve a particular applied ML problem at hand (e.g. forecasting the temperature in a chemical process or predicting the type of a patient's disease), the developer's problem is to develop toolboxes that help practitioners solve their respective problems more effectively. Specifically, we are concerned with the software design aspect of the developer's problem. The design of a toolbox crucially determines its effectiveness and applicability. Toolbox design – like any software design – is difficult. One has to identify relevant objects in the domain of interest, find abstractions for them at the right level of granularity, translate them into classes and functions with well-defined interfaces and specify clear hierarchies and relations among them.

A natural source of relevant ideas, useful formalism and best practices is found in the corpus of classical software design. Our approach largely falls under "domain-driven design" as put forward by Evans [78], but also draws on ideas from "design by contract" [188], "responsibility-driven design" [264–266] and "pattern-oriented design" [48, 90]. While much is direct transfer from existing software design, there is a substantial aspect in which ML departs from classical domains: algorithms, interfaces and workflows are closely intertwined with mathematical and statistical theory – to an extent that mathematical objects are not only at the methodological core, but a key element in their representation, workflow specification and user interaction. This situation poses a number of unique research questions which we try to address in the following: How can one identify and describe a "good" design for ML toolboxes? How can one find incisive abstractions in the domain of ML that are both linked to the underlying mathematical and statistical formalism in ML theory and translatable into software? And what generalizable design principles can be derived from existing toolboxes to guide the development of new ones?

Our approach to addressing these questions relies on a combination of formal mathematical statistics, a new type system called "scientific types", novel design principles as well as adapted software design patterns. In particular, we will see that the design of modular and principled ML toolboxes relies on a clear description of underlying learning tasks, algorithms and the relations between them. We believe that in order to advance toolbox design, one first has to advance the formal, mathematical specification of tasks and algorithms.

The rest of this part of the thesis is organized as follows:

- We begin in this chapter to state and further clarify the problem we are trying to solve – what we have called the developer's problem. We then review key ideas from classical software design, including domain-driven design and object-oriented programming (OOP).

- Next, in Chapter 5, we begin to describe our solution to the developer's problem. We do this by developing a conceptual model of ML theory. As is standard in software design, we start by separating out those elements in typical practitioners' workflows that are more likely to change than others. We take these elements as sensible points of abstraction. We then formalize, relate and abstract these elements based on two key aspects: (i) the set of operations that is typically performed on them and (ii) their statistical properties. We express these abstractions through what we call "scientific types" – a new type system that captures the scientific purpose of elements in a precise way that is both usable in ML theory and implementable in software.

- In Chapter 6, we can then derive, from our conceptual model, general design principles and reusable design patterns. The design patterns are inspired by the classical software design patterns put forward by Gamma et al. [90]. As we will see, our principles and patterns enable us to translate the abstractions we found during conceptual analysis into software in a systematic and rigorous manner.

- Before we discuss `sktime`, we review related software in Chapter 7 and discuss the limitations of the current software ecosystem and the rationale for a unified framework.

- Finally, in Chapter 8, we will be able to design the proposed framework for ML with time series and describe its implementation in `sktime`. We do this by combining our insights from previous chapters, including the taxonomy of learning tasks, the conceptual model of ML theory as well as our design principles and patterns.

In the next and final part of the thesis, we will illustrate `stkime`'s effectiveness and applicability by reproducing and extending the M4 competition, one of the major comparative benchmarking studies for forecasting.

4.2 Problem statement

In this section, we describe in more detail what we have called in Chapter 1 the "practitioner's problem" and the "developer's problem".

Practitioner's problem. The practitioner's problem is to solve a certain practical ML problem. For example, forecasting the temperature of a chemical process or predicting the type of a patient's disease. Solutions to a practitioner's problem typically involve specifying a suitable workflow from model specification, training and selection to validation and deployment. To execute such workflows, practitioners write software code in which they usually incorporate functionality from existing software libraries called "toolboxes". Toolboxes provide collections of related and reusable functionality that practitioners can import to write code. Instead of writing every application from scratch, they can simply put together the predefined pieces of code provided by toolboxes. In order to evaluate their solutions, practitioners estimate the predictive accuracy of their models – at least for predictive tasks – but also take into account other criteria such interpretability [95], fairness [187, 191] or computational efficiency. For example, while practitioners try to find the most accurate model, they may also need to ensure that the model's predictions are interpretable in terms of its input variables.

Developer's problem. The developer's problem, on the other hand, is to develop toolboxes. Example include the development of `scikit-learn` [205], a toolbox for cross-sectional ML or the development of `sktime`, the goal of this thesis. Solutions to the developer's problem typically involve identifying key elements in typical practitioners' workflows, find incisive abstractions for them, specify classes and functions with well-defined interfaces, define clear hierarchies and relations among them, and finally implement them in a reusable software library. In order to evaluate their solutions, developers usually consider the following three aspects:[1]

- the *content* (what the library does, e.g. whether it is correct or at least consistent with some theoretical specification),[2]
- the *design* (the language and structure of the library),
- the *performance* (how computationally efficient the library is in terms of run time and memory usage).

For example, developers want to test the library to ensure its consistency with some specification, make its syntax readable and write code that is efficient enough to be practically useful. Note that the evaluation aspects are not independent of each other. For example, more readable code may be easier to test but less efficient. In general, there will be no "pareto optimum" and trade-offs have to be made.

In this part of the thesis, we focus on the developer's problem, particularly on the design aspect. While the content and performance aspects are taken into account throughout the thesis, we only discuss them more explicitly in the context of `sktime` in Chapter 8. In the following, we make a few additional remarks to further clarify and motivate the design aspect of the developer's problem.

4.2.1 Design as a result

Design often refers to both the design *process* and the *result* of that process [45]. Viewed as a process, software design is about the software engineering life cycle activity in which software requirements are analyzed and translated into a software

[1]We leave aside hardware or compiler related aspects such as code portability as this is usually ensured on the level of the programming language.

[2]Correctness in software is always a relative notion. No software element is correct or incorrect per se. The only practically useful notion is that of consistency with some specification. Consistency can, at least partly, be ensured through testing.

plan that will serve as the basis of its implementation. Examples of design process methodologies include the "waterfall model" [220] or "agile development" [20, 58, 108].[3] Viewed as a result, design is about the overall architecture of software, its organization into modules, and the specific structure of key components within modules, their interfaces and relations to other components.

This thesis is mainly concerned with design as a result and not tied to any particular process methodology. We briefly comment on the design process in the context of `sktime` in Chapter 8.

4.2.2 Applications, toolboxes and frameworks

It will be helpful to distinguish three forms of code [90]:

- "Applications" are the *single-purpose* code that practitioners write to solve a certain applied problem.
- "Toolboxes" are collections of reusable and related functionality that practitioners can import to write applications. As such, toolboxes emphasize *code reuse*. Note that toolbox design is arguably harder than application design. This is because toolboxes have to work in many applications to be useful, and toolbox developers are often not in a position to know exactly what those applications or their special requirements will be.
- "Frameworks" not only offer reusable functionality, but also provide overall structure to application code. Frameworks capture common design decisions and distill them into reusable design templates. These templates may specify common ways of partitioning workflows into key components or define common blueprints and interfaces for those components. Instead of designing every application from scratch, practitioners can simply copy templates, thereby reducing the design decision they must take and allowing

[3]Note that the waterfall model and agile development are more general software development management methodologies that also apply to other parts of software development beyond design.

Example 1: `scikit-learn`'s cross-sectional classification API

```
classifier = RandomForestClassifier()
classifier.fit(y_train, X_train)
y_pred = classifier.predict(X_test)
```

`X_train` and `y_train` denote the training set of the cross-sectional feature data and label vector, `X_test` is the test set of the feature data, and `y_pred` the predicted labels.

> them to concentrate on the application specifics. Not only can practitioners write software faster as a result, but applications will have a similar structure. They will be more consistent, more reusable and easier to maintain. While toolboxes emphasize code reuse, frameworks emphasize *design reuse*, even though frameworks usually also provide reusable code that practitioners can put to work immediately. As such, frameworks are the way that software achieves the most reuse. Note that if applications are hard to design, and toolboxes are harder, then framework design is arguably the hardest. Framework developers have to find one design that is abstract enough to be practical for a wide range of problems, while at the same time specific enough to capture important details about these problems.

One of the main goals of this thesis is to develop a framework for ML with time series, where framework is understood in the above sense.

4.2.3 Toolbox design as a research problem

The developer's problem may seem trivial at first sight or perhaps not even worth scientific investigation. After all, popular toolboxes have been so widely adopted that it is hard to imagine what alternative designs might look like or what challenges had to be overcome in their development. However, when taking a closer look, it becomes clear that many questions remain unanswered. These questions often become clearest when one tries to explain certain design decisions.

Consider the design of the cross-sectional classification API in `scikit-learn` [205] shown in example 1. We may ask: Why is "classifier" as in `RandomForest-Classifier` a meaningful algorithm type? Why are the `fit` and `predict` methods its main interface points? What other key interface points are there? How is a classifier related to other algorithm types? And how can they interact? Besides these practical questions, there are also deeper methodological ones: Can we generalize `scikit-learn`'s API design to other algorithm types beyond the cross-sectional setting? How do we derive and motivate API design in the ML domain? And how do we make sure that our abstractions remain closely linked to the underlying ML theory, but are also easily translatable into software?

We believe that there exist principles that can answer these questions and that these principles can be discovered through analysis and research. Finding these principles is one of the main goals of this thesis. Our aim is to identify these principles and describe them in a way that developers can use them to create new toolboxes or improve existing ones, without having to rediscover these principles.

4.2.4 Evaluating design

It is difficult to define what exactly constitutes a "good" design.[4] We believe that toolbox design – much like any software design – remains partly at least an art,[5] however that there do exist some design quality criteria that can guide and assess design solutions. We will describe what we believe to be key quality criteria for ML toolbox design in Section 4.3 below.

Throughout this thesis, we will make qualitative arguments to support our design choices,, drawing on the similarity to well established APIs, notably `scikit-learn` [46], as well as the adherence to domain-driven design principles [78], common design patterns [90, 152] and other best practices in data science [136, 164, 261, 262] and classical software design [18, 214].

[4]For an overview of different approaches to assessing software design, see e.g. Budgen [45].

[5]Similarly, software design is sometimes characterized as a "wicked problem" with no definite solution [67]. For more details, see Budgen [45].

4.2.5 Related literature

The importance and challenges of toolbox design has been recognized in the data science literature. However, to our knowledge, discussions on design are still rare.

A part of the literature has focused on promoting open-source development [156, 238] and best practices for scientific computing, with the aim to make research more reliable and reproducible [136, 164, 261, 262]. However, discussions on the central role of ML toolboxes for data science practice are still largely missing.

Another part of the literature has concentrated on the "what" of design, i.e. software features and usage, while almost completely ignoring the "why" (see e.g. Buitinck et al. [46]). For example, developers often subscribe to a set of design principles when presenting their work. However, these principles typically remain too vague to explain concrete design decisions. Of the few discussion on design principles, most also concentrate on a single toolbox without attempting to generalize design ideas.[6] While the actual software tends to contain a wealth of design ideas, we are not aware of literature in which underlying design principles are studied. We believe that analyzing the "why" is crucial for generalizing successful designs from existing toolboxes to new ones like `sktime`.

The software design literature, on the other hand, has focused on the design of ML systems as a whole [196, 197, 226, 259], but paid little attention to the toolbox layer within these systems, even though toolboxes form a central part of those systems implementing core algorithmic and mathematical functionality. In fact, authors often point out that there is a lack of strong abstractions to support ML software, hindering much needed progress towards a more declarative SQL-like language for ML [226, 272].

We attempt to address these issues in this part of the thesis by deriving general design principles for ML toolboxes. Rather than describing a single toolbox, or analyzing ML software at a system level, we focus on general principles for ML

[6]A few exceptions include for example Howard and Gugger [117] on the design of `fastai` and parts in LeCun et al. [155] on the design of deep learning frameworks.

toolbox design. These principles, as we will see, cannot just explain key aspect of existing toolboxes, but also guide the design of new ones, including the proposed framework for ML with time series.

4.3 Toolbox design quality

In this section, we will describe what we consider to be key quality criteria for ML toolbox design. These quality criteria are based on the role of toolboxes in contemporary data science.

4.3.1 The role of toolboxes in data science

Toolboxes have become workhorses of modern data science and central components in scientific, commercial and industrial applications. We believe that toolboxes should be designed in a way that facilitates and promotes best practices in the key activities that data science practitioners and developers have to perform [136, 164, 261, 262].

For practitioners, this includes the following activities:

- *Writing and changing of workflows*, i.e. enabling practitioners to rapidly prototype and execute typical workflows from model specification to deployment, with a small amount of easy-to-write code, and to change key components, without having to change substantial parts of the workflow as a whole.

- *Reading and inspection of workflows*, i.e. enabling practitioners to easily understand a workflow's underlying mathematical, statistical and algorithmic operations, inspect key workflow components and their intermediate outputs, and check the scientific validity of workflows, reproduce their results, and compare them against alternative workflows.

For developers, this includes the following activities:

- *Writing and changing of tools*, i.e. enabling developers to implement new methods or enhance existing ones and make these tools easily accessible and interoperable as part of a wider ecosystem.
- *Maintenance and testing of tools*, i.e. enabling developers to refactor and improve code when necessary and easily test functionality.

4.3.2 Design quality criteria

From the above we can derive a set of quality criteria for toolbox design. In particular, we believe that these activities will be facilitated by toolboxes that exhibit the following design quality criteria:

- **Code reusability.** The code provided by toolboxes should be highly reusable across a wide range of practitioners' problems.
- **Design reusability.** The design should be highly reusable across a wide range of problems and enable practitioners to follow common workflow templates in their applications and developers to follow common templates when developing new methods.
- **Domain-driven design.** The structure and language of the framework should be closely aligned with the mathematical and statistical concepts of ML theory. The software will be more in tune with what practitioners are already familiar with as a result, thereby reducing their cognitive load and making it easier for practitioners and developers to communicate. We discuss domain-driven design in more detail in Appendix B.
- **Modularity.** Toolboxes should provide functionality that breaks down complex workflows into modular components that can be recombined and interchanged. As a result, practitioners will be able to substitute components without having to change other parts of the workflow and developers will be able to refactor code without affecting code in other parts of the library.

- **Parsimony.** Toolboxes should avoid unnecessary conceptualization, formalization and semantics.
- **Interoperability.** Toolboxes should be interoperable with other libraries in the same domain. Practitioners will be able to use methods from different libraries in a single workflow and combine advances from different areas.

Note that these criteria are already met, to varying degrees, by existing toolboxes. We will review existing toolboxes in Chapter 7.

Having described the developer's problem and key quality criteria of ML toolbox design, we are now in a position to start describing ways how to solve the problem.

4.4 Concluding remarks

After having developed the taxonomy of time series learning tasks in the previous chapter, we have begun to address the developer's problem in this chapter. In particular, we have further clarified and motivated the developer's problem, with a focus on software design. Based on this background, we are now in a position to start developing our solution to the design aspect of developer's problem in the next chapter.

Chapter 5

Scientific Types: A Conceptual Model for ML Theory

This chapter is partly based on:

- Franz J. Király et al. 'Designing Machine Learning Toolboxes: Concepts, Principles and Patterns'. In: *arXiv preprint* (2021)

5.1 Introduction

In the previous chapter, we stated the problem that we are trying to solve – what we have called the developer's problem – and reviewed key concepts from classical software design and object-oriented programming. In this chapter, we start developing our solution to the problem. The solution we present will form the basis of our second research contribution of this thesis as laid out in Section 1.6.

As we have seen, the developer's problem is about translating elements from ML theory to software. In line with domain-driven design [78], development of a conceptual model is the initial step in which relevant concepts are identified, abstracted and related. The software is then designed to closely match the conceptual model.

We have begun to develop our conceptual model of ML theory, particularly for ML with time series, by developing the taxonomy of learning tasks in Chapter 3. In this chapter, we will complete the conceptual model by analyzing relevant elements in ML theory more generally. We do this through conceptual analysis: we first identify, formalize and abstract common elements found in ML theory. We then express the abstraction we found in a new type system, which we call "scientific types", or "scitypes" for short. In this conceptualization, two realms are inextricably linked:

- The realm of software: programming languages, object-oriented programming and machine types.
- The realm of theory: mathematical objects, functions, signatures, domains and formal types.

We therefore try to ensure that both realms are taken into account simultaneously.

As we will see, scientific type are defined as a combination of structured data types and mathematical properties that all elements of that type must satisfy. Scientific types offer a rigorous and systematic way to translate elements between the two realms of ML theory and software. For example, we might say that a "pipeline" consists of a component of "feature extractor" type and a component which is of "supervised learner" type, put together in a certain form. We believe that the types in question can be made precise so that they can be leveraged for design considerations, rather than just being vague metaphors without tangible content. Scientific types enable us to formalize the conceptual abstractions in question, they make them both mathematically precise and easily translatable into software. Based on scientific typing, we will propose design principles and design patterns in the next chapter.

Much of our conceptual analysis can already be found in some form in ML literature, often in the framing of ML theory or in the documentation and discussions surrounding ML software. However, to our knowledge, the ideas we are presenting here have neither been put together as a coherent conceptual model, nor have they been framed as the underpinnings of ML toolbox design. A first draft of these ideas appeared in Király et al. [145]. We here refine some points, add details and align the terminology and overall structure with the rest of this thesis.

We subscribe to object-oriented programming (OOP). OOP is the predominant paradigm for ML toolboxes and thus often assumed without rationale. We give key reasons why object-oriented programming is especially well suited for building ML software in Section 5.5.

The rest of this chapter is organized as follows: In Section 5.2, we start developing our conceptual model for key objects in ML theory. We will then introduce our novel type system called "scientific types" in Section 5.3 and apply it to key ML objects in Section 5.4. Finally, we will make a few remarks about the implementation of scitypes in Section 5.5, including the rationale for using OOP. In the next chapter, we will derive key design principles and patterns from our conceptual model, and in particular scitypes.

We review key ideas from classical software design, including abstraction and domain-driven design, as well as object-oriented programming in Appendix B and A.

5.2 A conceptual model for ML theory

ML theory uses mathematical and statistical formalism to describe, analyze and relate common concepts in ML. In analogy to OOP, one can understand ML theory as a system made up of mathematical objects. As is standard, we start by identifying and separating out the more change-prone objects as sensible points for abstraction. These are:

- Problems or tasks that we want to solve (e.g. regression or classification),
- Algorithms that can solve these tasks (e.g. linear regression or a decision tree),

- Related mathematical objects, on the conceptual level, such as loss functions or distributions.

We have already discussed learning tasks in Chapter 3, both as a key concepts in ML theory and in terms of specific examples for cross-sectional and time series learning problems. We therefore focus on algorithms and related mathematical objects that are relevant for ML toolbox design for the rest of this chapter.

5.2.1 Examples of mathematical objects

There are a number of mathematical objects that appear repeatedly as variable parts in ML theory. Examples of recurring objects include:

- Data frames with definable and inspectable column types, e.g. numerical, categorical, or complex (in a formal mathematical sense),
- Non-primitive and composite data objects such as series, bags, shapes, or spatial data,
- Sets and domains, e.g. for specifying ranges, intervals, parameter domains,
- Probability distributions, e.g. arising as return types in Bayesian inference or probabilistic prediction,
- ML algorithms (sometimes also called "estimators", "strategies", "models" or "learning machines"),
- ML pipelines, neural networks, composites or entire workflows,
- Other related objects for ML workflows involving model tuning or model evaluation.

5.2.2 The common interface of mathematical objects

In ML toolboxes, mathematical objects, such as the examples above, appear both as at the methodological core (e.g. in the internal representation of methods) as well as key elements of user interaction and workflow specification (e.g. in interacting with data, specifying models or inspecting fitted models).

In these situations, one can identify a set of common interaction points in dealing with mathematical objects. We call these interaction points the common "interface" of mathematical objects, parallel to the notion of an interface in OOP. The common interaction points include:

- *Definition.* Creating variables that are mathematical objects, for example: "let X be a normally distributed random variable with mean 0 and variance 42, or "let `myseries` be a time series" for a given list of values and time points.
- *Parameter inspection.* "What is the mean of X?" or "what is the value of `myseries`?" at a given time point,
- *Type inspection.* Querying the mathematical type (e.g. domain) of an object: "what domain does X take values in?" or "how long is `myseries`?"
- *Trait and property inspection.* "Is the distribution of X symmetric?" or "are the time stamps of `myseries` equally spaced?"

5.2.3 The common attributes of mathematical objects

Based on the common interface, we postulate the following common attributes of mathematical objects:

(a) a *symbol* or *name*, e.g. X or `myseries`,

(b) a *type*, e.g. a "real random variable" or "real univariate time series",

(c) a *domain*, e.g. "the set $\mathbb{R}$" or "absolutely continuous distributions over $[0, 1]$",

(d) a collection of *parameters* on which the object may depend, being mathematical objects themselves,[1]

[1]Of course, there may not be infinite recursion. Therefore, eventually there are nested parameters without parameters.

(e) a collection of *properties and traits*, e.g. “this random variable has a finite domain” or “this time series is univariate”.

Here, the domain could be the universal domain and the collections of parameters, properties and traits could be empty collections. Based on the above description of the common interaction cases, we postulate these attributes as attributes intrinsic to any mathematical object.

As an example of a mathematical object and its attributes, as defined above, consider the common definition: “Let X be a random variable, distributed according to $\mathcal{N}(0, 42)$”. Here, one could define (a) the symbol being X, (b) the type being “real random variable”, (c) the domain being “absolutely continuous over $\mathbb{R}$“, (d) the parameters being $\mu := 0$, $\sigma^2 := 42$, with types of μ, σ being numeric or continuous and domains $\mu \in \mathbb{R}$, $\sigma^2 \in \mathbb{R}^+$, and (e) traits as X being a symmetric distribution.

Note an important subtlety here: defining a mathematical object with variable free parameters is not the same as defining a mathematical object with variable but set parameters, e.g. a generic $\mathcal{N}(\mu, \sigma^2)$ in contrast to a concrete $\mathcal{N}(0, 42)$. As we will see below, this duality is easily resolved in, and partly motivates the object/class distinction in the object-oriented paradigm, as discussed in Appendix A.

How are mathematical objects, as defined above, related to software? Attributes (a) and (b) are commonly found as defining attributes of any variable in common programming languages (e.g. in Java, Python or R). They can be regarded as standard, even without our conceptualization of mathematical objects introduced here. On the other hand, attributes (c), (d) and (e) are typically not found in programming languages, at least not as features of the base language, and therefore need dedicated implementation. We discuss implementation details later in Section 5.5 and in the context of design patterns in Chapter 6.

5.2.4 Statefulness: Value and entity objects

An important distinction that divides common objects in ML theory into two categories is that of *statefulness*:[2]

- Immutable *value objects* having the same properties independent of the condition or context,
- Mutable *entity objects* being subject to changes of properties, with a continuous identity that needs to be maintained.

The typical mathematical object, as introduced above, is understood as a value object – an object that, once defined in common mathematical formalism (e.g. “let $x := 42$”), remains unchanged within the scope of its existence. State changes are not properties one would commonly associate with mathematical objects. We, therefore, conceive of mathematical objects as value objects.

Note that because of this, state changes were not part of the common interface for mathematical objects defined above. Value objects, by definition, characterize objects without such interface points. Overall, value objects are also those for which there is unambiguous convention on mathematical notation in the ML literature.

On the other hand, ML objects, such as ML algorithms, are conceived of as entity objects. We argue that ML objects are primarily defined by their identity, rather than their attributes. They are typically characterized by state-changing operations, for example ingestion of training data and fitting of model parameters changing the state of an algorithm object from “unfitted” to “fitted”.

ML objects often possess multiple internal states between which they can transition. States often span several properties and states are often accessed and modified through several interaction points (e.g. data ingestion and fitting). Changes to those properties must be coordinated in order to maintain the consistency of the object. In addition, a threat of continuity needs to be maintained

[2]For a discussion on value and entity objects and how to decide between values and entities in design, see e.g. Evans [78] and Vernon [257].

between the estimation algorithm and the output of that algorithm, i.e. the fitted prediction functional. Otherwise, one may end up with a prediction functional that cannot be traced back to the algorithm and hyper-parameter configuration that was used for fitting it. Construing learning algorithms as entity objects helps both to manage state changes and to maintain continuity between the fitted prediction functional and the learning algorithm used to fit it.

In our conceptual model, we will therefore distinguish between "mathematical objects", that model formal mathematical objects in the classical sense and are value objects, and "ML objects" that model ML algorithms and are entity objects.[3] In the list of common examples above, the following are mathematical objects in this sense:

- Data frames with definable and inspectable column types, e.g. numerical, categorical, or complex,
- Non-primitive and composite data objects such as series, bags, shapes, or spatial data,
- Sets and domains,
- Probability distributions.

Note that we consider data frames and data objects as value objects. State changes are not usually a defining characteristic of data, at least not in the context of ML modeling. State changes to data are conceptualized as part of model specification, rather than the data object itself.[4] For example, we may specify a pipeline with a prior data transformation and a final learning algorithm.

On the other hand, the following are ML objects, and thus entity objects:

- ML algorithms,
- ML pipelines, networks, composites, or entire workflows,

[3]An ambiguous case, to which we will return in Section 6.3.3, is the nature of composition meta-algorithms, which are themselves value objects, but which specify entity objects.

[4]We leave aside special cases such as streaming data in which state changes may well be a defining characteristic of the data itself.

- Workflow elements such as for automated tuning or model performance evaluation.

All of these objects are characterized by operations which change states but with a continuous identity.

Note that in comparison to value objects, the ML literature disagrees more widely on conceptualization of ML entity objects. For example, an algorithm may be represented by a prediction functional on test data or a functional in which both training and test data have to be substituted, thus representing the fitting process by some of its argument.[5] Besides the lack of consistency, we argue that there is also a lack of expressivity which prevents clear conceptualization on both the theory and software level. We will attempt to partly remedy this situation by introducing scitypes in the next section.

5.3 Scientific types

So far our conceptual analysis has revealed that ML theory is made up of objects – objects which in analogy to OOP are characterized by having common attributes, interfaces and, in some cases, state changes. Our aim now is to further abstract these objects. We want to find incisive categories that capture their data scientific purpose and express them in way that is translatable into software.

We argue that a conceptual model for this purpose is naturally rooted in a type system. The reason is that type systems offer exactly what we need: a type makes explicit the category of objects in a precise way that is usable in mathematical statements but also implementable in software.

The challenge is then to identify the "right" type system. Type systems are both at the foundation of modern programming languages and the mathematical formalism that underlies much of ML theory. We argue that these type systems are not sufficient on their own to inform ML toolbox design. We therefore propose

[5]For example, compare the framing in Hastie, Tibshirani and Friedman [105, chapter 2] and Bishop [29, chapter 1].

a new type system called “scientific types”, which combines the type systems found in ML theory and software. As we will see, a scientific type, or “scitpye” for short, enables us to abstract mathematical objects, based on their interface and key statistical properties, in a way that is both mathematically precise, but also easily translatable into software.

To introduce this idea, we extend classical mathematical formalism to stateful, structured objects, along the lines of type systems found in programming languages [210]. We do this by making use of common formalism in type theory, extended with some novel aspects.

Before we introduce scitypes, we briefly motivate why the type systems found in mathematical formalism and programming languages are not sufficient on their own.

5.3.1 Limitations of existing type systems

Programming languages, particularly object-oriented ones, allow to create objects with complex internal representations, involving multiple properties and methods. However, while programming languages allow to construct almost arbitrary user-defined types, their types are ultimately defined in terms of a widely used set of primitive “machine types” (e.g. floats, strings or integers).[6] From a data scientific perspective, machine types are not enough to describe objects. They fail to capture key statistical properties which are crucial for our understanding of ML objects.

A simple example is that in a programming language, objects of $\mathbb{R}, \mathbb{R}^+$ or $[0, 1]$ may all be represented by floats. On the other hand, any collection of floats may be continuous in $\mathbb{R}$ or categorical in a predefined finite subset $\mathcal{C} \subseteq \mathbb{R}$, depending on knowledge or assumptions about the data generating process. This is not inferrable from the machine type alone.

Another example is the difference between cross-sectional learning algorithms

[6]Machine types are sometimes also called “data types”. They are called “machine types” because they are ultimately defined at the compiler or hardware level [142].

and their time series counterparts, as discussed in Section 3.3.1. Whereas cross-sectional algorithms make no assumption about the orderedness of feature data, their time series counterparts do. In other words, shuffling the order of the feature data makes a difference to time series algorithms, but not to cross-sectional algorithms. Again, this is not be inferrable from the machine type of an algorithm by itself.

On the other hand, mathematical types usually only extend to objects of a relatively simple structure. As we have seen in Section 5.2, mathematical objects are usually conceptualized as immutable value objects, without the capacity to maintain a continuous identity through state changes. As such, mathematical types are not enough either to express the more complex ML objects.

5.3.2 What is a type?

Having described the limitations of type systems in mathematical formalism and programming languages, we will now attempt to combine the two into what we call scientific types. We begin by defining what a type is.

The general concept of a "type" arises both in mathematical formalism, as well as in modern programming languages [210]. In both situations, any object may have one or multiple of the following:

(a) a *symbol* or *name*, e.g. x,

(b) a *type*, e.g. an "integer",

(c) a *value*, e.g. 42.

There is some ambiguity around the definition of "type" [142, 202]. For our purposes, it is enough to understand a type as being defined in terms of:

- a *named interpretation*, a meaningful name assigned to the type (e.g. "integers")
- a *representation*, a domain or value space into which the value of objects of the type must fall (e.g. the set of integers $\mathbb{Z}$),

- an *interface*, a set of operations that are well defined and associated with the representation (e.g. addition for integers).[7]

Assigning a type to a symbol or value is a purely formal act and in-principle arbitrary. It becomes practically useful through typing rules. Most importantly, any type has "inhabitants", that is, values that are admissible for a type. For example, the formal type "integer" is inhabited by the numbers 0, 1, -1, 2, -2, and so on. As such, the representation aspect of mathematical types can often be identified with their inhabitant sets.

In mathematical notation, the typing colon is used to denote an object's type. For example, the formal statement x : `int` states that the object denoted by symbol x is of type `int`. The set membership operator "$\in$" is also frequently used to denote an object's type, e.g. $x \in \mathbb{Z}$, but we will here prefer the typing colon, in line with common notation in programming languages.[8] Programmers will be familiar with the notation in their favored language, for example, in Java the statement would be `int x`.[9]

5.3.3 Composite types

So far we have mentioned examples of primitive types (e.g. integers, floats or strings). From primitive types, type operators allow construction of more complex composite types. Two important type operators are the conjunction,[10] which

[7]Kell [142] calls these aspects "named interpretation", "storage contracts" and "operational well-definedness over storage", and adds one more aspect, namely "semantic well-formedness", which is not directly relevant for our purposes.

[8]Readers with a mathematical background may also note the statement $x : \mathbb{Z}$ ("x is of type $\mathbb{Z}$") is different in quality from saying $x \in \mathbb{Z}$ ("x is an element of the set $\mathbb{Z}$"), even if, as a statement, the one is true whenever the other is. The difference lies in that the first is interpreted as an *attribute* of the *object* represented by x ("the object represented by the symbol x has integer type"), the second as a *property* of the *value* of x ("the value of the object represented by x is an integer"). While this may not be much of a difference on the conceptual level in mathematics, where symbols only serve as mental reference, there is a difference on the implementation level in a computer, where the symbol may be attached to an object or in-memory representation which implements the conceptual model. On the formal side, this interpretation is also reflected through differences in axiomatization of type systems, versus axiomatization of set theory.

[9]Even in weakly typed, duck typed, or untyped languages, there are usually mechanisms at the compiler or interpreter level that correspond to formal typing, even if they are not exposed to the user directly.

[10]One also finds the symbol "$\sqcap$" in literature.

can be identified with the Cartesian product "$\times$" for sets, and the arrow "$\rightarrow$", which allows to construct function types mapping inputs of one type to outputs of another type. Examples include collection types for tuples, lists, sets, dictionaries, vectors or matrices or function types such as $A \rightarrow B$ denoting the family of functions that map elements of type A to elements of type B. Table 5.1 gives more examples, highlighting the similarity between typing in software programming and mathematical formalism.

Table 5.1: Some examples of type notation in programming and mathematical type notation

Description	Programming language notation (pseudo-code)	Mathematical notation
x is an integer	`int x`	$x : \mathbb{Z}$
x is the integer 42	`int x = 42`	$x = 42 : \mathbb{Z}$
f is a function which takes as input an integer and outputs an integer	`func int f(int)`	$f : \mathbb{Z} \rightarrow \mathbb{Z}$
f is the function which takes as input an integer and outputs its square	`func nat f(int x) return x*x`	$f : \mathbb{Z} \rightarrow \mathbb{N};\ x \mapsto x^2$
x is a pair made of: an integer (1st), and a real number (2nd)	`[int,real] x`	$x : \mathbb{Z} \times \mathbb{R}$
f is the function which takes as input two integers and outputs their sum	`func int f(int x, int y) return x+y`	$f : \mathbb{Z} \times \mathbb{Z} \rightarrow \mathbb{Z};\ (x, y) \mapsto x + y$

Notes: We express the example statements in vernacular description (first column), pseudo-code of a stylized typed programming language (second column), and formal mathematical type notation with inhabitant-set identification convention (third column). Mathematical type operators used in the third column are the conjunction type operator, denoted as $\times$, and the arrow (type construction or function) operator, denoted as $\rightarrow$. A reader may recognize the colon commonly used for range/domain specification in mathematical function definition as the typing colon, and the arrow in function definition as the arrow operator (such as in third/fourth row, third column); whereas the typing colon for domain statement of a variable (such as in first row, third column) would be unusual in contemporary usage, outside of formal type theory.

Another important composite type is that of a "structured type", sometimes also called "structured data type", denoting the type of a structured object as commonly found in OOP. Concretely, a structured type is a container with named types in "slots", formally called "component types", which correspond to an object's attributes or methods. The inhabitants of a structured type are structured objects, where every slot is inhabited by a concrete value inhabiting the slot type.

Note that composite types are defined recursively in terms of their component types. For example, the type of a set is defined in terms of the types of its set members. The type of a function is defined by its signature, which in turn is defined by the types of its input and output arguments.[11]

5.3.4 Notation

To introduce the notion of a scientific type, we need some notation. While a more formal discussion of types, including structured types, can be found in Pierce and Benjamin [210], there are no notational conventions. We therefore proceed with the following ad-hoc conventions, inspired by UML class diagrams [152].

- Structured types are denoted by a Python-like `type` header giving a named interpretation for the structured type, followed by the slots in the composite. Slots are symbol/type pairs, separated by a typing colon.
- The slots are categorized under one of three headers: `params` for "hyper-parameter" slots that are immutable per inhabitant, `state` for state variables that are mutable per inhabitant, and `methods` which are of function type and correspond to object methods in OOP.[12]
- When specifying a type, a name of a symbol in scope can be used instead of its type. If a symbol in `state` is used in this way in a function in `method`, the function is interpreted to write to, or read from the value of that state.

[11]Of course, there may not be infinite recursion. In mathematical formalism, types are therefore ultimately defined in terms of their inhabitant sets (extensional definition). In programming languages, they are defined in terms of "machine types" at the compiler or hardware level [142].

[12]A "hyper-parameter" is a parameter to configure or control the learning process, in contrast to a "parameter" which is estimated or derived from the data during the learning process. We follow `scikit-learn` in using `params` to refer to hyper-parameters.

- Method implementations can, but need not depend on hyper-parameters in `params` as function arguments.
- If `x` is a structured type, with a slot of name `y`, the slot can be directly referred to by the symbol `x.y`, i.e. the symbol for the structured type, followed by a dot, followed by the symbol for the slot name.
- Inhabitants of a structured type are defined by values for `params`, `state` and `methods`. Inhabitants of a structured type can be both classes and objects, where an object is an instance of a class, as commonly defined in OOP. Classes and objects follow the same notational conventions as structured types with the follow exceptions: A class is denoted by a `class` header, an object by an `object` header. While the `state` is made explicit for classes, it is omitted for objects, as it is changed by objects. In objects, if a specific value is assigned to a slot, rather than a type, we use the equality sign instead of the typing colon. The object name may be omitted.

Following this notation, we can write a template for a structured type as:

```
type <name>
  params   <name>  :  <type>
  state    <name>  :  <type>
  methods  <name>  :  <type>
```

where `<name>` and `<type>` are placeholders to be made precise for any particular type. There may of course be more than one slot under each header. The `<name>` placeholder on the first line denotes the named interpretation of the type.

5.3.5 Example: Gaussian distribution

Given the above notation, we will now introduce scitypes with a simple example. In next section, we will provide a more formal definition.

The following structured type `GaussianDistribution` represents the input/output signature type of a Gaussian distribution object, defined by the cumulative distribution function (cdf) and probability density function (pdf):

```
type GaussianDistribution
  params    $\mu$     :  $\mathbb{R}$
            $\sigma$  :  $\mathbb{R}^+$
  methods   cdf       :  $\mathbb{R} \to [0,1]$
            pdf       :  $\mathbb{R} \to [0,\infty)$
```

An inhabitant of this type would be the structured object given by:

```
object
  params    $\mu$     =  $1$
            $\sigma$  =  $4$
  methods   cdf       =  $x \mapsto \Phi\left(\frac{x-\mu}{\sigma}\right)$
            pdf       =  $x \mapsto \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right)$
```

where Φ is the standard normal cdf. As a typing statement, we can write a : `GaussianDistribution`. To refer to the cdf, we can write a.`cdf` which is identical to the function $x \mapsto \Phi\left(\frac{x-1}{4}\right)$.

Note that the type `GaussianDistribution`, as defined above, does not fully capture what a Gaussian distribution object is. In terms of software, the implementation is missing. In terms of theory, the mathematical specification is missing. For example, the structured object:

```
object
  params    $\mu$     =  $1$
            $\sigma$  =  $4$
  methods   cdf       =  $x \mapsto 2\mu$
            pdf       =  $x \mapsto 42$
```

also inhabits the structured type `GaussianDistribution`.

To fix this issue, one can now define the "scientific type" of a Gaussian distribution object by adding the missing information into the definition. For example, by defining the following: "An object d has scientific type `GaussianDistribution` if it has structured type:

```
type GaussianDistribution
  params    $\mu$     :  $\mathbb{R}$
            $\sigma$  :  $\mathbb{R}^+$
  methods   cdf       :  $\mathbb{R} \to [0,1]$
            pdf       :  $\mathbb{R} \to [0,\infty)$
```

and if the methods have the following form: $d.\texttt{pdf}(x) = \Phi\left(\frac{x-d.\mu}{d.\sigma}\right)$, and $d.\texttt{cdf}(x) = \frac{1}{d.\sigma\sqrt{2\pi}} \exp\left(-\frac{1}{2}\left(\frac{x-d.\mu}{d.\sigma}\right)^2\right)$." While it is somewhat cumbersome in this case, we have defined, a scientific type `GaussianDistribution`. In particular, any object with the scientific type `GaussianDistribution` will, by definition, satisfy the compatibility between parameters and methods expected intuitively from a Gaussian distribution.

5.3.6 What is a scientific type?

We can now give a definition of a scientific type. We define a scientific type as a combination of:

- a *structured type* with slots made explicit, as in our notation introduced above,
- a set of *mathematical and statistical properties* that inhabitants of the type must satisfy.

The structured type does the work of a type as commonly used in programming languages or mathematical formalism: it defines the *representation*, or value space that all inhabitants must fall into, the *interface* that inhabitants must satisfy and gives a *named interpretation.* A scientific type, or "scitype" for short, extends that notion by including a set of mathematical or statistical properties that inhabitants must satisfy.

The named interpretation of a scitype, as we will see below, assigns a meaning to inhabitants based on their data scientific category or purpose, for example denoting a class as a "forecaster" or "cross-sectional regressor" indicates the data scientific purpose of those classes, namely algorithms for forecasting or cross-sectional regression, respectively.

Scitypes satisfy the postulates from our conceptual model of mathematical objects in the following way:

(a,b) name and type statement are intrinsic to the definition of the structured type,

(c) domains can be specified as inhabitant sets,

(d) hyper-parameter statements are intrinsic to our extended notation/definition of the structured type, in the `params` block,

(e) traits and properties are encoded by the slots in the `methods` block.

We make a few additional remarks:

- While the initial definition of a scitype may be a bit cumbersome, later reference to the type or an arbitrary inhabitant can be invoked with a few symbols, e.g. "let d : `GaussianDistribution`", and methods or parameters may also be invoked without prior appearance in the binding statement, e.g. $d.\mu$.

- The conceptualization resolves some items of common ambiguity of mathematical language. For example, which symbols are to be considered parameters? The resolution is: parameters are explicitly designated as a matter of definition. Or more subtle ambiguities around concept identity such as of a "distribution" which, across literature, may be defined or denoted in terms of different methods or properties (e.g. distribution defining functions, measures, maps, etc), depending on context. The resolution is: the object need not be identical with any of its methods, instead it has the methods as attributes.

- Subject to some formalization burden, the concept of scientific types can be fully formalized, along the lines of classical type theory [210]. This amounts to defining inhabitance by satisfaction of a definition property, and should be intuitively clear from the example. For readability, we refrain from adding further formal overhead.

- Note that the structured type notation are simplified representations of the scitype-defining interface. It is intended to capture the defining aspects of scitype interface and highlight differences between different scitypes. It necessarily abstracts away many details. For example, it ignores common methods for object initialization and inspection, among others. We give a more complete list of key interface points in Section 6.3.1. Also note that hyper-parameters in `params` may not only influence the model in `state`, but also change the behavior of methods. Similarly, the state of objects may in practice be represented through a number of fields with potentially different types, rather than a single `state` slot.

Scitypes for mathematical object may not be directly necessary or fully novel in isolation. However, in the next section, we will extend scitypes from mathematical objects to ML objects. Before we move on, we point out some important conceptual distinctions related to the concept of scitypes:

- A class in the OOP sense usually already contains implementations for some methods, whereas the structured type does not specify an implementation for the method. A structured type only records its input-output type and state access signature.
- Type systems usually present in programming languages (e.g. float, string, integers), called "machine types", are not the same as scitypes, nor are they sufficient to inform ML design. As argued above, the reason is that types in programming languages do not capture statistical properties which are crucial to identify and abstract elements in the context of ML modeling. Rather, scitypes are separate or orthogonal to the type systems typically found in programming languages.
- A scitype is not "meta-data", i.e. data about objects but rather a property of the object.

5.4 Scientific types for ML objects

Having defined scitypes, we will now discuss examples of a scitype for stateful ML objects. We will start with a simple example and then discuss ML algorithms as the central entity objects in ML theory.

5.4.1 A simple example

A simple example of an entity object is the structured type of the class `Power-Computer`, which computes the n-th power of a number in one of its slots, given by:

```
type PowerComputer
  params   n               :  ℕ⁺
  state    x               :  ℝ
  methods  store_number    :  ℝ → x
           compute_power   :  x → ℝ
```

Here, the exponent n is a parameter to the class, `store_number` is of type $\mathbb{R} \to \mathbb{R}$ and writes to x; the method `compute_power` is of type $\mathbb{R} \to \mathbb{R}$ and reads its input from state variable x. Implementations differ by choices of the parameter n. For example:

```
object Squarer
  params   n               =  2
  state    x               :  ℝ
  methods  store_number    :  z ↦ x
           compute_power   :  x ↦ xⁿ
```

is an *inhabitant* of this structured type which computes the square for some input z of type $\mathbb{R}$. It is also an *instance* of the class, as defined in OOP:

```
class PowerComputerClass
  params   n               :  ℕ⁺
  state    x               :  ℝ
  methods  store_number    :  z ↦ x
           compute_power   :  x ↦ xⁿ
```

where in said instance the parameter n is set to the value 2. We can thus write, formally, `Squarer` : `PowerComputer`. In common object constructor syntax, one has `Squarer` $:=$ `PowerComputerClass(`n `= 2)`.

Note the difference between object, class and structured type: an object is an *instance* of a class, both of which have the same structured type, *inhabited* by the object (a value in the value space defined by the structured type). A structured type does not specify a particular implementation as a class; multiple classes with different implementations may inhabit the same type. There may be multiple distinct instances of a class, each with their own parameter values and states.

5.4.2 Scientific types for ML algorithms

We are now ready to describe how scityping can be used to specify ML algorithms, the central entity objects in ML theory.

We argue that a clear problem specification is crucial to define an algorithm scitype. Without a clear problem description, it is not clear what algorithms should look like. On the other hand, once we arrive at a clear task specification, the algorithm scitype required to solve it is also defined. We discussed learning tasks as a formal problem specification in Section 2.3. Accordingly, in our conceptual model, we define an algorithm scitype in correspondence to a learning task.

In particular, the learning process specification can be used to define the key interface points. The learning process specification tells us at which point, what data is given and what needs to be returned by an algorithm in order to solve a given task. From this, we can define the methods necessary for solving the learning tasks and their signatures, where the signatures will be defined in terms of function types based on the given input and output types.

As a key step in our conceptualization, we adopt an interface perspective for learning algorithms. That is, we consider a learning algorithm as defined by the ways in which it interacts with its environment (e.g. other algorithms, other software or human users). In practice, a learning algorithm is a collection of individual algorithms.[13] In typical task specifications, we have the following interface points:

- Specifying a given algorithm and any hyper-parameter settings,

[13]An algorithm, in turn, is typically defined in computer science as a computational procedure for solving a class of problems, or to perform a computation, consisting of a finite sequence of well-defined, computer-implementable instructions.

- Ingesting data to "fit" or "learn" the (statistical) model represented by $\hat{f}$,
- Calling the fitted model $\hat{f}$ to generate predictions.

Each interface point is defined in correspondence to the task specification. For example, the interface point for data ingestion and fitting is defined in terms of types of the input data and the returned prediction functional. Since we conceive of ML objects as entity objects, the prediction functional, rather than being returned, would be absorbed in the object as a state change. Similarly, the interface point for generating predictions is defined in terms of the function type of the prediction functional.

Note that depending on the learning task we may require algorithms to have additional interface points. For example, setting or retrieving hyper-parameters or fitted parameters, updating an already fitted model in an online learning setting, or saving and loading a fitted model for deployment. We give a more complete list of interface points in Table 8.1.

We make a number of remarks about the interface perspective:

- An interface specification can be used to define a *type* of learning algorithm. In line with OOP, whenever another algorithm has the same interface, it is considered as having the same type. For example, all time series classification algorithms allow the modes of access described above – specification, fitting, prediction – and this can be used as a *defining property* subject to assumptions on their statistical nature.
- Mathematically, the task can be understood as the codomain of the algorithm scitype. While algorithms are functions that, given some data, fit and return a function, the task defines the set of destination into which all of the returned functions of the algorithm are constrained to fall.
- Due to its close correspondence to a learning task, a scientific type captures the scientific purpose of ML algorithms. Scitypes help to uncover the implicit

task specification by defining clear algorithm categories with "intention-revealing interfaces" that reflect the learning task they are meant to solve.

- It is important to separate the concepts of the learning algorithm and possible workflows it is part of. For example, predicted labels from a time series classification algorithm can then be used to obtain performance certificates for the model. However, note that this is no longer an interaction directly with the algorithm, but with artifacts it produced, namely the predictions. Conceptual separation of use cases, workflows and interfaces is common in software design practice, but not always present (or necessary) in methodology-oriented data science. Conceptual hygiene in this respect allows a clear treatment of methodological units, as well as clear formal separation of means from ends.

- The interface perspective is substantially different from common exposition in ML literature, where learning algorithms are often defined in terms of a mathematical, statistical or algorithmic model. However, this is not contradictory but complementary: both the mathematical and algorithmic specification, as well as the interface are important and necessary to fully specify a particular algorithm. The distinction between internal specification and interface perspective allows to formally distinguish specific algorithms and types of algorithms – a recurring theme in this thesis. Ultimately, the interface perspective enables us to separate *what* the algorithm does from *how* it does it.

We will now illustrate the definition of algorithm scitypes with an example.

5.4.3 Example: Cross-sectional supervised learning

Before turning to the time series setting, it will be helpful to consider the familiar example of cross-sectional supervised learning. Understanding the cross-sectional setting will also be helpful for understanding reduction approaches from time series to cross-sectional tasks. In particular, we derive an algorithm scitype for a cross-sectional classifier and regressor defined in Section 2.5. These examples will illustrate how scitypes can explain key aspects of cross-sectional frameworks such as `scikit-learn` [205]. In Chapter 8, we will show how scitypes can generalize the design to the time series domain and guide the development of new frameworks such as `sktime`.

As argued above, we start with the task specification for cross-sectional supervised learning. Following the "consensus" formulation of that task, introduced in Section 2.5, we can define a scitype for a cross-sectional classifier as a combination of a structured type specifying the interface and its statistical properties.

For cross-sectional classification, the structured type is given by:

```
type CrossSectionalClassifier
  params   paramlist  :  paramobject
  state    model      :  mathobject
  methods  fit        :  $(\mathcal{X} \times \mathcal{Y})^M \rightarrow$ model
           predict    :  $\mathcal{X} \times$ model $\rightarrow \mathcal{Y}$
```

where `paramobject` and `mathobject` are types of abstract representation of parameters and model objects respectively (e.g. umbrella types). The structured type would not yet be the full scitype without reference to the definition of the types of the input and output arguments in the method signatures, specifically $\mathcal{X}$, $\mathcal{Y}$ and M. We refer to Section 2.5 for these definitions.

We can of course define a `CrossSectionalRegressor` scitype in the same way, adjusting the domain of the target variable, $\mathcal{Y}$, accordingly, as discussed in section 2.5.

Note that the cross-sectional nature of the learning task implies two additional statistical properties. The learning algorithm, specifically the result of fitting, will be invariable to permutations of the i.i.d. instances and variables, barring variations due to the randomness involved in these algorithms. This means that, when represented in tabular form, the output of `fit` will not change if the order of the rows (instances) or columns (variables) is permuted. In other words, the order of instances and variables does not matter.

Example 2: `scikit-learn`'s implementation of the `MajorityDummyClassifier` class

```
classifier = DummyClassifier(strategy="most_frequent")
classifier.fit(y_train, X_train)
y_pred = classifier.predict(X_test)
```

Notes: `X_train` and `y_train` denotes the cross-sectional training data and label vector, respectively, `X_test` the cross-sectional test data, and `y_pred` the predicted label.

Inhabitants of the `CrossSectionalClassifier` scitype are specific classification algorithms that follow the interface and statistical properties specified by the scitype. For example, the following class implements a naive classification strategy for always predicting the majority label of the training set:

```
class MajorityDummyClassifier
  state     model     :  $\mathcal{C}$
  methods   fit       :  $((x_1, y_1), \ldots, (x_N, y_N)) \mapsto \underset{c \in \mathcal{C}}{\operatorname{argmax}} \sum_{i=1}^N \mathbb{1}[y_i = c]$
            predict   :  $(x, c) \mapsto c$
```

This class has no parameters, i.e. an empty parameter list. Note that this is not the only possible implementation of this strategy, since instead of only storing the majority class in `fit`, one could store all the training data and compute the majority class later in `predict`.

Example 2 shows the implementation of the `MajorityDummyClassifier` algorithm in `scikit-learn` [205]. When comparing the `CrossSectionalClassifier` scitype with `scikit-learn`'s core API, it becomes clear that this scitype is central to `scikit-learn`'s interface. In particular, the universal base class of `scikit-learn` [205] is the `BaseEstimator` class. It requires concrete implementations to specify scitype-defining `fit` and `predict` methods with a uniform interface. In addition, it implements `get_params` for hyper-parameter inspection, and generic functionality for hyper-parameter specification and construction (e.g. the `__init__` method in Python). In `scikit-learn`, there are also scitype-specific base classes for regressors and classifiers, namely `RegressorMixin` and `ClassifierMixin`, but their remit is mainly accessory functionality (scoring) and scitype inspection (signposting by inheritance), rather than handling the `fit`/`predict` interface. Note that in `scikit-learn` adherence to the underlying scitype is enforced through conventions and unit testing, following "design by contract" principles [188], rather than strict inheritance from abstract base classes. We comment on the implementation of scitypes in Section 5.5 below. Similar designs can be found in other frameworks such as `mlr3` [151], `MLJ` [31] or `Weka` [101].

Finally, note that we may also consider the following more general definition:

```
type SupervisedLearner
  params   paramlist  :  paramobject
  state    model      :  mathobject
  methods  fit        :  $(\mathcal{X} \times \mathcal{Y})^N \rightarrow$ model
           predict    :  $\mathcal{X} \times$ model $\rightarrow \mathcal{Y}$
```

which can be considered a parametric type in $N, \mathcal{X}, \mathcal{Y}$. Again, the structured type is not yet the full scitype, because supervised learners can be defined for multiple choices of N, $\mathcal{X}$ and $\mathcal{Y}$. Classifiers and regressors are defined by specific choices of $\mathcal{Y}$. Admissible input data types $\mathcal{X}$ define specific sub-types of supervised learners. For example, above, we have seen a scitype for cross-sectional supervised learners. In Chapter 8, we will define a scitype for time series supervised learners.

Having defined scitypes and discussed cross-sectional supervised learning as a key example, we are now in a position to derive key design principles and patterns for mapping objects from ML theory onto software. Before we move on, we make some remarks on the implementation of scitype systems and review prior art in existing literature and toolboxes.

5.4.4 Beyond scitypes: Finer categorization of objects

In some situations, we may want to introduce a finer categorization of classes of the same scitype, without having to define a whole new set of scitypes. In these situations, it may be helpful to consider class traits to complement a scitype system. For example, a class trait can be defined for whether a learning algorithm can handle missing data, whether a classifier can handle multi-label problems, or whether a forecaster accepts exogenous data. Such traits are useful for user guidance. They allow users to query and filter algorithms by trait in addition to scitype. They are also useful for testing. They allow developers to define a set of common tests to be run on all classes with the same traits.

Traits can be implemented as class attributes specified on the level of concrete classes, possibly with some functionality for managing traits residing in an abstract parent class. The use of traits for ML algorithms is common in existing toolboxes. For example, in `scikit-learn` [205] they are called "algorithm tags".

5.5 Implementation remarks

In this section, we make a few general remarks on the implementation of scitype systems. We discuss the implementation of specific scitypes in the next chapter.

5.5.1 The case for object-oriented programming

A key architectural principle which has already stood the test of time is that of representing learning algorithms as objects, in the sense of the programming paradigm called "object-oriented programming" (OOP). We gave an overview of key OOP concepts in Appendix A. OOP is the predominant programming paradigm for ML toolboxes.[14] While OOP is often assumed implicitly without rationale, we here present key reasons why using OOP is well suited for developing ML toolboxes.

OOP is a general, not ML-specific motif supported by a programming language. OOP may take different forms depending on the language. For example, it may be of imperative, functional or mixed flavor. Note that on first sight, our conceptual model may seem to exclude functional programming paradigms and favor imperative (classical) object orientation (e.g. insisting on lineage of entity objects or methods). This is not true. While it is less enforced by other languages, entity lineage (the object remembers its "identity") and method lineage (methods

[14]All of the major toolboxes mentioned in this thesis follow OOP, including `scikit-learn` [205] in Python, `Weka` [101] in Java, `MLJ` [31] in Julia, and `mlr3` [151] or `caret` [148] in R.

"belonging" to an object can be queried) is implementable in functional object-oriented paradigms. Some examples relevant for data science are: pure class-based OOP in Java or Python; functional, dispatch-based OOP in Julia [27]; and the various mixed OOP paradigms in R (S3, S4, R6) [50].

We argue that OOP is the natural programming language paradigm for designing ML toolboxes. The primary reason is that domain-driven design [78] calls for an implementation technology in tune with the particular design being followed. OOP allows us to create direct analogues to the mathematical objects found in our conceptual analysis keeping the software implementation in lockstep with the conceptual model. There are a number of reasons why conceptual objects found in ML theory lend themselves to OOP:

- Conceptual objects may exist in *multiple instantiations* (objects) following a common blueprint (class). For example, different instances 1 and 2 of a linear model fitted on some batch of data and instance 3 of a linear model fitted on some other batch.

- For a given (abstract) type of learning algorithm (e.g. cross-sectional classifier) we have *multiple concrete implementations* (e.g. random forest or support vector machines).

- Many programming task require the program to manage *multiple states*. For example, a newly constructed model and a fitted model after parameter estimation. We often access and modify the state through several operations (e.g. fitting and updating) and the state often spans several variables. Changes to those variables must be coordinated in order to maintain the consistency of the state. When construed as entity objects, these objects possess multiple internal states between which they can transition and which we can manage consistently through an interface.

- Objects *interact via defined interfaces* with each other. For example, a learning algorithm can ingests data represented as a data container object.

A pipeline object is composed of several component objects (e.g. transformers and a final learning algorithm).

5.5.2 Implementing scitype systems

Scitypes may be implemented in form of a full typing system on the level of a programming language as a typed language of objects, possibly dual to an existing machine type system. In this case, we can perform static checks at compile time to prevent ill-defined programs from running. However, limitations of most existing programming languages do not allow for more than a single (ordered) type system, and retro-fitting a type system onto a language not designed with type checking in mind can be challenging. In that case, we can still implement scitypes by performing sufficiently detailed dynamic type checks at run time to rule out ill-defined programs.

We can implement scitypes even in languages which only provide limited support for typing such as Python. For example following ideas from "design by contract" [188], we can define a scitype as a set of conditions. We write corresponding input checks (i.e. assertions) to ensure that these conditions are met by the input to an object's method. These input checks are called "pre-conditions". We can write corresponding output checks to ensure "post-conditions". Alternatively, we can use unit testing to check if the output from an object's method call complies with its scitype over a given range of common inputs. `scikit-learn` [205] follows this "design by contract" approach in defining and testing their algorithm types. Finally, even if scitypes are not implemented in software, they are still useful for conceptual analysis of ML theory.

5.6 Prior art

We are unaware of literature that discusses scitypes for ML theory objects in a general and mathematical manner.

In terms of formalism, the closest is perhaps the area of type theory for structured objects [210]. In terms of conceptual analysis, scitypes tend to be a frequent, though often implicit, concept appearing in discussion of ML algorithms, especially in the context of ML toolboxes. Implicit scityping of algorithms can be thought of as a conceptual guiding principle behind many existing toolbox designs, including `scikit-learn` [46, 205, 256], `mlr3` [151], `MLJ` [30] and `Weka` [101]. Scitypes for categorizing data sets (e.g. scientific types for data column in data frames) have been an increasingly common idea, but we are unaware of a fully explicit implementation of a full (sci)type system. Examples include data frame column typing in base R [215], explicit "semantic type" hints in `dabl` [195], and perhaps most stringently and explicitly realized in the parallel type system implemented by the `ScientificTypes.jl` module of the `MLJ` ecosystem [31], which also introduces the shorthand term "scitype".

5.7 Concluding remarks

In this chapter, we have developed our conceptual model for ML theory, with scientific types at its core. The solution we presented is the foundation for our second research contribution of this thesis laid out in Section 1.6, namely the development of a novel set of general, reusable design principles for ML toolboxes. These principles will provide answers to the research questions of how one can identify and motivate design solutions for ML toolboxes as discussed in Section 1.5.

We started by discussing parallels between common mathematical objects found in ML theory and software. We described common object attributes and interface points as well as statefulness, a key concept for ML objects. We then defined a new type system to categorize and abstract objects found in ML theory. As we have seen, a scientific type combines an structured type with a set of key statistical properties that objects of the type have to satisfy. Finally, we used scientific types to express common objects in ML theory in a way that is both mathematically precise, but also easily translatable into software. Using `scikit-learn` [205] as an example, we illustrated how scientific types can explain key aspects of contemporary toolbox design.

In the next chapter, we will derive key design principles and patterns based on scitypes.

Chapter 6

Design Principles and Patterns

This chapter is partly based on:

- Franz J. Király et al. 'Designing Machine Learning Toolboxes: Concepts, Principles and Patterns'. In: *arXiv preprint* (2021)

6.1 Introduction

Having introduced our conceptual model for ML theory in the previous chapter, with the scientific type system at its core, we are now in a position to derive practical design principles.

In this chapter, we express what we believe to be key principles for ML toolbox design. The solution we present will form the second research contribution of this thesis as laid out in Section 1.6. In particular, we will motivate, develop and formalize a novel set of general, reusable design principles for ML toolboxes. These principles will provide answers to the research questions of how one can identify and motivate design solutions for ML toolboxes as discussed in Section 1.5.

We propose these principles as our solution to the developer problem stated in Section 4.2. As discussed, the developer problem poses the research question of how to translate ML theory, and its underlying mathematical objects, into software. We derive our principles to address this problem from the conceptual model of ML theory developed in the previous chapter. Our conceptual analysis identified, formalized and abstracted key mathematical objects. Our principles, as we will see, describe key patterns for mapping these mathematical objects from ML theory onto classes and functions in a programming language.

We hope to show that our principles not only allow us to explain why certain designs are more successful than others, but also guide us in the design of new toolboxes. We believe that these principles, when followed, will ensure high-quality solutions, in the sense that they achieve the design quality criteria laid out in Section 4.3.

Together with the conceptual model from the previous chapter, we regard our principles as a first attempt to consolidate the science of ML toolbox design. Principles communicate the reasons for design decisions, not just the results [19]. In other words, they communicate the "why", not just the "what" of designs (e.g. software usage or features). We intend to present our principles in a way that other toolbox developer can follow them when improving existing designs or creating new ones.

The rest of this chapter is organized as follows: We will start in Section 6.2 by introducing three design principles for ML toolboxes. These principles serve as guiding principles. From these principles, we will derive key design patterns in Section 6.3. These patterns describe reusable solutions to recurring problems in ML toolbox design, building onto the classical design patterns proposed by Gamma et al. [90]. In Section 6.4, we give a brief overview of related literature.

Throughout the chapter, we illustrate how our principles and patterns can explain key aspects of existing frameworks with examples from `scikit-learn` [205]. For an illustration of how our principles and patterns can guide the development of new toolboxes, we defer to Chapter 8, in which we will develop the design of `sktime`. A first draft of these principles and patterns appeared in Király et al. [145]. We here refine some points, add details and align the structure and terminology with the rest of the thesis.

6.2 Design principles

In this section, we discuss the following three guiding principles for ML toolbox design:

- **Architectural separation of key conceptual concerns**. Key concerns are specification of tasks, data, learning algorithms and workflows. This is a special case of the common "separation of concerns", "layering" or "decoupling" design principles, applied to the categories found during conceptual analysis.
- **Scitype-driven encapsulation**. This includes learning algorithms and other key conceptual objects (e.g. data, distributions, loss functions). Interface points should mirror the conceptual abstractions for the scitype and its defining interface points. For example, parameter setting, fitting and prediction define a learning algorithm and its interface. This can be seen as a special case of the classical "encapsulation", "modularity" and "interface abstraction" design principles.
- **Domain-driven and declarative syntax.** Following domain-driven design [78], the language of the software should closely match the conceptual model of ML theory. For example, specification of learning algorithms should be as close as possible to their mathematical definition in terms of scitypes. This can be seen as a special case of classical "symbolic abstraction" and "language abstraction" design principles.

We proceed by discussing each of these principles in more detail.

6.2.1 Architectural separation of key conceptual concerns

The first design principle is about module-level architecture. To identify the right architectural categories means identifying key abstraction points – i.e. those elements that may be "switched out" while others are kept constant, as discussed in Appendix B. For example, the same algorithm can be applied to different data sets, or different algorithms of the same type can be used to solve the same task. Thus, tasks, data sets and algorithms should be abstraction points. In this vein, based on our conceptual model, we can identify four key abstraction points and thus architectural categories:

- *Learning tasks and related concerns*; specification of the data setting, learning process and performance indicators. For example, the cross-sectional learning task in abstraction as defined in Section 2.5, or in concrete reference to a data set and specific performance metrics such as the mean squared error.
- *Data sets and data related concerns*; representation and manipulation of concrete data containers. For example, the concept of a data frame and operations on it.
- *Learning algorithms and related concerns*; specification of abstract algorithm types and concrete algorithms, their specific operations such as data ingestion, fitting or application of the algorithm. For example, the concept of a specific cross-sectional classification algorithm, such as a random forest, including functionality for fitting and prediction.
- *Workflows and related concerns*; as special cases, workflows for model inspection, evaluation, deployment and monitoring, with key concerns being specification and execution. For example, the concept of a predictive benchmarking experiment, model performance diagnostics or interpretability analysis.

We argue that these concerns should be the basis for abstraction since they delineate "independent moving parts" in the conceptual model. While there are some inherent conditionalities between the categories (e.g. only certain workflows make sense for cross-sectional tasks), varying parts from key use cases are typically localized within just one of the categories (e.g. choice of data set or learning algorithm). In the following, we discuss important consequences from this principle and address common "anti-patterns".[1]

[1]An "anti-pattern" is a common but poor solution to a recurring software design problem as

6.2.1.1 Separation of model specification and data specification

Models can be fitted or applied to data, but specification of learning algorithms should be separate from data specification. Application of learning algorithms, such as fitting or prediction, should be separate from operations intrinsic to the data.

- Important consequence 1: Software design of learning algorithms should be abstract in the sense that it can be applied to data sets of a certain kind.
- Anti-pattern 1: Writing code where the learning algorithm is specific and hence inseparable from a particular data set. The level of generality depends on the use case, but it usually should go substantially beyond a "data frame with the same column names".
- Important consequence 2: Data cleaning operations sit in the data category, data transformations for the purpose of modeling sit in the learning algorithm category. This may put the same operation in either depending on purpose. For example, data cleaning for the purposes of consolidating a data set should fall into the data category. Missing data imputation for the purpose of applying an algorithm that does not support data missingness, should usually fall in the algorithm category, as part of a pipeline that includes the imputation and final prediction algorithm, since the primary purpose is not related to data but to modeling.
- Anti-pattern 2: Conflating concerns of modeling or model evaluation with data preparation, for example placing imputation or feature extraction for the purpose of modeling not with modeling but with data loading; or making the creation of training and test sets for algorithm training part of the data loading and cleaning workflow. In this context, the classical software design anti-pattern of conflating conceptual concerns can easily enable common methodological mistakes along the lines of information leakage.

described by Brown et al. [44].

6.2.1.2 Separation of model specification and task specification

Fitting and application of learning algorithms requires task information (e.g. target variable or forecasting horizon), but model specification should be separate from task specification.

- Important consequence 3: Software design of learning algorithms should be abstract in the sense that it can be applied to tasks of a certain type, for example, the cross-sectional learning task as described in Section 2.5 or the forecasting task described in Section 3.3.2. Consequently, one should be able to specify a cross-sectional learning algorithm without reference to the target variable to predict, or any actual data set for that matter. Similarly, one should be able to specify a forecaster without reference to a forecasting horizon. The information of specific data sets should be recognized by, but extrinsic to the learning algorithm.

- Anti-pattern 3: A design that fails to separate specification of learning algorithms and tasks. For example, a learning algorithms that requires its hyper-parameters to be specified together with the target variable to be predicted. That is, a design where representation of a learning algorithm is not possible without a specification of how it is applied to a specific learning task.

6.2.1.3 Separation of model specification and workflow specification

A learning algorithm may be used in accordance with a specific task or within a workflow for model evaluation or deployment, but the workflow in which the learning algorithm is used should be separate from the learning algorithm itself.

- Important consequence 4: Concerns specific to model evaluation workflows (e.g. training and test sets, cross-validation or loss functions) should not be considered part of learning algorithms or their design. As before, the placement of a specific methodological concept should be motivated by its purpose. For example, re-sampling and performance estimation for the purpose of tuning is part of a learning algorithm, thus should be part of the learning algorithm category. By contrast, re-sampling and performance estimation for the purpose of model evaluation is part of the workflow, thus should be part of the workflow category.

- Anti-pattern 4: Conflation of learning algorithms and performance evaluation (e.g. a design where tuning and model evaluation are considered identical). As in anti-pattern 2, this anti-pattern facilitates common methodological mistakes, such as evaluating on the training set or conflating validation and test splits.

- Important consequence 5: Concerns related to monitoring and scrutiny of a learning algorithm (e.g. logging, diagnostics, interpretability) should be separate from the learning algorithm.

- Anti-pattern 5: The "learning algorithm that scrutinizes itself", conflating the concern of carrying out learning with the concern of checking how well it has been carried out. As in anti-pattern 2 and 4, this anti-pattern facilitates common methodological mistakes, such as circular reasoning in performance estimates and certification of algorithm predictive reliability.

6.2.2 Scitype-driven encapsulation

The next set of design principles primarily concerns module-level and object-level design within the architectural categories established by the previous principle. Our key argument is that the design of units within modules, especially classes and objects, should be driven by the scitypes identified during conceptual analysis. The rationale for this principle is the centrality of scitypes in our conceptual model. According to domain-driven design [78], such conceptual lines should guide design decisions, especially if objects can be delineated and formalized as mathematical units. Following this reasoning, the correspondence between scitype and software units should be taken into account for the definition of:

- Modules within the above architectural categories, particularly when thinking about which functionality to "bundle",
- Objects, functions and their interfaces; especially when considering which functionality to encapsulate and their key interface points,
- Entity objects, their state changes and classes implementing entity objects.

In the following, we derive some important consequences for object-oriented design.

6.2.2.1 Representation of scitypes as the simplest possible abstract classes

The scitypes found during conceptual analysis should be mapped onto the simplest possible abstract classes. A scitype will define the class interface that multiple classes can follow, ensuring non-proliferation of classes. Scitype inhabitants can then be implemented as concrete classes inheriting from the abstract class.

- Important consequence 1: Key interface points identified during conceptual analysis define the class design for instances of a given scitype. For example, the scitype of a cross-sectional learning algorithm maps onto the familiar `fit`/`predict` interface in `scikit-learn` [205].
- Anti-pattern 1: Individual objects of the same scitype are implemented in separate, disconnected parts of the software or different objects of the same scitype have inconsistent interfaces.
- Important consequence 2: Scitypes defining entity objects with non-trivial state changes should map onto classes or interfaces that ensure that the state and its changes remain traceable with the "entity". For example, if algorithms are conceived of as entity objects – as in our conceptual model – then they should retain their identity when fitted.

- Anti-pattern 2: Different states of entity objects are separated and state changes are not traceable. For example, fitting a cross-sectional prediction algorithm produces a separate object representing the fitted model which does not allow to trace back its entity lineage to the model and parameter specification that were used in fitting.

6.2.2.2 Operations with conceptual objects are methods of classes, operations on conceptual objects are operations on classes

Operations *of* conceptual objects should be methods of the class implementing that conceptual object (e.g. model fitting or application of a fitted model). Operations that work *on* conceptual objects should be operations on classes (e.g. pipeline building). As we will see in the next section, operation *of* objects invoke behavioral design patterns (e.g. encapsulation, interface abstraction, inheritance and the strategy pattern), whereas operations *on* objects invoke structural patterns (e.g. composite, adapter, decorator pattern).

- Important consequence 3: During conceptual analysis, mathematical definitions should be scrutinized for which operations of an object are key, in the sense of being a defining characteristics of its scitype. Key operations should be methods of the class that implements the object, or otherwise closely coupled to the object (such as by being a queriable part of the same module). For example, fitting a model should be a method of the class that implements the model.

- Anti-pattern 3: Uncoupling the representation of a conceptual object from the functionality that is considered part of it in the conceptual model. For example, a design where it is impossible to find the function to train a cross-sectional prediction model given only the model.

6.2.2.3 Operations on conceptual objects with a scitype define a higher-order scitype

A higher-order scitype should be considered in design if it is a recurring element in typical workflows. For example, composition of individual learning algorithms into a pipeline or ensemble. In terms interface design and implementation via classes, higher-order algorithm scitypes found during conceptual analysis should be considered first-class types.

- Important consequence 4: A good design will treat both primitive (zeroth-order) scitypes (e.g. cross-sectional learning algorithm) and higher-order scitypes (e.g. composition algorithms) consistently. In particular, class and interface designs should be consistent between primitive and higher-order scitypes.
- Anti-pattern 4: Higher-order scitypes are second-class citizens, for example higher-order scitypes may lack encapsulation, have inconsistent interfaces or may be implemented as functions while primitive scitypes are implemented as classes.

6.2.3 Domain-driven and declarative syntax

So far we have argued that conceptual analysis and scitype formalism should guide module and object level design. With the third and last principle, we argue that scitypes should determine syntax of usage.

Most importantly, this should cover the following use cases:

- Specification of conceptual objects, including algorithms and other mathematical objects, especially for hyper-parameter settings, composites, and higher-order constructs,
- Execution or application of conceptual objects, especially when using scitype-defining methods,

- Inspection of conceptual objects, especially when inspecting parameters, properties, or state.

We motivate this design principle from domain-driven design, more specifically the principle that the language and structure of software should closely follow the conceptual model [78]. Since conceptual algorithmic or mathematical objects are key concepts in our conceptual model, and operations on said objects key interface points, it follows that the software syntax for specifying and interacting with such objects should be as close to the conceptual model as possible.

We list a few requirements arising from this principle:

- *Level of abstraction.* The highest level of abstraction should be at least at the level of key mathematical and algorithmic objects found during conceptual analysis. It should not be necessary to access or configure specifics of implementation. This is a special case of the "no boilerplate code" design principle, where, in accordance with this principle, boilerplate is defined as being lower-level than the key conceptual objects.

- *Congruence of syntax and conceptual language.* Syntax should follow, as closely as possible, the conceptual model of conceptual objects, particularly the scitype formalism.

- *Consistency of syntax.* Syntax of specification and usage should be consistent across different conceptual objects, and across different levels of abstraction. This should in particular be the case for properties shared across objects with different scitypes, such as the representation of parameters, states or methods.

- *Parsimony of syntax.* Syntax of specification and usage should be as simple as possible, while being at a sufficient level of abstraction. This is a special case of the "simplicity" design principle.

Until now we have discussed three guiding principles for ML toolbox design. In the next section, we will derive from the above principles key design patterns that can drive practical design solutions.

6.3 Design patterns

Having laid out the above set of guiding principles, we are now in a position to derive practical design patterns. These patterns are based on the well-known patterns proposed by Gamma et al. [90]. A summary of the most relevant patterns for our purposes can be found in Section B.3.

Our patterns describe ways of mapping objects found during conceptual analysis from ML theory to software. Each pattern names, describes and illustrates an important recurring design solution to the developer's problems. Design patterns make it easier to reuse successful toolbox designs by making them more understandable and accessible to developers.

In particular, we will discuss the following design patterns:

- Universal interface points for conceptual objects,
- The scitype strategy pattern,
- The scitype composition patterns, both for run-time and compile-time compositions of conceptual objects,
- The scitype co-strategy pattern.

We proceed by discussing each of these patterns in more detail.

6.3.1 Universal interface points

Our conceptual analysis of ML theory as outlined in Chapter 5 implies a number of key interface points that implementations of any conceptual objects should have. We call these "universal interface points".

Description

We list the universal interface points in Table 6.1.

Note that hyper-parameters and properties are separate concepts by definition of the interface. We explicitly require these concepts to be separated by the interface, due to them being separate in the conceptual model, rather than due to being an implementation necessity. Further, the respective universal interface points should not only returns values, but also the set of hyper-parameters or properties. This is an important requirement for later composition patterns.

Entity objects have additional universal interface points related to managing the object's state. These are listed in Table 6.2. Note that we do not include state change itself as a universal interface point, since the number and quality of interface points for state changes depend on the specific scitype of the object.

Table 6.1: Universal interface points for conceptual objects

Interface point	Description
Specification	Construction of the object, and specification of hyper-parameters (e.g. constructing a normal distribution with `mean`=0, `variance`=1)
Scitype inspection	Query that returns the scitype of the object (e.g. a distribution)
Domain inspection	Query that returns the domain of the object (e.g. absolutely continuous distributions over the reals)
Hyper-parameter inspection	Query that returns names, values, and type/domain of hyper-parameters (e.g. retrieving the value (=1) and type (real number) of `mean`)
Hyper-parameter setting	Setting values of hyper-parameters (e.g. setting `mean` to 2)
Property inspection	Query that returns names, values, and type/domain of properties including parameters and traits (e.g. kurtosis of the distribution)
Object persistence	Saving and retrieving of objects (e.g. storing the distribution once constructed)

Implementation

This pattern can be implemented as follows:

Table 6.2: Universal interface points for entity objects

Interface point	Description
State inspection	Query that returns the state of the object, e.g. “fitted” or “unfitted” for a linear regression model
State variable inspection	Query that returns names, values, and type/domain of state-defining variables, e.g. regression coefficients

- *Use of abstract base classes and inheritance for universal interface points.* Abstract functionality for universal interface points should be implemented by a base class template. This follows the strategy and template pattern from Gamma et al. [90]. In contrast to the scitype strategy pattern discussed below, this pattern is not specific to a scitype, but applies to all conceptual objects. Any class implementing a conceptual object should inherit from this base class, in the process being endowed with the interface of the scitype (strategy pattern) and any predefined functionality specific to the scitype (template pattern) [90]. As a result, methods and usage of universal interface points is designed to be universal across conceptual objects regardless of their scitype. For example, the universal base class would implement general hyper-parameter functionality or trait inspection. Depending on use case or language, universal interface points for entity objects may also be part of that base class (and not used for value objects), or added by mix-in classes or decorator patterns.

- *One method (at most) per universal interface point.* Universal interface points should map onto one and only one method. For example, there should be one method by which all parameters can be inspected, as opposed to one method for each parameter. Specific scitypes may implement “shorthand” methods for individual parameters or traits, but the pattern requires a universal method per interface point.

- *Compositionality requirements.* Interface implementations need to satisfy certain requirements to ensure that objects are composable, as discussed in the next two patterns.

Examples

The universal base class of the `scikit-learn` [205] toolbox is the `BaseEstimator` class. It implements some of our suggested universal interface points. For example, `get_params` for parameter inspection and generic functionality for construction and specification in the constructor (e.g. `__init__` in Python). Notably absent from `BaseEstimator` are scitype inspection and a uniform interface for state variable inspection (e.g. attributes that change during fitting and change the state of the estimator from "unfitted" to "fitted"). Instead, inspection is specific to concrete classes and added on a descendant level. We argue that algorithms should also have a uniform interface for inspecting inference results (e.g. fitted coefficients) – something which is currently missing from most major toolboxes. This would allow us to develop higher-level functionality, such as composition algorithms or model interpretability diagnostics, which make use of fitted parameters of component algorithms. We give an example of a composition algorithm that use fitted parameters in Section 8.5.3.

6.3.2 The scitype strategy pattern

The central pattern used in contemporary toolbox design, and one that is thought of as the defining characteristic for scikit-learn-like toolboxes is what we call the "scitype strategy pattern". It can also be considered the primary pattern implied by the design principles in Section 6.2.2. As already evident from the naming, this pattern follows the strategy pattern from Gamma et al. [90] in the specific context of scitypes. As we will see below, it is also closely related to the template pattern.

Description

The basic idea of the strategy pattern described by Gamma et al. [90] is to define a family of algorithms, encapsulate each algorithm and makes the algorithms interchangeable within that family. In other words, the strategy pattern lets the algorithm vary independently from clients that use it, be it users or other objects. This avoids hard-wiring variable parts into code, lets you vary the algorithm independently of its context. This makes it easier to switch, understand and extend algorithms. For the scitype version of the pattern, we define the "family of algorithms" as algorithms of the same scitype.

The pattern applies to concrete classes that implement specific conceptual objects of a given scitype. For example, concrete probability distributions (e.g. Gaussian distributions) that follow the "distribution" scitype. Or concrete learning algorithms (e.g. random forest classifier) that follow the "cross-sectional classifier" scitype. Through the named interpretation, a scitpye reveals the data scientific purpose of the conceptual object.

The pattern provides both abstraction and conformity for instances of conceptual objects of a given scitype. The pattern ensures that objects that only differ with respect to their internal representation, but have the same scitype, also share have the same interface. The pattern avoids exposing object-specific details to users. It makes objects with the same purpose easier to use, compare and interchange. When applied to learning algorithms, the pattern encapsulates the intricacies of different algorithms ("how") behind a clear interface ("what"). As a result, the interface of an algorithm will reflect the type of task that it is designed to solve, in line with the idea of "intention-revealing" interfaces from domain-driven design [78]. The key advantage of a common interface for algorithms are interchangeability of algorithms of the same scitype at run time. As we will see in the next pattern, the strategy pattern also enables us to develop modular composition algorithms which require component algorithm to have the same interface.

Implementation

The pattern can be implemented as follows:

- *Use of abstract classes and inheritance for scitype-specific methods.* Abstract functionality for universal interface points should be implemented by a scitype base class. Any concrete object of a scitype should inherit from the corresponding scitype base class, in the process being endowed with the interface of the scitype (strategy pattern) and any predefined functionality specific to the scitype (template pattern) [90]. Alternatively, a template can be adhered to by a combination of implicit specification and inheritance applied to concrete objects.

- *Use of the strategy pattern for concrete objects of the same scitype.* Concrete conceptual objects of the same scitype should behave interchangeably within key workflows, and share the same interface for key operations. For example, all cross-sectional classifiers should implement a `fit` and `predict` method with the same signature which are called during training and prediction within a workflow. This is ensured by adhering to the strategy pattern where the "family of algorithms" is defined in terms of being of the same scitype [90].

Concrete classes should (compatibly) inherit both from the universal base class (see above) and adhere to a scitype-specific strategy pattern. This can be handled in multiple ways where suitability may also depend on language idioms. For example, the scitype base class inheriting from the universal base class, multiple inheritance, or a combination of inheritance and implicit convention.

Examples

In `scikit-learn` [205], classifiers and regressors adhere to the scitype strategy pattern for the scitypes of cross-sectional classifiers and regressors. Classifiers and regressors are required to implement scitype-defining `fit` and `predict` methods with a consistent interface. Interface conformity is enforced through unit testing rather than explicit inheritance following a "design by contract" approach as discussed in 5.5. There are also scitype-specific base classes for regressors and classifiers called `RegressorMixin` and `ClassifierMixin`, but their remit is mainly accessory functionality and scitype inspection (signposting by inheritance), rather than handling the `fit` and `predict` interface.

6.3.3 Scitype composition patterns at run-time

In this section, we cover a number of construction patterns for use cases that involve implementations of operations on conceptual objects at run time.

Description

A composition algorithm takes in one or more conceptual objects of potentially different scitypes and combines them into a single, composite object. Common examples are tuning, ensembling, pipelining or reduction. The basic idea of the scitype composition pattern is to compose objects into composite objects that lets clients treat individual objects and composite objects uniformly [90].

It is important to distinguish the composition operation (the "compositor") from the inputs to that operation (the "components") and the output of that operation (the "composite"). In general, we can define a composition operation in terms of formal scitype notation as follows:

$$A_1 \times ... \times A_N \to B \tag{6.1}$$

where A_i for $i = 1, \dots, N$ with $N \in \mathbb{N}^+$ and B denote scitypes. We call the A_i input or component scitypes and B the output or composite scitype. From this general formulation, we can express common ML composition algorithms as special cases:

- A *modification* can be defined by $N = 1$ and $A = B$, i.e. $A \to A$. In words, a modification takes in a single object and builds another object out of it, of the same scitype. Examples include tuning or homogeneous ensembling where all ensemble members are instances of the same concrete algorithm class (e.g. random forest).

- A *reduction* can be defined by $N = 1$ and $A \neq B$, i.e. $A \rightarrow B$. In words, a reduction takes in a single object and builds another object out of it, of different scitype. Examples include reduction from forecasting to cross-sectional regression where $A =$ `CrossSectionalRegressor` and $B =$ `Forecaster`, as discussed in Section 3.4.[2]
- A *scitype-homogeneous composition* can be defined by $A_i = B$ for all i. In words, a scitype-homogeneous composition takes in a number of objects of the same scitype and builds a object of the same scitype. Examples include heterogeneous ensembling where each ensemble member is an instance of a different concrete algorithm class.
- A *scitype-heterogeneous composition* can be defined by $A_i = B$ for at least one i and $A_j \neq B$ for at least some j. In words, a scitype-heterogeneous composition takes in a number of objects of the varying scitypes and builds a object of the same scitype as one of the input objects. Examples include pipelining where the composite object takes on the scitype of the last input object.

Note that for any of the above cases, there may be multiple concrete algorithms that implement the composition operation. For example, there are various concrete algorithms for model tuning (e.g. successive halving [133, 157], full grid-search or randomized grid-search, among others).

Before discussing the implementation of this pattern, we highlight an important consequence and anti-pattern:

- Importance consequence: An interesting, and often misunderstood, consequence of the pattern is the "currying out of method inputs" – that is, the

[2]The `CrossSectionalRegressor` and `Forecaster` scitypes are defined in Section 5.4 and 8.3.3, respectively.

inputs to component methods do not become part of the (implementation of the) composite, and instead inputs to the composite's methods.[3]

- Anti-pattern: For example, the operation of tuning a cross-sectional algorithm, if naively implemented, takes the cross-sectional algorithm and some training/test data on which the best parameters are determined. The output is the cross-sectional algorithm whose parameter are set to the "best parameters". However, this does not follow the composite pattern as outlined above. If it is correctly followed, the data is *not* an input to the composition operation. Instead, tuning is performed inside the fitting method of the composite, which has access to the data in deployment. Since the data is passed to the fitting later anyway (namely: at deployment), there is no need to pass the data to the composition operation too.

Implementation

For the implementation, the scitype composition pattern depends on the scitype strategy pattern discussed above. This is because a composite may delegate a method calls to its components. Call delegation can only be implemented reliably if the composite can expect components to have a consistent interface. This is precisely what the strategy pattern ensures. The composite may of course performs additional operations before and after delegating the call.

For example, consider a composite learning algorithm such as an ensemble. In this case, at training time, the composite (ensemble) object will delegate the `fit`

[3]Currying is a mathematical technique of converting a function that takes multiple input arguments into a sequence of functions that each take a single input argument. Given a function $f : A \times C \to B$, currying constructs a new function $h : A \to [C \to B]$. That is, h takes an argument from A and returns a function that maps C to B. Let us consider the example of model tuning, particularly the training of a tuned model. In this case, f would be the naive implementation, taking in some algorithm of scitype A together with data of type C and returning the tuned algorithm of scitype B, which in the case of tuning will be equal to A. By contrast, "currying out the method inputs" means to first apply the tuning composition operation to construct a new composite object with a `fit` method of type $[C \to B]$. We can then call `fit` with suitable training data of type C to produce the tuned algorithm of scitype B. Note that two operations happen during training: finding the best set of hyper-parameters and fitting a model with the best set of hyper-parameters to the training data.

call to its components (ensemble members, i.e. individual learning algorithms). At prediction time, the composite will first delegate the `predict` call and then aggregate the predictions from the individual components before returning them.

Composition pattern can be implemented as follows:

- *Compositions are concrete classes with composite's scitype.* The composition operation is modeled by a concrete class which has the scitype of the composite. For example, thresholding a cross-sectional regressor's output is implemented as a class that also follows the cross-sectional regressor scitype and its scitype specific strategy pattern. This is in line with principles 3 in Section 6.2.2. Note that composition learning algorithm are often modular: they work with any algorithm of a certain scitype and may even work with multiple scitypes. However, in some cases, composition algorithms are associated with specific concrete algorithms. In these cases, they are implemented as separate classes, following the compile-time composition pattern discussed below.

- *Composition as model specification.* The components are passed to the composite as objects at construction time, together with hyper-parameter settings specific to the composite, using a combination of the composite pattern and the factory pattern from Gamma et al. [90]. In composition that require multiple instantiations of the same object, the prototype pattern [90] is used to create new instances by copying the given "prototype" instance (e.g. in homogeneous ensembling a single instance is cloned a number of times to create the ensemble). This also leads to a first-order-like specification syntax in accordance with requirements in Section 6.2.2.

- *Components are properties of the composite.* The components are treated as properties of the composite.

- *Component state as part of the composite's state.* The components become part of the composite's state. Whenever a component's state changes, the composite's state changes.

- *Compositionality requirements for universal and scitype-specific interfaces.* In order for ML composition to work well, composition must allow nested hyper-parameter and parameter access. Component interface points should be visible via the composite interface points (e.g. hyper-parameter interfaces or state interfaces). This also puts additional requirements on the implementation of the universal interface points as well as scitype-specific interface points, including conventions on how to access hyper-parameters of components via the hyper-parameter interface of the composite, or how to list state variables of components via the state variable interface of the composite.

We proceed with outlining some variation of the pattern depending on the nature of the composition operation.

- *Adapter pattern for reduction.* For the reduction case, the adapter pattern from Gamma et al. [90] is suitable, in the form of adapting the input scitype to the interface of the output scitype. By adapting the interface of one algorithm to that of another one, the adapter pattern lets algorithms work together that could not otherwise because of incompatible interfaces.

- *Wrapper and visitor patterns for simple compositions.* In the case of simple modifications or reductions, wrapper and visitor patterns from Gamma et al. [90] may be alternatives worthwhile considering.

Note that in all of the above special cases, except reduction, the composite's scitype depends on the input scitype(s). We call these operations "output-polymorphic".

- *Strategy dispatch for output-polymorphic composition.* If there can be multiple composite scitypes dependent on component scitype, this behavior can be modeled by strategy dispatch on input scitype. This can be achieved with a suitable application of the strategy pattern (selecting by component scitype), by run-time decoration dependent on component scitype following the decorator pattern from Gamma et al. [90], by method dispatch (on component scitype), or a suitable combination thereof.

Examples

In `scikit-learn` [205], the tuning operation `GridSearchCV` and pipelining operation `Pipeline` adhere to the composite pattern. Both operations are output-polymorphic. `GridSearchCV` addresses the output-polymorphism by method polymorphism (the `score` method) and run-time decoration. For example, in `scikit-learn`, the `GridSearchCV` class exposes certain methods from its component algorithm. `Pipeline` addresses the output-polymorphism by method dispatch (to `predict` or `transform` depending on resultant scitype).

6.3.4 Scitype composition patterns at compile-time

There are situations in which operations may be preferable on the class level (i.e. at compile-time) rather than on the object level (i.e. at run-time).

Description

The most common situations are as follows:

- *Composition at compile-time.* Creating a class whose instances are composites as if obtained by the scitype composition pattern.
- *Contraction to a primitive.* Turning a composite with specific parameter settings into a predefined object, for example creating a class whose instances are fully specified cross-sectional prediction pipelines with a given parameter specification.
- *Contraction of components.* Removing the component hierarchy while leaving parameters settable at construction.
- *Contraction of parameters.* Fixing some parameters or components to specific values at compile-time.[4]

[4]Mathematically, this is related to the idea of partial function application, in which we fix a number of input arguments to a function producing a function of smaller arity.

The main advantage of compile-time composition over run-time composition are as follows:

- *Leveraging blueprints to create other blueprints*, for example, bulk-converting loss functions for cross-sectional regression to loss functions for cross-sectional prediction of numerical-valued sequences or segmentation.

- *Fixing a reference implementation for an important composite algorithm*, for example, for convenience in frequent reuse or reproducibility. A common example is the composite of a grid-search tuned support vector machine with prior input normalization, as the algorithm does not tend to perform well in isolation. This is especially important in ML with time series where many state-of-the-art algorithms are composites.

We proceed with outlining some variation of the pattern depending on the nature of the composition operation:

- *Compile-time decoration.* This is using the decorator pattern from Gamma et al. [90] (at compile-time), and is mostly suitable for operations with a single component, for example, modification or simple reductions. The pattern is used to decorate or wrap the component on class-level.

- *Compile-time factories.* This is using the factory pattern from Gamma et al. [90] (at compile-time). A typical implementation would suitably construct the final composite at construction of the factory class and return itself.

Examples

In `scikit-learn` [205], there exists `LogisticRegressionCV` and other related "compiled" tuning classes. These classes implement tuning meta-algorithms for specific concrete algorithm (e.g. `LogisticRegression`) which are more efficient than modular composition via `GridSearchCV`.

Any naive definition of short-hands for a frequently used composites is compile-time composition. Applying standard reductions such as thresholding in probabilistic classifiers can be seen as an instance of compile-time reduction.

6.3.5 The scitype co-strategy pattern

The last design pattern we propose is the "scitype co-strategy pattern".

Description

The pattern abstracts and encapsulates key concepts related to model specification or task specification as objects in their own right and with their own scitype. We call these concepts "motifs". Examples of such motifs include the forecasting horizon or complex model hyper-parameters. Note that the scitype is not defined from first principles in relation to a learning tasks, but in secondary relation to the primary object(s) from which it is abstracted.

We call this the co-strategy pattern because it forms the counterpart to the strategy pattern. Co-strategies often implement what one may conceptually define as a "specification", for construction or behavior of an algorithm. However, we avoid that term since "specification pattern" is an established design pattern with a different meaning.

The main use cases include:

- *Model specification.* Setting the values of hyper-parameters at model specification, especially if these hyper-parameters represent complex objects themselves, for example specifying the kernel function in a support vector machine.

- *Task specification.* For example, specifying task information, such as the target variable or forecasting horizon, and passing that information to learning algorithms, especially if representing that information requires complex objects.

- *Model evaluation workflows*. Specifying model evaluation workflows, for example re-sampling schemes such as cross-validation.

Implementation

The pattern can be implemented as follows:

- *Consistency of co-strategies.* If the same conceptual motif occurs as parameters or arguments in the context of multiple operations, it should be handled by co-strategies of the same scitype. For example, if cross-validation specifications appear in tuning operations and evaluation workflows, they should be handled by the same class or scitype, as opposed to one class for tuning and one class for evaluation.

- *Use of the strategy pattern for the motif.* The scitype strategy pattern can implement a "conceptual kind" of the motif. For example, specification of a concrete class for a certain type of learning task, with instance representing specific tasks with reference to a data set to which a learning algorithm is applied, or specification of cross-validation parameters used in model tuning.

- *Methods implement operations intrinsic to the motif.* Functionality that is specific to the motif without the context should be located with the class implementing the motif rather than the related primary object. For example, if the motif abstracted is a data transformation, functionality to execute the data transformation should be localized with the specification of the data transformation. Or if the motif is task specification, functionality related to the task specification should be located with the class representing the tasks specification (e.g. manipulations of the forecasting horizon should reside in the class implementing the forecasting horizon).

Examples

Cross-validation specifications (such as the cross-validators in the `model_selection` module of `scikit-learn`) can be seen as a co-strategy obtained from encapsulating the configuration for cross-validation in tuning via `GridSearchCV`. Transformers, a widely used conceptualization of data manipulation operations (e.g. feature extraction) used as part of a pipeline composite, can also be considered a co-strategy obtained from encapsulating prior data manipulation steps in the modification in which we first perform the data manipulation and then apply the cross-sectional algorithm.

6.4 Prior art

We are unaware of any literature that formulates design principles specific to ML toolboxes and derives these from domain-driven conceptual modeling. To our knowledge, while there a number of references that discuss design principles, discussion usually take place in the context of a specific toolbox without generalizing principles beyond that setting. For example, Buitinck et al. [46] discuss some of the consequences mentioned above: compare separation of concerns in Section 6.2.1 with the banana/gorilla/jungle metaphor, consequence 1 and 3 in Section 6.2.2 with "non-proliferation" and "composition", and consistency of syntax in Section 6.2.3 with "consistency".

The design patters, to our knowledge, are entirely novel in the sense of general and systematic formulation, while drawing inspiration from the classical and non-ML-specific software design patterns put forward by Gamma et al. [90] and domain-driven design practice described by Evans [78]. Many of the patterns presented here have been implicitly used in the design or implementation of existing toolboxes, notably `Weka` [101], `scikit-learn` [46, 205, 256], `mlr3` [28, 151] and `MLJ` [31], but without being explicitly identified as generalizable design patterns. The co-strategy pattern, and especially the encapsulation of task information in objects, is partially inspired by the APIs of `mlr3` [28, 151] and the `openML` platform [252].

6.5 Concluding remarks

In this chapter, we derived key object-oriented design principles and patterns from our previous conceptual analysis in Chapter 5. These principles and patterns address the developer's problem, stated in Section 4.2, which poses the research question of how to translate key concepts from ML theory into software. They are the second research contribution of this thesis as laid out in Section 1.6.

In particular, we first derived three general guiding principles for ML toolbox design, advocating for the architectural separation of key conceptual concerns, scitype-driven encapsulation as well as domain-driven syntax. Inspired by the classical software design principles proposed by Gamma et al. [90], we then derived key design patterns from these principles. Our design patterns provide reusable solutions to recurring problems in toolbox design.

Throughout the chapter, we motivated and illustrated our principles and patterns with the example of `scikit-learn`. We hope to have shown that our principles and patterns can explain key aspects of existing frameworks like `scikit-learn`.

In Chapter 8, we will use our principles and patterns, together with the taxonomy of tasks developed in Chapter 3, to design `sktime`. Before discussing the design of `sktime`, we will will review related software in the next chapter.

Chapter 7

A Review of Related Software

7.1 Introduction

One of the main goals of this thesis is to develop a unified software framework for ML with time series. Before we discuss the design and implementation of `sktime` in Chapter 8, we will review related software in this chapter. The review constitutes the third research contribution of this thesis as laid out in Section 1.6.

Our review focuses on open-source software in Python, mainly because we choose to implement `sktime` in Python. Our choice of programming language is motivated in Section 8.7. However, wherever relevant, we will also include software in other popular programming languages, particularly Java, R and Julia.

As we will see, various well-established ML frameworks exists, but most of them focus on cross-sectional data. To our knowledge, no comparable frameworks exist for ML with time series. At the same time, various smaller toolboxes exist for time series analysis, but most of them are limited in important respects: they focus on particular model families, learning tasks or steps in typical modeling workflows and often lack integration with more foundational cross-sectional frameworks.

The rest of this chapter is organized as follows: We start by reviewing cross-sectional frameworks in Section 7.2. We then turn to time series analysis toolboxes in Section 7.3. Finally, we review time series data containers in Section 7.4.

7.2 Cross-sectional machine learning toolboxes

The software ecosystem for the cross-sectional ML setting, as defined in Section 2.5, generally seems to be more mature than the time series setting, with various well-developed toolboxes and major frameworks.

The most popular framework for cross-sectional ML is arguably `scikit-learn` [46, 205, 256] in Python. Similar frameworks are available in other languages. In Julia, there is `MLJ` [30]. In Java, there is `WEKA` [101, 112], which additionally offers a graphical user interface (GUI). In R, there is `mlr3` [28, 151], `caret` [149] and `tidymodels` [148, 149], which form part of a wider ecosystem of related ML toolboxes. All of these frameworks offer an extensive range of tools, covering typical practitioners' workflows from model specification, training and tuning to validation and deployment. They include algorithms and functionality for data transformations, composite model building, tuning and model evaluation.

More importantly, these frameworks have established standards for the structure and language of applications and other toolboxes in the same domain. This includes key design patterns, such as representing learning algorithms as objects, the adherence to a uniform interface for fitting, predicting, handling of hyperparameter and composite model building. This is typically supported by a clear separation of components (e.g. data, algorithms), separation of algorithm types for different learning tasks (e.g. classifiers and regressors), and clear hierarchies between primitive algorithms (e.g. random forest) and higher-order functionality (e.g. pipelining). Many of these patterns are captured in reusable design templates. For example, `scikit-learn` provides templates for different types of cross-sectional algorithm (e.g. classifiers and regressors). Through these templates, they have set standards which have been adopted by other toolboxes in the ecosystem (see e.g. `mlextend` [216] and `LightGBM` [141]). As a result, most applications and toolboxes have a similar structure. They are more consistent, easier to use and more interoperable.

However, these frameworks have a clearly defined scope. They are limited to cross-sectional data. Most if not all of their functionality assumes cross-sectional data. As we have seen, cross-sectional data is typically expected to be formatted in a tabular two-dimensional array, with rows representing i.i.d. instances and columns representing features. Time series data does not naturally fit into this setting due to its inherent dependency structure. Time series observations typically depend on past observations, violating the assumption of cross-sectional data of having i.i.d. observations.[1] As a consequence, cross-sectional frameworks usually consider ML with time series out of scope or provide only limited functionality that can be supported within a cross-sectional API.

7.3 Time series analysis toolboxes

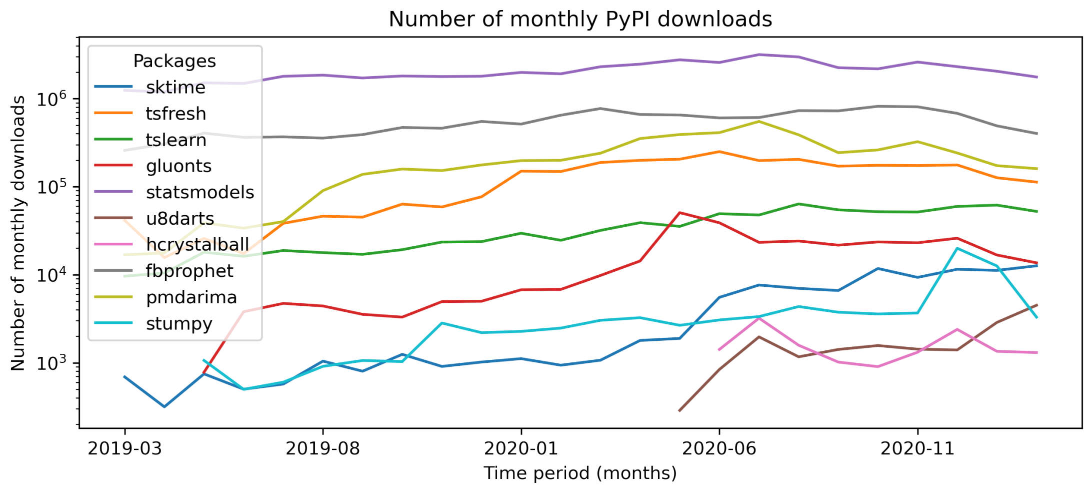
Most of the major ML libraries, as discussed above, focus on the more traditional cross-sectional setting. Beyond that setting, toolbox capabilities remain limited. We will now review open-source toolboxes for time series analysis in Python.

Numerous specialized time series analysis toolboxes exist. Some provide rich interfaces to specific learning tasks (e.g. forecasting or time series classification). Others provide implementations of specific model families (e.g. ARIMA). Still others provide tools for specific steps of common time series modeling workflows (e.g. feature extraction). In the following sections, we discuss popular open-source Python libraries in more detail.[2] Table 7.1 provides an overview of these libraries and their key functionality. Figure 7.1 shows the number of downloads for libraries

[1]The problem is not the data format per se, but rather the assumed statistical setting in which these algorithms have been developed. For example, one could represent a single time series in the tabular format, with rows representing time points and columns variables. However, this would violate the assumption that rows represent i.i.d. instances. While this may still provide a working solution in some cases, common model evaluation workflows that assume i.i.d. data are no longer reliable (e.g. random splitting of the data into train and test folds during cross-validation). Similarly, one could represent panel data in the tabular format, with rows representing instances and columns time points (at least univariate equal-length panel data). Again, while this may provide a working solution, it would violate an implicit assumption on the data. In other words, even though the data is represented in the same format, the meaning of the data has changed.

[2]Our selection of libraries is partially based on a public and collaborative list of time series analysis libraries available at `https://www.sktime.org/en/latest/related_software.html`.

Figure 7.1: Number of monthly PyPI downloads for popular open-source time series toolboxes in Python



Notes: The y-axis shows the total number of downloads per month from the Python Package Index (PyPI) for popular time series analysis toolboxes. The x-axis shows the month, for the last 2 years. Note that the number of downloads from PyPI may not be a reliable estimate of the number of users, as the number of downloads also include downloads from automated systems such as continuous integration services. Also note that libraries are not independent. For example, every time someone downloads `sktime` from PyPI, its dependencies are also downloaded, including `statsmodels` and `pmdarima`, among others. A full list of dependencies is given in Section 8.7.

over the last 24 months.

7.3.1 Time series classification/regression/clustering

Some libraries extend libraries for the cross-sectional setting, such as `scikit-learn` or `Weka`, to the closely related series-as-features setting, defined in Section 3.3.1. The series-as-features setting includes the time series learning tasks of time series classification, regression and clustering. While these libraries provide functionality for one or more of these tasks, they usually do not provide tools for other tasks. They also often lack tools for reduction to related cross-sectional tasks.

Notable examples of series-as-features libraries include the following:

- `tslearn` [244] offers algorithms and composition tools for time series classification and clustering.

Table 7.1: Overview of popular open-source time series analysis libraries in Python

Library	Classification	Regression	Clustering	Forecasting	Annotation	Transformations	Composition	Distances
`adtk` [8]					✓	✓	✓	
`cesium` [198]						✓		
`darts` [251]				✓		✓	✓	
`dtadistance` [186]			✓					✓
`fbprophet` [247]				✓		✓		
`gluonTS` [4]				✓	✓	✓	✓	
`HCrystalball` [106]				✓		✓	✓	
`pmdarima` [233]				✓		✓	✓	
`PyFlux` [246]				✓				
`pysf` [98]				✓		✓	✓	
`pyts` [80]	✓					✓	✓	✓
`ruptures` [250]					✓		✓	
`seglearn` [47]					✓	✓	✓	
`sktime` [171]	✓	✓		✓		✓	✓	✓
`statsmodels` [207]				✓	✓	✓		
`STUMPY` [153]						✓		
`tbats` [130]				✓				
`tsfresh` [55, 56]						✓		
`tslearn` [245]	✓		✓	✓		✓	✓	✓

Notes: The first column lists libraries in alphabetical order. The remaining columns indicate included functionality. Classification, regression and clustering refer to the time series, not the cross-sectional, tasks. Forecasting and annotation include different variations of the respective tasks (e.g. univariate, multivariate, panel and supervised forecasting). Transformations include different kinds of data transformations (e.g. data preprocessing and feature extraction). Composition refers to modular functionality for pipelining, ensembling, tuning or reduction. Distances refers to dedicated time series distances (e.g. dynamic time warping). For a regularly updated and more extensive overview of Python libraries for time series analysis, see `https://www.sktime.org/en/latest/related_software.html`. Also see the systematic review of Python libraries for time series analysis by Siebert, Groß and Schroth [230].

- `pyts` [80] provides algorithms and composition tools for time series classification.
- `seglearn` [47] provides `scikit-learn` compatible composition tools for reduction from time series tasks to cross-sectional classification and regression, however without providing dedicated API for the time series tasks.
- `dtadistance` [186] provides time series clustering algorithms and distances for comparing time series (e.g. dynamic time warping).

In terms of API design, `tslearn` and `pyts` closely follow `scikit-learn`, accepting panel data instead of cross-sectional data as input data during fitting. As a result, they provide APIs largely in line with the principles and patterns outlined in Chapter 6. For example, they separate model specification from data and task specification (see Section 6.2.1), follow scitype-driven encapsulation (see Section 6.2.2) and implement key universal methods (see Section 6.3.1) and the scitype strategy and composition patterns (see Sections 6.3.2 and 6.3.3).

The main downside of these libraries is that they do not offer a unified interface for learning tasks beyond the series-as-features setting of classification, regression and clustering. Note that while these libraries can still be used to solve other tasks (e.g. forecasting via reduction), they do not provide dedicated APIs for these tasks. Consequently, practitioners are required to write additional code to implement reduction approaches. For example, in order to solve a forecasting task with `seglearn`, one can use the provided functionality to transform the training series into the required cross-sectional data format and then apply cross-sectional regression algorithms from `scikit-learn`. However, `seqlearn` does not provide a dedicated forecasting API (e.g. for specifying task information such as the forecasting horizon). Instead, data and task information have to be converted first into a format that is compatible with the cross-sectional API. We discuss forecasting API design in more detail in Section 8.3.3.

Also note that, to the best of our knowledge, there exists little literature and software on time series regression as defined in Section 3.3.1.[3] This is surprising because many time series classifiers are composite algorithms that internally rely on a cross-sectional classifiers. So, to define time series regressor, one would simply have to replace the final cross-sectional classifier with its regressor counterpart. Yet, we are not aware of any Python library that provides time series regression algorithms.

7.3.2 Forecasting

There are a number of toolboxes designed for solving forecasting tasks.[4] One of the most popular forecasting toolboxes is arguably `forecast` [127] developed by Hyndman and Khandakar [124] in R, with its newest iteration called `fable` [249]. Together with their companion libraries (e.g. `hts` [115, 121] or `tsfeatures` [218]), they provide extensive functionality for forecasting, including statistical algorithms (e.g. exponential smoothing) and encapsulated ML algorithms (e.g. a common neural network architecture), as well as tools for data preprocessing, visualization and model evaluation. While there is still no comparable toolbox in Python, a number of forecasting toolboxes do exist:

- `statsmodels` [207] is mainly a library for econometric and statistical analysis, implementing many of the traditional panel data methods[5] and forecasting algorithms. It has submodules for generalised linear models, mixed effects models as well as for classical time series analysis and forecasting, including auto-regressive (AR), auto-regressive integrated moving-average (ARIMA), vector auto-regressive process (VAR) and Gaussian state space models. But it lacks the typical ML algorithms and tools for model selection and

[3]One of the exceptions is the repository developed by Hamilton [102] which is a small toolkit that provides specific reduction approaches from different time series learning tasks, including time series regression, to cross-sectional tasks. Also see Tan et al. [243] and the accompanying code repository.

[4]For a brief overview of available Python libraries for forecasting, also see Januschowski, Gasthaus and Wang [134]. For a larger encyclopedic survey, with a focus on the ecosystem in R, see Petropoulos et al. [209].

[5]For panel data modeling, also see `linearmodels` [17].

evaluation and is not compatible with more foundational frameworks like `scikit-learn`.

- `gluonTS` [4] provides a deep learning framework for probabilistic forecasting. While it does offer interfaces to other packages, including `forecast` in R, `gluonTS` focuses on deep learning and offers only limited support for statistical models, model composition or reduction to tasks covered by more foundational ML libraries like `scikit-learn`.
- `pmdarima` [233] ports `forecast`'s Auto-ARIMA algorithm [124] into Python and provides additional tools for seasonality testing, preprocessing and pipelining, but is limited to the ARIMA model.
- `fbprophet` [248] is available in both R and Python and provides a single algorithm based on a general additive model designed for modeling different forms of seasonality.
- `PyFlux` [246] offers a wide range of classical time series analysis and forecasting models. It implements generalized autoregressive conditional heteroskedasticity (GARCH) [32] and generalized autoregressive score (GAS) [62] models, among others. But like `statsmodels`, it lacks support for typical ML tools for composite model building, tuning and reduction.
- `darts` [251] provides various forecasting algorithms and composition tools, interfacing other libraries such as `statsmodels`, but lacks integration with more foundational frameworks like `scikit-learn`.
- `HCrystalball` [106] extends `scikit-learn` to forecasting, again interfacing other libraries such as `statsmodels` and `fbprophet` with limited support for composite model building.
- `pysf` [98] provides `scikit-learn` compatible composition functionality for supervised forecasting. But it is still at an early stage of development.
- `tbats` [130] provides BATS and TBATS algorithms [166] ported from R.

In addition, there are a number of repositories which collect and combine popular forecasting models via interfaces to existing libraries and tools to automate workflows, such as `atspy` [236] and the Microsoft forecasting repository[6], but none of them support composite model building.

Many of the existing libraries are collections of particular forecasting methods or family of methods, without providing a software framework as describe in Section 4.2.2. This includes `pmdarima`, `fbprophet` and `tbats`. Other libraries like `HCrystalball`, `darts`, `gluonTS` and `PyFlux` implement frameworks, but are mostly limited to the learning task of forecasting, with some functionality for closely related tasks of time series annotation. As a result, they only offer limited support for reduction and require custom code from users to implement reduction approaches and applications of cross-sectional or series-as-features algorithms.

`statsmodels` is arguably the most established library for forecasting algorithms in Python. However, `statsmodels` API lacks many of the key features of `scikit-learn` and often does not align with our principles and patterns defined in Chapter 6. Key points of departure from these principles and patterns include:

- Scitype-driven encapsulation is not consistently implemented (see Section 6.2.2). For example, many algorithms of the same scitype have different interface points for generating predictions. This deviates from the scitype strategy pattern (see 6.3.2).

- Given the deviation from the scitype strategy pattern, the scitype composition pattern is not followed either (see Section 6.3.3). As a consequence, composite models are often implemented in a single, fixed class or through optional hyper-parameters rather than through reusable composition algorithms with modular components. For example, deseasonalisation or detrending strategies are not configurable as components of a pipeline but instead as hyper-parameters. Model composition is complicated by the fact that model specification is not separated from data specification in

[6]`https://github.com/microsoft/forecasting`

statsmodels (see Section 6.2.1). Instead of specifying the training data during fitting, it has to be passed already during model construction. This makes it difficult to specify composite models such as ensembles or pipelines, without specifying the same training data for each component model.

- Algorithms are conceived of as value objects rather than entity objects (see Section 5.2.4 and 6.2.2). Fitting of algorithms produces a new value object, rather than changing the state of the algorithm itself. This makes it harder to trace fitted objects back to the algorithm and the hyper-parameters used for fitting.
- Algorithms lack important uniform implementations of universal interface points such as hyper-parameter inspection (see 6.3.1). For example, unlike `scikit-learn`, `statsmodels` does not have a `get_params` methods.

Also note that all of the above toolboxes define incompatible APIs. They require different ways of specifying, fitting and applying algorithms. For example, all of them require a different representation of the forecasting horizon, a key piece of information for specifying forecasting tasks. `statsmodels` requires an interval with a start and end value based on the absolute index of the training series. `fbprophet` requires the absolute time points. `pmdarima` requires the number of steps ahead relative to the end of the training series. Similarly, other libraries require varying specifications of the forecasting horizon. Neither follows the proposed co-strategy pattern which would allow us to specify the a more uniform task specification (see Section 6.3.5). We return to this point when discussing the implementation of `sktime` in Chapter 8.

In summary, while some of these toolboxes provide rich frameworks for forecasting, they often specialize on certain methods, lack support for model composition, are incompatible with more foundational frameworks such as `scikit-learn` (e.g. via reduction), and are often incompatible with each other. For practitioners, this makes it hard to combine functionality from different toolboxes and to fully leverage reduction relations. For developers, this makes it hard to build and integrate new methods without having to re-implement functionality that already exists elsewhere.

7.3.3 Transformations

As we have seen, most of the existing ML toolboxes focus on the more traditional cross-sectional setting. As discussed in Section 2.5, this setting presupposes cross-sectional data, which is typically expected to be formatted in a tabular matrix. If one wants to leverage the methods developed in this setting for solving time series tasks (i.e. via reduction), it is necessary to first transform the given time series data into the required tabular input format. There are a few specialized libraries that provide different strategies for that transformation:

- `tsfresh` [55, 56] allows to extract numerous features from time series using various time series analysis and decomposition techniques and to select the most relevant features based on statistical hypothesis testing. The extracted features describe basic characteristics of the time series, such as the number of peaks, the maximal value or more complex features such as the time reversal symmetry statistic. `tsfresh` support panel data and thus allows to transform panel data into the expected tabular format, so that one can apply cross-sectional algorithms. Although `tsfresh` even offers a transformer that can be integrated into a `scikit-learn` pipeline, it is still restricted by `scikit-learn`'s requirement that all input data – including for pipelines – must be in a tabular format. We, therefore, still have to write additional code to combine functionality from both libraries. In terms of API design, this can be seen as an instance where the scitype of the feature extraction algorithms implemented by `tsfresh` does not match with the class that implements the algorithm. That is, `tsfresh` implements algorithms that transform panel data into cross-sectional data, but is encapsulated in a transformer class defined in `scikit-learn` as a transformer that transforms cross-sectional data to cross-sectional data (see Section 6.2.2 for the principle of scitype-driven encapsulation).

- `STUMPY` [153] is a library for computing what is called the "matrix profile", a vector that represents the distances between all sub-sequences of a given length within a time series and their nearest neighbors [269].[7]
- `hctsa` (highly comparative time series analysis) [86–88] is a toolbox for extracting, visualizing and interpreting time series features. While based in MATLAB, `hctsa` provides an interface to Python. The related "catch22" set of 22 canonical time series characteristics can be extracted with the `catch22` library [172, 173].

In summary, the above toolboxes provide fundamental techniques to analyze time series. As such, they are applicable to a variety of time series learning tasks. But they only provide functionality for a specific step in typical workflows without the necessary tools to compose the workflow itself. Furthermore, they often do not provide or follow a consistent API for specifying and applying transformations, making it hard to combine them in the same applications.

7.4 Time series data containers

Any data analysis library depends on a data container. The data container is fundamental. It determines how the data in memory is represented, accessed and manipulated. Notable data containers in Python include:

- `NumPy` [104, 201, 258] augments Python by array objects for multi-dimensional data and fundamental data scientific operations. `NumPy` has become the standard library for handling numerical data and scientific computing in Python [190, 200]. However, `NumPy` lack a meaningful representation and integrated tools for labeled data, i.e. data that comes with metadata associated with each data point (e.g. column names or time index).

[7]For more details on the matrix profile and its applications, see `https://www.cs.ucr.edu/~eamonn/MatrixProfile.html`.

- pandas [184, 185] extends NumPy's arrays to data structures and tools with support for type-heterogeneous data (e.g. columns with different data types) and integrated handling of data labels, including column names and time indices. pandas has become the most comprehensive and flexible data container in Python. It now is the standard library for most complex data processing in Python. It supports various operations based on time indices (e.g. up/down sampling and slicing based on time points). However, while it offers extensive tools for processing time series, it is fundamentally designed for handling tabular data.[8] For example, to represent panel data, one usually has to work with a multi-index, where the first index level represents instances and the second level represents the time index. While this representation does allow to store and manipulate panel data, it has limitations for developing ML workflows. The reason is that in this representation, rows no longer represent i.i.d. instances. Much of the existing functionality for re-sampling provided by cross-sectional toolboxes like scikit-learn is therefore no longer usable. In addition, this representation usually assumes that the time index is shared either across instances or across variables.

- xarray [118] extends pandas to hierarchical (or multi-dimensional), labeled data structures. However, xarray does not offer support for variable-length panel data.

- awkward-array [211, 212] is a library for hierarchical variable-length data, partly based on NumPy, with support for data labels and data type heterogeneity. While awkward-array does allow to represent variable-length panel data, it has limited support for time index operations.

In summary, there are powerful and flexible data container libraries in Python, most notably NumPy [104] and pandas [185], with extensive support for time series.

[8]Note that pandas historically supported a data structure called the Panel for three-dimensional data, but its fixed rank design made it unsuitable for applications that require arbitrary rank arrays. Additionally, many of the features that made the pandas' DataFrame and Series objects so useful, were not fully available on the Panel, and it has been deprecated as of version 0.20 in favor of xarray.

However, most of them have limitations for developing ML libraries for time series, especially in the context of panel data and time-heterogeneous data (see Section 3.2).

7.5 Concluding remarks

This chapter reviewed open-source toolboxes for time series analysis and data handling, as well as more foundational toolboxes for cross-sectional ML. The review forms the third research contribution of this thesis as described in Section 1.6.

In summary, we have seen that major toolboxes exist for the cross-sectional setting,. But they usually consider time series analysis out of scope or only provide limited support for temporal data. Beyond the cross-sectional setting, toolbox capabilities remain more limited: various specialized toolboxes exists, but no overarching frameworks exist, at least none comparable to frameworks such as `scikit-learn`. The main limitations of the current open-source time series analysis ecosystem in Python can be summarized as follows. Libraries often (i) lack integration with more foundational cross-sectional toolboxes, (ii) are often incompatible with other time series analysis libraries, (iii) provide only little support for composite model building or (iv) focus on specific learning tasks, model families or workflow steps. As a result, practitioners typically face one of the following options, all of which have drawbacks:

- Use of a *specialized library* for specific model families, learning tasks or modeling steps (e.g. `statsmodels` [207]). The drawback is that these libraries offer limited functionality (e.g. usually no tools for model composition or tuning) and are incompatible with other libraries in the same domain. This makes it hard to combine functionality from different libraries and compare algorithms.

- Use of *reduction* from a time series task to a cross-sectional task. This is typically done via some data transformation, with feature extraction being the most common one (e.g. `tsfresh` [56]), and application of a cross-sectional algorithm from any library that provides cross-sectional algorithms. The drawback in this case is that reduction, without an overarching framework, requires hand-crafted and often error-prone code. The reduction approach is often neither adaptive, modular, composable nor tuneable (see Section 3.4). Extra care is needed in interpreting performance estimates since the estimates from the reduced task in general do not hold for the original task that one tries to solve.

- Use of *deep learning frameworks* (e.g. `Pytorch` [203] or `TensorFlow` [1]) with specific architectures for sequential data (e.g. recurrent neural networks [109, 221]). The drawbacks here are the same as in the first option: without a unified interface, it is hard to combine and compare different algorithms. In addition, this option usually requires practitioners to be capable of building the specific architectures for sequential data.

To address these issues, the goal of this thesis is to develop a unified framework for ML with time series. We discuss the design and implementation of that framework in the next chapter. Ultimately, our aim is to make the time series analysis ecosystem more usable and interoperable as a whole.

Chapter 8

sktime: A Unified Framework for Time Series Analysis

This chapter is partly based on:

- Markus Löning et al. 'sktime: A Unified Interface for Machine Learning with Time Series'. In: *Workshop on Systems for ML at NeurIPS 2019* (2019)
- Markus Löning and Franz J. Király. 'Forecasting with sktime: Designing sktime's New Forecasting API and Applying It to Replicate and Extend the M4 Study'. In: *arXiv preprint* (2020)

8.1 Introduction

In this chapter, we describe the design and implementation of `sktime`[1] – a unified software framework for ML with time series. The design of `sktime` is based on the combined insights from the previous chapters, including the taxonomy of time series learning tasks from Chapter 3, our conceptual model of ML theory and scitypes from Chapter 5, as well as our design principles and patterns from

[1]`sktime` is available at `https://github.com/alan-turing-institute/sktime`. The name `sktime` stands for "science toolkit for time series" following a common naming convention in the Python ecosystem, in which multiple "science toolkits", or "scikits" for short, have been developed (e.g. `scikit-learn`). "scikit" is then often further abbreviated to "sk" (e.g. `scikit-learn` is often called `sklearn`).

Chapter 6. `sktime` forms the fourth research contribution of this thesis as laid out in Section 1.6.

As discussed in Section 4.2, we focus on design as a result, rather than design as a process. In terms of overall architecture, `sktime` is organized into modules. There is one module for every supported task, including time series classification, regression and forecasting. Tasks provide a set of cohesive, high-level categories to group functionality, while being part of the unified framework highlights their relations. Additional modules exist for shared functionality such as transformations and time series distances, among others. In the following, we will discuss core modules in greater detail. In particular, we will motivate and describe the design of key elements within these modules, their interfaces and the relations between them.

`sktime` follows the design principles and patterns proposed in Chapter 6. This chapter will illustrate how these principles and patterns can guide practical design. Much of the design is of course not entirely new. As we have seen in previous chapters, our approach draws design ideas from existing toolboxes and classical software design. However, to our knowledge, we are the first to fully motivate and derive a design from general principles and patterns for ML toolbox design.

Since its release in 2019, `sktime` has become one of the most popular open-source Python libraries for ML with time series. It has grown from the initial academic collaboration into a community-driven project with contributors from around the world. At the time of writing, `sktime` has over 80 contributors, 500 forks, 100 dependent projects, and 3.9K stars on GitHub. `sktime` is now used in methodological and applied research for ML with time series as well as commercial and industrial applications.

The rest of this chapter is organized as follows: We start in Section 8.2 by defining `sktime`'s scope. In Section 8.3, we describe the design for primitive (or atomic) algorithms, with a focus on time series classification, regression and forecasting. In Section 8.4, we discuss the design for composite algorithms (e.g. ensembling, pipelining, tuning and reduction). In Section 8.5, we discuss the design of time series transformations. Throughout these sections, we will highlight novel design ideas and illustrate our design choices with code examples from `sktime`. The examples are based on version 0.6.0 of `sktime`. In Section 8.6, we will give an overview of the implemented functionality. At the end of the chapter, we briefly discuss the distribution, community building and development process of `sktime`.

8.2 Scope

In this section, we define the scope of `sktime` in terms of software design, content and performance. We also motivate the choice of Python, the programming language in which we have implemented `sktime`.

Our overall objective is to create a unified framework for ML with time series that supports multiple learning tasks. We want to provide reusable functionality that practitioners can import and use when writing new applications as well as reusable design templates that developers can copy when developing new algorithms. In this thesis, we focus on three key learning tasks, time series classification, regression and forecasting, leaving the design and implementation of functionality for other learning tasks for future work.

8.2.1 Design

Our target audience are data science practitioners capable of basic programming. In terms of design, the objective is therefore to create a consistent and modular API, supported by a conceptual domain model and clear design principles. We wan do this in a programming environment that is accessible to non-ML experts and reusable in various scientific, commercial and industrial applications. This is why we focus on building an API that makes use of the enhanced interactive Python interpreter [206], specifically designed for scientific computing, rather than expending efforts on creating a command-line interface, let alone a graphical user interface (GUI).[2]

In addition, we aim to reuse available Python libraries as much as possible, notably `NumPy` [104], `pandas`, `SciPy` and `scikit-learn` [205]. We want to ensure that `sktime` seem familiar to most practitioners and developers. We aim to follow `scikit-learn`'s syntax and logic whenever possible and to reuse much of their functionality. For other time series libraries, we aim to provide adapters to make them accessible through `sktime`'s unified interface.

8.2.2 Content

In terms of the content, the objective is to cover the key steps in typical modeling workflows. These include: task specification (e.g. the forecasting horizon), model specification including composite model building, model training, model inspection and application, model selection (e.g. hyper-parameter tuning using re-sampling schemes such as cross-validation) and model validation. In this thesis, we focus on (deterministic) point-predictive modeling workflows (see Section 2.2). In future work, we want to add support for explanatory modeling (e.g. time series clustering) and probabilistic predictions.

8.2.3 Performance

In terms of performance, `sktime` is designed for medium-sized data that fits into memory of a single machine. For larger data sets, we intend to integrate existing tools for parallelized and distributed computing over multiple machines (e.g. `Dask` [219]).

8.2.4 Programming language: The case for Python

While the realization of the proposed framework is not bound to a specific programming language, we choose Python for its implementation. Python has a

[2]The interactive Python interpreter, called `ipython` [206], provides a Read-Evaluate-Print-Loop (REPL) programming environment which executes code piecewise. It takes in user input, executes it and returns the result to the user.

number of advantages over other common languages used in data science:

- Python is free and open-source unlike MATLAB and other proprietary software.
- In contrast to more specialized languages such as R, Python is a general-purpose programming language. It offers good support for object-oriented programming and allows to implement many different projects in a single framework. It therefore makes `sktime` more accessible to non-ML experts and reusable in various scientific, commercial and industrial environments. In addition, it keeps setup and dependencies to a minimum and facilitates software exchange and collaboration among researchers.
- Python has becomes one of the most popular programming languages for data science. It has caught up, if not surpassed, other languages such as R or MATLAB. Python has a vibrant community of developers and a continuously growing ecosystem of stable, well-developed libraries for numerical and scientific computing, notably `NumPy` [104], `Matplotlib` [119], `pandas` [185], and `IPython` [206] as well as state-of-the-art ML capabilities. In addition, there is a growing number of tools and web services that streamline the installation process, lowering the barriers to entry for new users and ensuring easier reproducibility. These tools enable easier collaboration on software development, easier writing and sharing of software documentation, and easier testing of software [217].

8.3 Learning algorithms: Primitives

Having illustrated how scitype-based patterns can motivate and explain design choices of existing toolboxes such as `scikit-learn` in Chapter 5, we will now turn to the design of `sktime`. We begin, in this section, by discussing the design of `sktime`'s API for learning algorithms, the central objects in ML workflows. We first discuss primitive algorithms (i.e. those algorithms that do not consist of modular components). In the next section, we turn to composite algorithms, including pipelining, ensembling, tuning and reduction.

Our overall approach is as follows:

1. As argued in Chapter 5, in order to define scitypes for algorithms, we first need a formal problem specification, that is, a learning task. The algorithm scitype is the counterpart to the task specification. While the tasks defines *what* to solve, the algorithm defines *how* to solve it. We make use of the taxonomy of time series tasks defined in Chapter 3.

2. We can then derive the algorithm scitype from the task specification. As discussed in Chapter 5, a scitype defines both an abstract class interface and key statistical properties. Each algorithm scitype is implemented as an abstract base class, following the scitype strategy pattern described in Section 6.3. We discuss universal interface points for learning algorithms below.

3. Specific algorithms of a given scitype are then implemented as concrete classes inheriting from the corresponding abstract base class.

Throughout the rest of this section, we will discuss key scitypes in `sktime`, particularly the APIs for time series classification, regression and forecasting.

8.3.1 Universal interface points for learning algorithms

Table 8.1 describes universal interface points for learning algorithms. Since we conceptualize learning algorithms as entity objects, these are based on a combination of the universal interface points for all conceptual objects listed in Table 6.1 and for entity objects listed in Table 6.2. We make a few remarks:

- As is standard, we have separate interface points for (i) ingesting data and fitting of algorithms (e.g. `fit`) and (ii) the application of the fitted algorithm (e.g. `predict` or `transform` for transformers, as discussed in Section 8.5). Conceptually, the separation between training and application interface points reflects the mathematical distinction between the learning algorithm and the fitted prediction functional as discussed in 3.3. Practically, fitting a model is often computationally expensive, and once we have a fitted model we can use it repeatedly to generate predictions. So, separating the two methods allows to first fit a model and then deploy it, potentially calling `predict` repeatedly, without having to refit it.

- Learning algorithms are conceived of as stateful entity objects. For our purposes, the only interface points that may change the state are training and updating. For example, when calling `fit`, the algorithm's state changes from being "unfitted" (or "initialized") to "fitted". Note that `fit` does not return the prediction functional. Instead, the output is absorbed in the algorithm object as a state change.[3] Other interface points may not change the algorithm's state. But they may of course access it. For example, `predict` may access the algorithm's state, but may not change it. Because some interface points, including `predict`, rely on the fitted state, calling `predict` requires a prior call to `fit`. Calling `predict` on an unfitted object will raise an error. Note that objects with other scitypes than the ones discussed in this thesis may have different interface points for state changes. The number and quality of interface points that may change an object's state, in general, depends on its scitype, as discussed in Section 6.3.1.
- While encapsulation will hide some detail, key information such as state variables (e.g. fitted model parameters) or hyper-parameters are made accessible through a common inspection interface.

In the following, we will discuss the API design for time series classification, regression and forecasting.

[3]Technically, calling `fit` returns `self`, a reference to the fitted algorithm instance, following `scikit-learn` API specification. Returning `self` is necessary to allow method composition, e.g. chaining method calls.

Table 8.1: Universal interface points for learning algorithms

Interface point	Description
Specification	Model construction and specification of hyper-parameter
Training	Data ingestion and parameter estimation (e.g. `fit`)
Application	Application of fitted model, typically transforming data or generating predictions (e.g. `predict`, `transform`)
Updating	Incremental data ingestion and updating of fitted parameters in an online learning setting (e.g. `update` or `partial_fit`)
Inspection	Access to key information such as hyper-parameters and fitted parameters (e.g. `get_params` and `get_fitted_params`)
Persistence	Saving of a fitted model for later deployment (e.g. `save`, `load`)

Notes: The universal interface points for learning algorithms listed in this table are derived from the universal interface points for conceptual models in Table 6.1 and for entity objects in Table 6.2.

8.3.2 Time series classification/regression

We now describe the design of the time series classification and regression API. As we have seen in Section 3.3.1, time series classifiers and regressors are direct extensions of their cross-sectional counterparts to panel data. This is why our API design largely follows `scikit-learn`.

Following the formulation of the time series classification learning task in Section 3.3.1, we can define a scitype for a time series classifier as a combination of a scitype specifying the interface and its statistical properties. The scitype can be define as follows:

```
type TimeSeriesClassifier
  params   paramlist  :  paramobject
  state    model      :  mathobject
  methods  fit        :  $(\text{series}(\mathcal{Z}) \times \mathcal{Y})^M \rightarrow$ model
           predict    :  $\text{series}(\mathcal{Z}) \times$ model $\rightarrow \mathcal{Y}$
```

A `TimeSeriesRegressor` can be defined in the same way, adjusting the domain of the target variable, $\mathcal{Y}$, accordingly. The nature of the panel input data, series($\mathcal{Z}$), implies three key statistical properties. As in the cross-sectional setting (see Section 5.4.3), we assume permutation invariability for the i.i.d. instances. We also still assume variable permutation invariability, where variables now no longer represent scalar values, but entire series. In addition, we now also assume that the

Example 3: `sktime`'s API for time series classification

```
classifier = TimeSeriesForestClassifier(n_estimators=100)
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Notes: `X_train` and `y_train` denotes the panel training data and label vector, respectively, `X_test` the panel test data, and `y_pred` the predicted label.

result of fitting may change if the order of individual (time series) observations within an instance is permuted.

Following the scitype strategy pattern, we can implement an abstract base class from the above scitype. Concrete time series classification algorithms can then inherit the scitype-defining interface from the base class. Example 3 shows `sktime`'s core API for time series classification. The key difference to `scikit-learn`'s API is that `X_train` and `y_train` now represent panel data, not cross-sectional data. We will later describe key composition algorithms for time series classification and regression, but first discuss the forecasting API.

8.3.3 Forecasting

We now describe the design of `sktime`'s forecasting API. The design of the cross-sectional API may have been obvious, especially in hindsight, given the various well established frameworks such `scikit-learn`. Similarly, the time series classification and regression API is relatively straightforward because it is a direct extension of the cross-sectional API. Design of the forecasting API, on the other, is less obvious. Various toolboxes exist, as discussed in Section 7, but none of them has been able to establish a standard API.

As before, we follow the strategy pattern and encapsulate forecasters in classes with a uniform interface. The interface is based on the algorithm scitype which we derive from the forecasting task defined in Section 3.3.2. The scitype can be specified as follows:

Example 4: `sktime`'s API for (univariate) forecasting

```
forecaster = ExponentialSmoothing(trend="additive")
forecaster.fit(y_train)
fh = ForecastingHorizon([1, 2, 3])
y_pred = forecaster.predict(fh)
```

Notes: `y_train` denotes the training series. `fh` denotes forecasting horizon indicating a three step ahead prediction relative to the end of the training data. `y_pred` denotes the series of predicted values.

```
type Forecaster
  params    paramlist  :  paramobject
  state     model      :  mathobject
  methods   fit        :  𝒵 → model
            predict    :  𝒯 × model → 𝒵
```

The time series nature of the input data implies two key statistical properties:

- Orderedness of time series observations, meaning that shuffling the observations may change the results of `fit`,
- A shared domain between the observed time points in $\mathcal{Z}$ and the forecasting horizon $\mathcal{T}$.

Example 4 shows how the scitype is reflected in `sktime`'s API. We discuss key design decisions in more detail below.

Model specification interface

Following the design principle of separating model specification and data specification (see Section 6.2), the constructor takes in only hyper-parameters but no data. This is in line with `scikit-learn`'s API specification, but different from `statsmodels`' API [207], where the training data has to be specified during model construction.

Training interface

The algorithm's training interface can be used to ingest the training data and fit its parameters. In addition to the training data, all forecasters in `sktime` accept the forecasting horizon already at training time. This is because some models require the forecasting horizon already during fitting. This includes all those models that fit a separate set of parameters for each time point in the horizon, as discussed in Section 3.3.2. A common example is the "direct" reduction strategy from forecasting to cross-sectional regression in which a separate cross-sectional regressor is fitted for each time point in the horizon [34]. This design choice enables us to include a wide variety of forecasting algorithms with a uniform interface. If a forecasting horizon is passed to a model that does not need it during fitting, the horizon is simply stored and retrieved during prediction.

Task specification: The forecasting horizon

The forecasting horizon specifies the time points we want to predict. The forecasting horizon can be represented in a number of ways. For example, as the total number of time points that we want to predict, relative to the last point of the training data (i.e. the "cutoff point") as in `pmdarima`. Or as a set of absolute time points as in `fbprophet`. Or as an interval between start and end points as in `statsmodels`. Each representation has advantages and disadvantages.

A relative horizon, as opposed to an absolute one, has the advantage that it allows to update the predictions, as time moves on, without having to update the forecasting horizon. A disadvantage of relative horizon is that in-sample horizons are specified as negative steps, going backwards from the end of the training series, which may be counter-intuitive. Note that we want to avoid having a separate interface point for in-sample prediction, as we will discuss below. Another disadvantage is that forecasters have to keep track of the cutoff point as the reference point for the forecasting horizon. Representing the forecasting horizon as an interval of time points or simply the number of steps ahead has the advantage of convenience, but may not be precise enough. This is because some forecasters may fit separate parameters for each time point in the horizon. To avoid needless computation, it is necessary to specify the exact time points we want to predict.

In `sktime`, we support two representations and conversions between them: a set of integer steps, relative to the cutoff point and a set of absolute time points. The `ForecastingHorizon` class shown in example 4 provides the necessary functionality to support different representations and conversions between them. Supporting multiple representations is crucial for creating a unified interface in which we interface other libraries with different representations. Note that, in the example, we provide the task information explicitly to the algorithm using the `fh` object. Following the co-strategy pattern described in Section 6.3, we define the `ForecastingHorizon` class to carry that information. By contrast, in the previous examples of the cross-sectional and time series classification API, the task information is provided implicitly. In those setting, the key information is the reference to the target variable, which in `scikit-learn`'s API is given implicitly by passing the target variable and feature data as separate input arguments. Note that both designs adhere to the design principle of keeping the task specification separate from model specification, as described in Section 6.2.

Also note that because forecasting algorithms need to keep track of the forecasting horizon and cutoff point, the internal state of forecasters is more complex than that of cross-sectional or time series regressors or classifiers which only need to keep track of model parameters.

Prediction interface

For a given forecasting horizon, the fitted algorithm can be used to generate predictions. In forecasting, there are two ways of generating predictions. We call them "fixed-cutoff" and "moving-cutoff" predictions. Fixed-cutoff predictions are generated at a single, fixed cutoff point. Moving-cutoff predictions are generated by recursively making fixed-cutoff predictions and moving the cutoff point forward. For example, out-of-sample predictions involve fixed-cutoff predictions with potentially multi-step ahead forecasting horizons. In-sample predictions, on the other hand, involve moving-cutoff predictions, iterating over all time points in the training data and repeatedly making single-step-ahead fixed-cutoff predictions.

Since in-sample and out-of-sample predictions require different ways of generating predictions, one may be tempted to expose them via two separate interface points, as for example done in `pmdarima`. However, exposing only a single method and differentiating between in-sample and out-of-sample predictions internally depending on the given forecasting horizon has a key advantage: clients – be it users or other objects – can always rely on a single method and simply delegate the decision whether to generate in-sample or out-of-sample forecasts to the algorithm. For example, by exposing a single method, composite forecasters can simply forward the forecasting horizon to its component forecasters, without having to distinguish between different method calls. We discuss detrending as a concrete example of a composite algorithm that involves in-sample and out-of-sample predictions in Section 8.5.3.

Update interface: Online learning

In addition to fitting, we introduce an interface point for incrementally updating a fitted forecaster as new data comes in. Updating has two purposes: First, it keep track of the cutoff point, always moving it to the last time point of the available data, as new data is ingested. Predictions are therefore always generated relative to the most recent time point. Second, it provides an interface point for implementing online learning algorithms to update a model's fitted parameters incrementally without having to refit the entire model. This is important in situation where refitting the whole algorithm is too costly and more efficient online learning algorithms are available.[4]

Inspecting fitted algorithms

In addition to the common hyper-parameter interface provided by `scikit-learn`, we introduce a new interface for retrieving fitted parameters from algorithms. For example, this allows us to write composition algorithms which rely of fitted

[4]For examples of online learning forecasting algorithms, see e.g. Liu et al. [165] or Anava et al. [6].

parameters of their component models. We discuss feature extraction as an example in Section 8.5.4.

Until now, we have discussed primitive learning algorithms. We now turn to composite algorithms.

8.4 Learning algorithms: Composites

Composite algorithms, sometimes also called “meta-algorithms”, consist of one or more component algorithms. Through composition interfaces, we enable practitioners to build a wide variety of models with a small amount of easy-to-read code. Many algorithms are expressible as composites. `sktime` provides meta-algorithms for standard composition techniques like pipelining, tuning, ensembling and reduction.

The primary pattern that we follow in our design of the API for composite algorithms is the scitype composition pattern as described in Section 6.3. Following this pattern, meta-algorithms both consist of one or more algorithms but are also algorithms themselves. As such, they have the same uniform interface as primitive algorithms. The key advantage of this pattern is that composite and primitive algorithms can be treated uniformly and become interchangeable at run time. As a consequence, users and client algorithms can ignore the difference between primitive and composite algorithms.

In the following, we discuss selected examples of meta-algorithms in `sktime`, highlighting novel algorithms for time series classification, regression and forecasting. As we have argued in Section 6.2, important operations on objects with a scitype should define higher-order scitypes. To express meta-algorithms, we will therefore give examples of higher-order scitypes throughout the rest of this section.

8.4.1 Ensembling

For all series-as-features learning tasks, including time series classification and regression, we can reuse much of `scikit-learn`'s ensembling functionality, as the APIs are sufficiently similar. Below we discuss two examples of ensembling in `sktime`.

Ensembling for forecasting

Following `scikit-learn`, we provide a simple meta-algorithm for ensembling multiple forecasters. We can define the higher-order scitype of the ensembling operation for forecasters as follows:

$$\texttt{Ensembler}\colon (\texttt{Forecaster})^N \to \texttt{Forecaster}.$$

In words, the ensemble algorithm takes in $N \in \mathbb{N}^+$ algorithm instances of type `Forecaster` and ensembles them into a single, composite algorithm of the same type.[5] In the resultant ensemble, during training, each component algorithm is fitted separately to the training data. At prediction time, the ensemble then aggregates the predictions from all components before returning them.

Variable-based ensembling for time series classification/regression

We also add a new ensembling meta-algorithm for multivariate time series classification and regression. This allows us to fit different algorithms on different time series variables in multivariate panel data. This is illustrated in example 5.

8.4.2 Reduction

One of the main reasons for developing a unified framework is reduction, that is, the application of an algorithm for one task to another. Many ML approaches

[5]Technically, this expresses a heterogeneous ensemble where each of the passed algorithm instances may be a different concrete algorithm. Homogeneous ensembles may be expressed as $\texttt{HomogeneousEnsembler}\colon \texttt{Forecaster} \to \texttt{Forecaster}$ where the meta-algorithm would take in an additional argument for the number of component algorithms to use. All component algorithms would then be instance of the same concrete algorithm class.

Example 5: Variable-based ensembling in `sktime`

```python
classifier = ColumnEnsembleClassifier(
  ("tsf", TimeSeriesForestClassifier(), [0]),
  ("boss", BOSSEnsemble(), [1])])
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Notes: `X_train` and `X_test` contain multivariate training and test data, and `y_train` the associated labels. We use the `ColumnEnsembleClassifier` to ensemble two component classifiers over two time series variable (or columns): a `TimeSeriesForest` on the first column (`[0]`) and a `BOSSEnsemble` on the second one (`[1]`) using Python's zero-based indexing. Predictions are averaged over both components.

for time series work through reduction. While reduction is not a new idea, to our knowledge, we are the first to encapsulate them as meta-algorithms, following the composition pattern described in Section 6.3. As discussed in Section 3.4, reductions have a number of key properties. These properties make them well suited to be expressed as meta-algorithms:

- **Adaptive.** Reductions are adaptive. They adapt the interface of the component algorithm to the algorithm scitype required for solving the new task. This enables us to use the model evaluation tools appropriate for the new task.[6]

- **Modular.** Reductions convert any algorithm of a certain scitype to an algorithm of a new scitype. Applying a reduction approach to N algorithms for one task gives N new algorithms for a new task. Any progress on the base algorithm immediately transfers to the new task, saving both research and software development effort [25, 26].

- **Composable.** Reductions are composable. For example, we can first reduce forecasting to time series regression which in turn can be reduced to

[6]Note that it is common to express reductions as transformers rather than composite estimators. This works if the interface does not change from the original to the new task. For example, the interfaces of time series and cross-sectional classifications in `sktime` and `scikit-learn` are sufficiently similar that the reduction can be expressed through a transformer. However, for other reduction approaches where the interface does change, it is preferable to use composite algorithms. The reason is that composite algorithms can adapt the interface, whereas transformers cannot.

Example 6: Reducing forecasting to cross-sectional regression in `sktime`

```python
regressor = RandomForestRegressor()
forecaster = make_reduction(regressor,
  window_length=10, strategy="recursive")
forecaster.fit(y_train)
y_pred = forecaster.predict(fh=1)
```

Notes: `RandomForestRegressor` comes from `scikit-learn`. `make_reduction` a factory method that return the appropriate reduction meta-algorithm for the given hyper-parameters. We here specify additional hyper-parameters, including the window length used for the sliding window transformation and "recursive" strategy to generate predictions, as described in Section 3.4. `y_train` denotes the training series. `y_pred` denotes the series of predicted values for the given single-step ahead forecasting horizon.

cross-sectional regression via feature extraction.

- **Tunable.** Most reductions require modeling choices that we may want to optimize or tune. By expressing reductions as meta-algorithms, we expose these choices via the common hyper-parameters interface and make them tunable.

Without a unified framework, reductions are often hand-crafted – the code used in the M4 competition [182] being one example. The consequence is that they are neither modular, tunable, composable nor adaptive. Below we return to the example of reducing a forecasting task to a cross-sectional regression task from Section 3.4 and illustrates how this works in `sktime`.

Reducing forecasting to cross-sectional regression

A common approach is to solve forecasting via cross-sectional regression. The higher-order scitype for this reduction approach can be expressed as follows:

$$\texttt{Reducer}: \texttt{CrossSectionalRegressor} \rightarrow \texttt{Forecaster}.$$

In words, the reduction algorithm takes in an algorithm of scitype `CrossSectionalRegressor` and returns an algorithm of scitype `Forecaster`. We explain how the resultant composite algorithm works internally in Section 3.4. Example 6 shows what reduction to cross-sectional regression looks like in `sktime`.

Other reduction approaches are of course possible. For example, `sktime` also provides a meta-algorithm for reduction to time series regression, so that any of `sktime`'s time series regressors can be used to solve a forecasting task. We illustrate the effectiveness of reduction-based ML models in our extension of the M4 forecasting competition in Chapter 9.

8.4.3 Model selection

Model selection, or tuning, is another example of a modular composite algorithm. For all series-as-features learning tasks, including time series classification and regression, we can reuse `scikit-learn`'s tuning functionality, as the APIs are sufficiently similar.

For forecasting, `sktime` provides its own tuning class. We can express the higher-order scitype for tuning as follows:

$$\texttt{Tuner}: \texttt{Forecaster} \to \texttt{Forecaster}.$$

where the tuning meta-algorithm takes in an algorithm of type `Forecaster` and returns an algorithm of the same type. At training time, the resultant algorithm will tune the component algorithm for a given set of tuning instructions, including the cross-validation scheme, the grid of hyper-parameters to search over, and the loss function for selecting the best performing model. For example, it may perform temporal grid-search cross-validation for a given grid of parameter names and values. At prediction time, it will use the model with the best performing set of hyper-parameters to generate predictions. Example 7 shows how tuning works in `sktime`.

Following the co-strategy pattern described in Section 6.3, we express the set of tuning instructions as objects in their own right. The `SlidingWindowSplitter` is a cross-validation iterator and encodes the way how to split the training series into sliding windows.

Example 7: Temporal grid-search cross-validation in `sktime`

```
forecaster = make_reduction(RandomForestRegressor(),
    window_length=3)
param_grid = {"window_length": [3, 5, 7]}
cv = SlidingWindowSplitter()
gscv = ForecastingGridSearchCV(forecaster, param_grid, cv)
gscv.fit(y_train)
y_pred = gscv.predict(fh=1)
```

Notes: As before, `RandomForestRegressor` comes from `scikit-learn`. `make_reduction` is the reduction meta-algorithm discussed in example 6. `param_grid` denotes the grid of hyper-parameters to search over for the window length parameter. `cv` denotes the cross-validation iterator which encodes the sliding-window cross-validation scheme used to split the training data. `ForecastingGridSearchCV` is the tuning meta-algorithm. `y_train` denotes the training series. `y_pred` denotes the series of predicted values for the given single-step ahead forecasting horizon.

Example 8: Forecasting pipeline in `sktime`

```
forecaster = TransformedTargetForecaster([
  ("deseasonalize", Deseasonalizer(sp=12)),
  ("forecast", NaiveForecaster(strategy="last"))
])
forecaster.fit(y_train)
y_pred = forecaster.predict(fh=1)
```

Notes: `TransformedTargetForecaster` is the pipelining meta-algorithm. We here specify a pipeline consisting of a `Deseasonalizer` transformer to deseasonalize the training data and a `NaiveForecaster` forecasting algorithm. We set the seasonal periodicity (`sp`) to 12, indicating monthly data and the strategy of the `NaiveForecaster` to always predict the last value observed in the training series. `y_train` denotes the training series. `y_pred` denotes the series of predicted values for the given single-step ahead forecasting horizon.

8.4.4 Pipelining

One of most common composite algorithms is pipelining. In its simplest form, a pipeline chains multiple data transformations with a final learning algorithm. We first discuss pipelining here, and then turn to data transformations in the next section.

Again, for series-as-features tasks, we can reuse `scikit-learn`'s functionality. For forecasting, `sktime` provides its own pipeline meta-algorithm. The higher-order scitype of a pipelining meta-algorithm can be expressed as follows:

$$\texttt{Pipelining: (SeriesToSeriesTransformer)}^N \times \texttt{Forecaster} \rightarrow \texttt{Forecaster}.$$

In words, pipelining takes in N algorithms of type `SeriesToSeriesTransformer`, which we will define below in Section 8.5.3, and a single algorithm of type `Forecaster` and returns a single composite algorithm of type `Forecaster`.

When fitting the resultant composite forecaster, the data is first transformed by the transformers and then passed to the final forecaster. At prediction time, the order of operations is reversed. The composite forecaster first generates predictions for a given forecasting horizon. The predicted values are then inverse-transformed by the transformers before being returned.

Example 8 illustrates how a pipeline can be specified in `sktime`. In particular, it shows how the "naive2" model[7] from the M4 forecasting competition can be expressed as a pipeline consisting of a deseasonalization transformer and a naive forecaster. Since the transformers work on the target series to be predicted, we follow `scikit-learn` in calling this meta-algorithm `TransformedTargetFore-caster` rather than `Pipeline`. In this example, we use a single transformer. One can of course chain together multiple transformers.

So far we have discussed primitive and composite learning algorithms. In the next section, we turn to transformers.

8.5 Transformations

In general, learning algorithms can be split into two categories: predictive and transformative algorithms. Both algorithms are fittable to data. But while predictive algorithms generate predictions, transformative algorithms apply transformations to data. Until now we have discussed predictive algorithms, both primitive and composite ones. In this section, we will discuss transformative algorithms. As is standard, we call transformative algorithms "transformers" and continue to call predictive algorithms "learning algorithms".

Within ML modeling workflows, data object are usually conceived of as an immutable value object as discussed in Chapter 5. So, rather than keeping track

[7]The "naive2" model is described in more detail in Table 9.3.

of changes to data in the data object itself, we treat data transformations as part of the model specification. For this purpose, we encapsulate them as transformers.

Unlike the predictive learning algorithms discussed so far, transformers by themselves do not correspond to any learning task. We can, however, still define scitypes for transformers in relation to the learning task that they are applied to. In terms of scitypes, we do this by distinguishing between "primary scitypes" that directly address a task and "secondary scitypes" which arise as operations related to primary scitypes. For example, in the context of cross-sectional learning pipelines, transformers need to be compatible with the established scitype for cross-sectional learning algorithms. In particular, this implies adherence to the universal interface points listed in Table 8.1, including separate methods for fitting and application of the fitted model (i.e. `predict` and `transform`, respectively) as well as the common hyper-parameter interface. In this sense, transformers can be conceived of as modifications to learning algorithms together with the pipelines that they are part of.

As we have seen in Section 3.2, time series data can come in different forms, including univariate, multivariate and panel data. In `sktime`, we distinguish between the following types of transformation in our API design, depending on the signature of their `transform` method:

- Series to scalar,
- Series to series,
- Panel to cross-sectional,
- Panel to panel.

Before we discuss key examples of each transformation type in more detail, we review the cross-sectional API design for transformers in `scikit-learn`.

Example 9: `scikit-learns`'s API for cross-sectional transformers

```
transformer = PCA()
transformer.fit(X, y)
X_transformed = transformer.transform(X)
```

Notes: `PCA` is from `scikit-learn`. `X` and `y` denote cross-sectional feature data and target vector. `X_transformed` denotes the transformed data.

8.5.1 Cross-sectional transformations

Example 9 shows the cross-sectional transformer API in `scikit-learn`. We can see how the transformer API in `scikit-learn` implements the following scitype for cross-sectional transformations:

```
type CrossSectionalTransformer
  params   paramlist  :  paramobject
  state    model      :  mathobject
  methods  fit        :  $(\mathcal{X} \times \mathcal{Y})^N \rightarrow$ model
           transform  :  $\mathcal{X} \times$ model $\rightarrow \mathcal{U}$
```

This is a cross-sectional transformer scitype, given the cross-sectional domain of the input data $\mathcal{X}$ and transformed output data $\mathcal{U}$, defined in Section 2.5. As a consequence, the same statistical properties hold as for the `CrossSectionalClas-sifier` or `CrossSectionalRegressor` scitype discussed in Chapter 5.

In the following, we discuss `sktime`'s API design for time series transformations.

8.5.2 Series-to-scalar transformations

The simplest time series transformation type is the series-to-scalar transformation. As the name suggests, it transforms a time series to a scalar (e.g. a number or string). The series can be univariate or multivariate. If it is multivariate, the transformation will return a scalar for each variable. Examples of series-to-scalar transformations include extracting the mean or some percentile from the values of a time series.

In terms of API design, these transformations are typically defined as functions, not classes. Encapsulating them in classes is not necessary since these transformation are deterministic. They do not involve any fittable parameters or state changes which we need to be managed.

However, in the context of pipelines, these transformations can still involve modeling choices which we may wish to tune (e.g. the exact percentile to extract). To expose these choices as tunable hyper-parameters in pipelines, we can use `scikit-learn`'s `FunctionTransformer`, which adapts a function to the typical transformer interface, including the common interface for hyper-parameters.

8.5.3 Series-to-series transformations

Series-to-series transformations take in a series, either univariate or multivariate, and return a series. Examples include taking the logarithm, the Box-Cox transform [37] or the Fourier transform. However, not all of these transformation are deterministic. Some may involve fittable parameters. Examples of transformation with fittable parameters include extracting auto-regressive coefficients from a series, in which one first fits an auto-regressive model to the data and then returns its fitted parameters as used in composite time series classification algorithms like RISE [163], or residual detrending as discussed below.

In terms of API design, as before, if a transformation is deterministic, we can simply define it as a function and use the `FunctionTransformer` in pipelines. Otherwise, we encapsulate it in a class. The scitype of series-to-series transformers is given by:

```
type SeriesToSeriesTransformer
  params   paramlist  :  paramobject
  state    model      :  mathobject
  methods  fit        :  series($\mathcal{Z}$) $\rightarrow$ model
           transform  :  series($\mathcal{Z}$) $\times$ model $\rightarrow$ series($\mathcal{V}$)
```

Here, series($\mathcal{V}$) denotes the domain of the transformed series. Both the value and time domain may be transformed. For a example, a Fourier transform transforms the time domain into a frequency domain. An outlier detector may transform a real-valued input series into a binary categorical or boolean series indicating whether or not a time point is an outlier.

Example 10: Residual detrending in `sktime`

```
forecaster = PolynomialTrendForecaster(degree=1)
transformer = Detrender(forecaster)
transformer.fit(y_train)
y_transformed = transformer.transform(y_train)
```

Notes: `Detrender` and `PolynomialTrendForecaster` come from `sktime`. We here specify a polynomial of degree 1 (i.e. linear). `y_train` denotes the training data. `y_transformed` denotes the transformed data. We here use the same data in `fit` and `transform`, but in-principle different series could be passed.

Example: Residual detrending

As an example, we discuss a new modular transformer in `sktime` for residual detrending. At construction time, the transformer takes in a forecaster. When calling `fit`, the transformer fits the forecaster to the input data. When calling `transform`, it first generates predictions from the fitted forecaster using the time index of the input data as the forecasting horizon, then returns the residuals of the predictions, where the residuals are typically computed by subtracting the predicted values from the input series. Depending on the input series passed to `fit` and `transform`, this requires to generate in-sample or out-of-sample forecasts. In terms of higher-order scitypes, the meta-transformer can be defined as follows:

$$\texttt{Detrending}\colon \texttt{Forecaster} \to \texttt{SeriesToSeriesTransformer}.$$

Example 10 illustrates how we can use such a detrending transformer in `sktime` to remove a linear trend from a time series. In the example, we use a linear trend forecaster, but in-principle the `Detrender` works with any forecaster, including non-linear ones. In a pipeline, the `Detrender` can be seen as a form of one-step residual boosting, by first detrending a time series and then fitting another forecaster on the residuals [241]. We investigate the potential of boosting a statistical method with an ML model in this way in our extension of the M4 forecasting competition in Chapter 9.

8.5.4 Panel-to-cross-sectional transformations

Panel-to-cross-sectional transformations take in panel data, either univariate or multivariate, and return cross-sectional data. A common example is feature extraction as provided by libraries like `tsfresh` [55, 56]. As we will see below, one can think of feature extraction in this setting as a instance-wise composite transformer, in which a series-to-scalar transformer is applied to all instances (series) in the panel data.

In terms of API design, these transformations are usually encapsulated in transformer classes. We can define the scitype as follows:

```
type PanelToCrossSectionalTransformer
  params   paramlist  :  paramobject
  state    model      :  mathobject
  methods  fit        :  $(\text{series}(\mathcal{Z}))^M \rightarrow$ model
           transform  :  $\text{series}(\mathcal{Z}) \times$ model $\rightarrow \mathcal{U}$
```

As before, $\mathcal{U}$ here denotes the cross-sectional domain of the transformed series. As in the cross-sectional case, input and output data always has the same number of instances, but may have different numbers of variables. The time index may also be changed during transformation. Below we discuss an example of feature extraction in `sktime`.

Example: Using a forecaster for feature extraction

Forecasting algorithms cannot only be used to solve forecasting tasks, but also to solve other related learning tasks. A common approach is to use forecasters as a feature extractors for solving tasks such as time series classification. This approach involves the iterative application of a series-to-scalar transformation to each instance (series) of the panel data. As such, it amounts to a reduction from time series to cross-sectional classification. In particular, a forecaster is first fitted to each instance of the available panel data, its fitted parameters are then returned as scalar features. There are both bespoke algorithms which make use of this approach (see e.g. RISE [14] for time series classification) and toolkits such

Example 11: Feature extraction for cross-sectional classification in `sktime`

```
forecaster = ARIMA()
classifier = Pipeline([
  ("extract", FittedParamExtractor(
      forecaster, params=["ar_params"])),
  ("classify", RandomForestClassifier())
])
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Notes: `ARIMA` comes from `sktime`. `Pipeline` and `RandomForestClassifier` come from `scikit-learn`. `X_train` and `y_train` denotes the panel training data and label vector, respectively, `X_test` the panel test data, and `y_pred` the predicted label.

as `tsfresh` [55, 56]) which allow to automatically extract numerous features from time series, including fitted parameters from forecasters, however not in a modular fashion.

`sktime` provides a modular feature extraction transformer, which is a meta-algorithm that extracts the fitted parameters from a forecaster. Example 11 shows how `sktime`'s implementation of this transformer, called the `FittedParamExtractor`, can be used as part of a pipeline for time series classification. Note that this transformer relies on the uniform interface for inspecting fitted model parameters, as listed among the universal interface points in Table 8.1.

8.5.5 Panel-to-panel transformations

The last transformation type that we discuss is panel-to-panel transformation. These transformations take in panel data and return panel data. Examples include variable concatenation as discussed below and composite transformation such as the iterative application of series-to-series transformation to each instance of the panel data (e.g. taking the logarithm of all instances (series) in panel data). We can define the scitype for this type of transformation as follows:

`type PanelToPanelTransformer`			
`params`	`paramlist`	:	`paramobject`
`state`	`model`	:	`mathobject`
`methods`	`fit`	:	$(\text{series}(\mathcal{Z}))^M \rightarrow$ `model`
	`transform`	:	$\text{series}(\mathcal{Z}) \times$ `model` $\rightarrow \text{series}(\mathcal{V})$

Example 12: Reducing multivariate time series classification to univariate time series classification via variable concatenation in `sktime`

```python
classifier = Pipeline([
  ("concatenate", ColumnConcatenator()),
  ("classify", TimeSeriesForestClassifier(n_estimators=100)),
])
classifier.fit(X_train, y_train)
y_pred = classifier.predict(X_test)
```

Notes: `TimeSeriesForestClassifier` and `ColumnConcatenator` come from `sktime`. `Pipeline` comes from `scikit-learn`. `X_train` and `y_train` denotes the panel training data and label vector, respectively, `X_test` the panel test data, and `y_pred` the predicted label.

As in the cross-sectional case, input and output data always has the same number of instances, but may have different numbers of variables. The time index may also be changed during transformation.

Example: Variable concatenation for multivariate time series classification

A simple way to solve multivariate time series classification tasks is to reduce it to a univariate time series classification task. One way to achieve this reduction is to concatenate all time series variables into a single, long time series, ignoring the fact that the series represent different kinds of measurements. Example 12 illustrates how one can reduce a multivariate time series classification problem to a univariate time series classification problem by concatenating all time series variables into a single, long time series.

Having discussed `sktime`'s API design for learning algorithms and transformers, we will now give a brief overview of the implemented functionality in `sktime`.

8.6 Overview of implemented functionality

In this section, we will give an overview of the implemented functionality in `sktime` and its dependencies. A more detailed and up-to-date API reference is available in `sktime`'s online documentation.[8] As we will discuss in Section 8.7, much of the

[8] `https://www.sktime.org/en/latest/`

available concrete functionality in `sktime` has been implemented by open-source contributors using the API specification and design templates presented above.

8.6.1 Transformations

Various transformers have been implemented. These include:

- **Data reshaping.** Transformers for padding, truncating, variable concatenation (see example 12) and interpolating.
- **Feature extraction.** A modular transformer for extracting fitted parameters from forecaster (see example 11), an adapter for `tsfresh` [55, 56] transformer for automatic feature extraction, and a matrix-profile transformer based on `STUMPY` [153].
- **Standardization.** A modular detrending transformer (see example 10), a deseasonalization transformer as well as other common transformations such as logarithms and the Box-Cox transform [37].
- **Adapters.** An adapter to apply any transformer from `scikit-learn` to single series and to apply our single-series transformers to panel data for time series classification and regression.

8.6.2 Time series classification/regression

The time series classification module contains various state-of-the-art time series classification algorithms. The majority of the algorithms has been tested for correctness against reference implementations in the Java-based `tsml` library [14].[9] They have also been used to reproduce key results from comparative benchmarking studies for time series classification [12, 13, 15]. The implemented algorithms include:

[9] `tsml` is a Java-based framework, built on top of `Weka`, and provides algorithms and composition tools for time series classification and clustering. For more details, go to `https://github.com/uea-machine-learning/tsml/`.

- **Interval based.** A modular time series forest (TSF) [71] classifier, a TSF regressor which support configurable time series trees, the random interval spectral ensemble (RISE) [163], and the canonical interval forest (CIF) [189].
- **Distance based.** Distance measures form a fundamental primitive of many time series tasks. Eight distance measures have been implemented using Cython [21] and `numba` [150] for enhanced performance and several classifiers that use them, including the Elastic Ensemble [162], Proximity Forest [174] as well as the ROCKET [68] and MiniROCKET transform [69].
- **Shapelet based.** The shapelet module includes an implementation of the shapelet transform [35] and MrSEQL classifiers [154].
- **Dictionary based.** The symbolic aggregate approximation (SAX) [159] and symbolic Fourier approximation (SFA) [224] transform and the associated bag of patterns (BOP) [160], and bag of SFA symbols (BOSS) [223].
- **Deep learning.** A recent review paper described nine deep learning algorithms for time series classification [131]. These algorithms have been ported into `sktime-dl` [232], our deep learning extension package.
- **Composition.** `sktime`'s compatibility with `scikit-learn` enables us to re-use `scikit-learn`'s functionality for common composition techniques, like pipelining, ensembling and tuning. In addition, variable-based ensembling functionality (see example 5) has been implemented.

8.6.3 Forecasting

The forecasting module includes a number of basic algorithms and provides adapters to various existing specialized libraries as discussed in the review of related software in Chapter 7. The majority of these algorithms has been used to reproduce key results from comparative benchmarking studies for forecasting [167], as discussed in Chapter 9. The implemented algorithms include:

- **Baselines.** Algorithms for simple rule-based or naive strategies and polynomial trend extrapolation.

- **Adapters.** Adapters have been implemented for forecasting algorithms in `statsmodels` [207], `pmdarima` [233], `fbprophet` [247] and `hcrystalball` [106], following the facade pattern in Gamma et al. [90]. They enable us to use algorithms from these packages from within the unified API provided by `sktime`.

- **Composition.** Meta-algorithms for pipelining, ensembling, reduction and tuning have been implemented. In addition, temporal cross-validation functionality has been included (e.g. iterators for using sliding-window and expanding-window schemes). Modular reduction meta-algorithms for forecasting have also been implemented. They allow us to use any cross-sectional regressor from `scikit-learn` and any time series regressor from `sktime` for forecasting (see example 6).

- **Deep learning.** Our extension package `sktime-dl` [232] includes a few standard neural network architectures for forecasting, including simple RNN and LSTM network architectures [109, 221].

8.6.4 Dependencies

`sktime` (version: 0.6.0) requires Python 3.6, 3.7 or 3.8, and a small number of core dependencies: `NumPy` [104] and `pandas` [185] for data handling; `SciPy` [100, 137], `scikit-learn` [46, 205, 256] and `statsmodels` [207] for modeling; Cython [21], `numba` [150] and `joblib`[10] for more efficient computations.

All of the above are hard dependencies. They are required for installing `sktime`. In addition, we interface a number of specialized time series libraries as soft dependencies. These dependencies are optional and only required for using certain functionality, including: `pmdarima` [233] for the Auto-ARIMA algorithm,

[10] `https://github.com/joblib/joblib`

STUMPY [153] for matrix profile computations, tsfresh [55, 56] for automatic feature extraction, and fbprophet [247] for the Prophet forecasting algorithm, among others.

For deep learning, sktime has a companion package, called sktime-dl [232], based on TensorFlow [1] and Keras [54].

8.7 Distribution, documentation and open-source development

In this section, we briefly discuss how sktime is distributed, documented and developed.

8.7.1 Distribution

sktime is distributed under a permissive BSD-3-clause license.[11] sktime is available via PyPI[12] and conda-forge[13] with precompiled files for Windows, MacOS and Linux for easy installation on different operating systems.

8.7.2 Documentation

A detailed and versioned documentation is available online via Readthedocs.[14]. We also provide interactive tutorials on Binder [49] allowing users to run sktime online without having to install anything.

[11]See Lin et al. [158] for an overview of open-source licenses. Also see the Open Source Initiative at https://opensource.org/docs/definition.html. Different understandings of open-source are also often summarized under the Free-Libre-Open-Source Software (FLOSS) acronym.

[12]https://pypi.org

[13]https://conda-forge.org/

[14]sktime's documentation is hosted at https://sktime.org. To find out more about ReadtheDocs, go to https://readthedocs.org

8.7.3 Open-source development

The basic idea of open-source development is simple: everyone with an internet connection can read, modify and redistribute the source code of a piece of software. They can also suggest changes to the source code and contribute their opinion when changes to code, governance or any other aspect of the project are discussed [89].

Solutions to the developer's problem, as described in Section 4.2, often need to involve a wider community of practitioners and developers, which requires building and sustaining an open-source community and governance structures with clear pathways for contributing and reporting bugs, discussion forums, and guidelines for community participation and decision-making [11]. The `sktime` project has created clear guidelines for participation and decision making. We follow a consensus-seeking approach with an community steering committee to oversee the development process and avoid deadlocks.

Development and most discussions take place on GitHub.[15] We use continuous integration[16] services for unit testing on Windows, MacOS and Linux platforms as well as automatic code quality checks for all contributions in order to ensure a consistent style and best practices for code throughout our code base.

8.8 Concluding remarks

In this chapter, we described the design and implementation of `sktime` – the first unified framework for ML with time series. `sktime` constitutes the fourth research contribution of this thesis as laid out in Section 1.6. Much of the design was guided by our taxonomy of time series learning tasks put forward in Chapter 3, the conceptual model of ML theory and scientific types developed in Chapter 5,

[15]`https://github.com/alan-turing-institute/sktime`

[16]Continuous integration is a software development practice where members of a team integrate their work frequently, usually each person integrates at least daily – leading to multiple integrations per day. Each integration is verified by an automated build and test to detect integration errors as quickly as possible.

as well as our design principles and patterns described in Chapter 6.

With `sktime`, we have illustrated how our domain-driven approach to the developer's problem cannot only explain key aspects of existing frameworks like `scikit-learn`, but can also guide the development of new frameworks. While this chapter largely focused on API design, we also gave an overview of the concrete functionality already implemented `sktime`. Finally, we described `sktime`'s distribution, documentation and development process.

After having reviewed the time series analysis ecosystem and its limitations in Chapter 7, we hope to have shown in this chapter that `sktime` is capable of addressing many of the current limitations. In particular, we believe that the unified framework can reduce confusion between different learning tasks, enable practitioner to fully leverage reduction relations between tasks by integrating with more foundational frameworks such as `scikit-learn`, and ultimately make the existing ecosystem more usable and interoperable as a whole.

In the next and final part of this thesis, we will use `sktime` to reproduce and extend the M4 competition, one of the major comparative benchmarking studies for forecasting.

Part III

Comparative Benchmarking of Algorithms

Chapter 9

Reproducing and Extending the M4 Forecasting Competition

This chapter is partly based on:

- Markus Löning and Franz J. Király. 'Forecasting with sktime: Designing sktime's New Forecasting API and Applying It to Replicate and Extend the M4 Study'. In: *arXiv preprint* (2020)

9.1 Introduction

In the previous part, we have developed `sktime`, a unified framework for ML with time series in Python. In this final part of the thesis, we use `sktime` to reproduce and extend the M4 forecasting competition. The reproduction and extension of the M4 competition is the fifth and final research contribution of this thesis as laid out in Section 1.6.

Reproducing the M4 competition will allow us to do three things: test our framework for correctness by comparing it against reference implementations, illustrate its effectiveness and applicability, and validate the published results. Extending the M4 competition will allow us to implement and evaluate previously unstudied algorithms and investigate whether simple, reduction-based ML algorithms can match the state-of-the-art performance of the custom-built algorithms that won the M4 competition. To this end, we use `sktime` to re-implement key algorithms included in the competition and add reduction-based models that are easily implementable in `sktime`. To our knowledge, this is the first complete reproduction of the M4 competition within a single framework independent of the published code. [1]

There is a long history of empirical comparison of forecasting algorithms. For a review, see Hyndman [129]. The M4 competition [176, 181, 182] is the penultimate competition in an influential series of forecasting competitions organized by Spyros Makridakis since 1982 [183]. The latest edition, the M5 competition, was recently carried out on Kaggle.[2] Previous competitions include one on energy demand [116], one on tourism data [10], and the M3 competition [3, 63, 179]. Several articles have reviewed the competition results, many of which were published as part of a special issue of the International Journal of Forecasting on the M4 competition [33, 85, 94, 180, 240]. For the M5 competition, a new special issue is forthcoming. Additional reviews have been published with a focus on ML and deep learning models [107, 270, 271].

9.2 Problem statement

The problem we are trying to solve is the univariate forecasting learning task defined in Section 3.3.2. In brief, the task is to use a univariate time series $\mathbf{y} = (y(t_1) \dots y(t_T))$, with discrete, equidistant time points, observed up to time point t_T to find a forecaster $\hat{f}$ which can make accurate temporal forward predictions $\hat{y}(h_j) = \hat{f}(h_j)$ for the given forecasting horizon $h_1 \dots h_H$, where H denotes the length of the forecasting horizon.

Note that we focus on models that use only a single series for training. By contrast, many of the ML models submitted to the M4 competition require multiple

[1] Note that there is some ambiguity in the term "reproducibility" [7, 76, 96, 213, 225]. We here mean to recreate the results of the M4 competition by running the same analysis on the same data using software code which we developed independently of the code used in the original study.

[2] `https://www.kaggle.com/c/m5-forecasting-accuracy/overview`

series for training, so-called "cross-learning" approaches [227]. We only consider (deterministic) point predictions in this thesis, leaving probabilistic forecasting for future work.

9.3 Evaluation

In order to evaluate predictive accuracy, the M4 competition uses two performance metrics[3], mean absolute scaled error (MASE) and symmetric mean absolute percentage error (sMAPE), which are defined as follows:

$$\text{sMAPE} = \frac{200}{H}\sum_{i=1}^{H}\frac{|y(h_i) - \hat{y}(h_i)|}{|y(h_i)| + |\hat{y}(h_i)|} \tag{9.1}$$

$$\text{MASE} = \frac{1}{H}\sum_{i=1}^{H}\frac{|y(h_i) - \hat{y}(h_i)|}{\frac{1}{T+H-m}\sum_{j=m+1}^{T+H}|y(t_j) - y(t_{j-m})|} \tag{9.2}$$

The denominator of MASE denotes the naive seasonal in-sample forecasts, described as the "sNaive" model in Table 9.3 below, where m denotes the seasonal periodicity of the data (i.e. the periods per year, e.g. 12 for monthly data). We scale the error in the numerator by 200 to express the error as a percentage. MASE and sMAPE are scale-independent and therefore appropriate for comparing predictions across series with different scales [125].

To rank models, the M4 competition additionally uses the OWA (overall weighted average) metric. Unlike MASE and sMAPE, OWA is a metric aggregated over multiple series, given by:

$$\text{OWA} = \frac{1}{2}\left[\frac{\frac{1}{N}\sum_i^N \text{sMAPE}_i}{\frac{1}{N}\sum_i^N \text{sMAPE}_{i,\text{Naive2}}} + \frac{\frac{1}{N}\sum_i^N \text{MASE}_i}{\frac{1}{N}\sum_i^N \text{MASE}_{i,\text{Naive2}}}\right] \tag{9.3}$$

Here N is the number of time series we aggregate over, with the subscript i denoting the index of an individual series. $\text{sMAPE}_{i,\text{Naive2}}$ and $\text{MASE}_{i,\text{Naive2}}$ are the respective metrics on series i for the naive out-of-sample forecasts from the

[3]For an overview of forecasting performance metrics, see e.g. Hyndman and Koehler [125].

"Naive2" model described in Table 9.3.

In addition, we test if found performance differences between models are statistically significant. This extends the previous statistical analysis of the M4 results [147, 182]. In particular, we test whether our reproduced results are significantly different from the published results for the same model using paired t-tests. Note that standard errors are not based on repeated applications of the same model on the same series, but instead on a single application on multiple series. Given the provenance of the M4 data set, we cannot plausibly assume that the series are independent. Consequently, the test results have to be interpreted with caution.

We also check if found performance differences between different models are statistically significant. We use Friedman tests [84] to check whether the found average sMAPE ranks are significantly different from the mean rank expected under the null-hypothesis at the 5% level. We then use post-hoc Nemenyi [199] tests to find those pairs of models that are significantly different, where model pairs are defined by the originally published and reproduced performance estimates. We summarize our findings visually in critical difference diagrams, as proposed by Demšar [70]. To cross-check our results, we also use pairwise Wilcoxon signed rank tests [260] together with Holm's correction procedure for multiple testing [111], as recommended by García and Herrera [91] and Benavoli, Corani and Mangili [22].

9.4 Data

The M4 competition is based on a data set consisting of 100K time series, mainly from business, financial and economic domains [176, 177, 181]. The series are grouped by sampling frequency into yearly, quarterly, monthly, weekly, daily and hourly data sets. Tables 9.1 and 9.2 present summary statistics, showing wide variability in time series characteristics and lengths of the available training series.

Table 9.1: The number of M4 series per sampling frequency and domain

	Demographic	Finance	Industry	Macro	Micro	Other	Total
Yearly	1088	6519	3716	3903	6538	1236	23000
Quarterly	1858	5305	4637	5315	6020	865	24000
Monthly	5728	10987	10017	10016	10975	277	48000
Weekly	24	164	6	41	112	12	359
Daily	10	1559	422	127	1476	633	4227
Hourly	0	0	0	0	0	414	414
Total	8708	24534	18798	19402	25121	3437	100000

Notes: Rows show M4 data sets grouped by sampling frequency. Columns show M4 data sets grouped by domains. Values show the number of available series.

Table 9.2: Summary statistics of the length of time series in the M4 training set

	Mean	Std	Min	25%	50%	75%	Max
Yearly	31.3	24.5	13	20	29	40	835
Quarterly	92.3	51.1	16	62	88	115	866
Monthly	216.3	137.4	42	82	202	306	2794
Weekly	1022.0	707.1	80	379	934	1603	2597
Daily	2357.4	1756.6	93	323	2940	4197	9919
Hourly	853.9	127.9	700	700	960	960	960
Total	240.0	592.3	13	49	97	234	9919

Notes: Rows show M4 data sets grouped by sampling frequency. Columns show summary statistics of the distribution of the length of the training series.

9.5 Technical implementation

Our code for reproducing and extending the M4 competition can be found in our code repository on GitHub.[4] We ran the experiments on machines with Linux CentOS 7.4, 32 CPUs and 189 GB RAM.

For all forecasters and composition tools, we use `sktime` (version 0.4.0). Forecasters are specified as composite models whenever possible, using the composition classes described in Section 8.4. For all regressors except XGB and RNN, we use `scikit-learn` [205]. For XGB, we use `xgboost` [53]. For RNN, we use `skime-dl` [232]. The versions of the libraries used are specified in the code repository.

[4]`https://github.com/mloning/sktime-m4`

9.6 Reproducing the M4 competition

9.6.1 Models

In order to reproduce key results from the M4 competition, we implement and re-evaluate all baseline forecasters of the M4 competition in `sktime`, except the automatic exponential smoothing model (ETS) [120, 126] which was not yet implemented in `sktime` at the time. In addition, we re-implement and evaluate the improved model by Legaki & Koutsouri, the best performing statistical model in the M4 competition. We call the model "Theta-bc" since it involves a Box-Cox transformation [37]. We give an overview of all models we re-implement in Table 9.3.

9.6.2 Results

We compare our results for each model against the published results in terms of average performance and computational run time. An overview of our results is presented in Table 9.4, which shows the percentage differences between reproduced and published sMAPE values for the data sets grouped by sampling frequency. Corresponding results for MASE and OWA are shown in the appendix in Tables C.1 and C.2.

We test whether the found differences are statistically significant using a paired t-test. Detailed results of the significance tests for sMAPE and MASE values are shown in the appendix in Table C.3 and C.4, respectively. Summary results are shown in Table 9.5.

Our main findings are as follows:

- For all naive forecasters, we can reproduce the published results perfectly, barring negligible differences due to numerical approximations. This validates our experimental workflow for training, prediction and model evaluation.
- For the statistical models, we find small but often statistically significant

Table 9.3: Overview of reproduced M4 models

Name	Category	Description
Naive	naive	Always predicting the last value of the training set.
sNaive	naive	Always predicting the last value of the training set of the same season.
Naive2	naive	Like Naive, but with seasonal adjustment by applying classical multiplicative decomposition [57] and an auto-correlation test at the 90% significance level to decide whether or not to apply seasonal adjustment [81].
SES	statistical	Simple exponential smoothing and extrapolating [113, 114, 263], with no trend, and seasonal adjustment as in Naive2.
Holt	statistical	Like SES, but with a linear trend.
Damped	statistical	Like Holt, but with a damped trend [92].
Theta	statistical	As applied to the M3 Competition [179] using two Theta lines, $\theta_1 = 0$ and $\theta_2 = 2$, with the first one being extrapolated using linear regression and the second one using SES. The forecasts are then combined using equal weights [9]. This is equivalent to special case of simple exponential smoothing with drift [123]. Seasonal adjustments are considered as in Naive2.
Theta-bc	statistical	Like Theta, but with Box-Cox adjustment [37], where lambda is estimated via maximum likelihood estimation and constrained to the $(0, 1)$ interval. Submitted to the M4 competition by Legaki & Koutsouri (submission #260).
Com	statistical	Simple arithmetic mean of SES, Holt and Damped.
ARIMA	statistical	An automatic selection of possible seasonal ARIMA models is performed and the best one is chosen using appropriate selection criteria [39, 124].
MLP	ML	A multi-layer perceptron of a very basic architecture and parameterization via a recursive reduction strategy with window length set to 3 (see Section 3.4). Linear detrending and seasonal adjustments is applied as in Naive2 to facilitate extrapolation.
RNN	ML	A recurrent network of a very basic architecture and parameterization via a recursive reduction strategy with window length set to 3 (see Section 3.4). Linear detrending and seasonal adjustments is applied as in Naive2 to facilitate extrapolation.

Notes: The forecasters are described in detail in the original M4 competition [182]. Following the M4 competition, models are grouped into three categories: "naive", "statistical" and "ML". We follow the categorization of the M4 competition for consistency, but more informative categorizations have been proposed by Januschowski et al. [135].

differences. The largest difference we find for sMAPE is 4% for the Holt model on the yearly data set. There appears to be no clear trend in the differences. In some cases, published results are better, in others ours. A

Table 9.4: sMAPE percentage difference between reproduced and published results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Naïve	0.000	0.000	0.000	0.000	0.000	0.000
Naïve2	0.000	0.000	0.000	0.000	0.000	0.000
sNaïve	0.000	0.000	0.000	0.000	0.000	0.000
SES	−0.004	0.069	0.016	−0.005	0.011	0.000
Holt	4.063	−1.528	3.916	−3.365	0.286	−4.347
Damped	1.586	−1.024	0.010	−0.694	1.036	−0.783
Com	1.659	−0.615	1.123	−1.418	0.498	−1.538
ARIMA	1.617	4.572	2.418	0.851	−2.142	−2.017
Theta	−1.514	0.046	0.078	0.174	0.057	0.008
Theta-bc	−0.948	−0.096	−0.092	−0.043	0.050	3.378
MLP	−11.936	−24.109	−27.567	−52.596	−61.727	−4.558
RNN	−22.728	−29.329	−30.815	−25.979	−33.001	−9.332

Notes: Rows show forecasters described in table 9.3. Columns show M4 data sets grouped by sampling frequency. Values show the percentage difference between reproduced and published mean sMAPE values relative to the published values. Negative values indicate that reproduced results are lower (better) than published ones.

possible explanation of the differences is the randomness involved in the optimization routines used during fitting. However, since we do not apply the same model repeatedly on the same series, we cannot reliably quantify that source of variation. Of course, differences may also be due to implementation differences or bugs.[5]

- For the ML models, including MLP and RNN, we find large and statistically significant differences. Differences are entirely negative, ranging from −11% for MLP on the yearly data set to −61% on the daily data set, indicating that these models now perform better than in the original competition. A possible explanation is that the algorithm implementations in `scikit-learn` [205] and `TensorFlow` [1] have been improved since the M4 competition, which may also explain the improved run times shown in Table 9.5.

[5]For example, `statsmodels`' exponential smoothing model seemed to return wrong forecasts in a few cases, as reported at `https://github.com/statsmodels/statsmodels/issues/5877`. We also discovered that the M4 competition used inconsistent seasonality tests for Python and R which we take into account when reproducing the results, as report at `https://github.com/Mcompetitions/M4-methods/issues/25`.

Table 9.5: Summary of reproduced M4 results

	Mean rank (sMAPE)			**Replicated metrics**			**Run time (min)**		
	Replicated	Original	Change	sMAPE	MASE	OWA	Replicated	Original	Factor
Theta-bc	5.454	5.242	-0.212	11.952	1.583	0.876	8.100	25.00	0.3
Theta	5.649	5.437	-0.212	12.264	1.669	0.900	6.268	12.70	0.5
Com	5.726	5.454	-0.271	12.668	1.687	0.914	69.473	33.20	2.1
Damped	5.925	5.635	-0.291	12.692	1.718	0.920	53.448	15.30	3.5
ARIMA	5.748	5.473	-0.275	12.992	1.673	0.920	14992.879	3030.90	4.9
SES	6.788	6.700	-0.088	13.090	1.885	0.970	5.902	8.10	0.7
Holt	6.001	5.780	-0.222	14.160	1.830	0.997	11.720	13.30	0.9
Naïve2	7.029	6.736	-0.292	13.564	1.912	1.000	3.664	2.90	1.3
RNN	6.784	8.314	1.529	15.122	1.902	1.067	38941.684	64857.10	0.6
Naïve	7.337	7.050	-0.287	14.208	2.044	1.072	1.035	0.20	5.2
sNaïve	8.046	7.729	-0.316	14.657	2.057	1.105	1.028	0.30	3.4
MLP	7.513	8.450	0.937	16.480	2.079	1.156	157.884	1484.37	0.1

Notes: Rows show forecasters described in Table 9.3. Reproduced run times are scaled to the number of CPUs used in the original M4 competition.

In addition, we compare the computational run times between `sktime` and the model implementations used in the original M4 competition, which mostly relied on the `forecast` library in R. Since we use different machines to reproduce the results (see Section 9.5 for more details), run times are not directly comparable. To make them more comparable, we scale our obtained run times to the number of CPUs used in the M4 competition. The scaled values are shown in Table 9.5. Most notably, ARIMA takes approximately 5 times longer than in the M4 competition. This is likely because R's `forecast` library supports the conditional sum of square approximation technique for model estimation [36, p. 209ff.], which is considerably faster, especially for long series, but not yet supported by `pmdarima` and `statsmodels`, the libraries we rely on internally. On the other hand, MLP and RNN, based on `scikit-learn` [205] and `TensorFlow` [1], are now substantially faster than in the M4 competition. The remaining run times are more or less on par. SES, Holt and Theta are slightly faster in `sktime`, the naive forecasters and Damped are slightly slower.

9.7 Extending the M4 competition

Having reproduced key results from the M4 competition, we want to extend it for two reasons:

- First, we want to further investigate the potential of ML models for forecasting. In contrast to most of the ML models in the original M4 competition, we focus on models that require only a single series for training and hence cannot make use of consistent patterns across multiple series.

- Second, we want to illustrate the effectiveness and applicability of `sktime` for solving forecasting tasks. We hope to show that, thanks to its modular API, `sktime` allows to easily specify, tune and evaluate models, including common ML techniques like pipelining, reduction and boosting.

9.7.1 Research questions

In particular, our investigation is guided by three research questions:

Question 1: Can simple reduction approaches based on cross-sectional regression algorithms outperform the statistical models included in the M4 competition?

Question 2: Can one-step residual boosting with simple cross-sectional regression algorithms enhance the predictive performance of Theta-bc (see Table 9.3), the best performing statistical model in the M4 competition?

Question 3: Does tuning of the window length hyper-parameter in reduction approaches based on cross-sectional regression algorithms improve performance?

9.7.2 Models

To investigate these questions, we evaluate five previously unstudied ML models in the M4 competition. These models are described in detail in Table 9.6. All our models rely on linear detrending of the data, as the sliding window transformation of the reduction approach makes it difficult for ML models to pick up trends. We evaluate each of the models with four simple regression algorithms: Linear regression (LR), K-nearest-neighbors (KNN), random forest (RF), and gradient boosted trees (XGB). For the Theta-based models #4 and #5 in Table 9.6, we exclude LR, as Theta already includes some linear extrapolation. This leads to a total of 18 new models that we add to the original M4 competition. For more details on the regressors and their hyper-parameter settings, see Table 9.7.

9.7.3 Results

Detailed results for all models are reported in the appendix in Tables C.5, C.6, C.7, and C.8. Below we discuss our three research questions in turn.

Table 9.6: Overview of ML models used to extend the M4 competition

#	Name	Category	Description
1	{regressor}	ML	Regression via reduction, using the standard recursive strategy for generating predictions (see Section 3.4). No seasonal adjustment, but linear detrending and standardization (removing the mean and scaling to unit variance) is applied. The window length is set to $max(m, 3)$, where m is the seasonal periodicity of the data.
2	{regressor}-s	ML	Like #1, but with seasonal adjustment as in Naive2.
3	{regressor}-t-s	ML	Like #2, but with tuning of the window length. We use a simple temporal cross-validation scheme, in which we make a single split of the training series, using the first window for training and the second window for validation. The validation window has the same length as the forecasting horizon (i.e. the test series). We search over the following window length values: 3, 4, 6, 8, 10, 12, 15, 18, 21, 24.
4	{regressor}-Theta-bc	Hybrid	One-step residual boosting of Theta-bc (see Section 8.5.3). Standardization is applied as in #1 to the Theta-bc residuals. Theta-bc is defined in Table 9.3. Window length is set as in #1.
5	{regressor}-Theta-bc-t	Hybrid	Like #4, but with tuning of the window length as in #3.

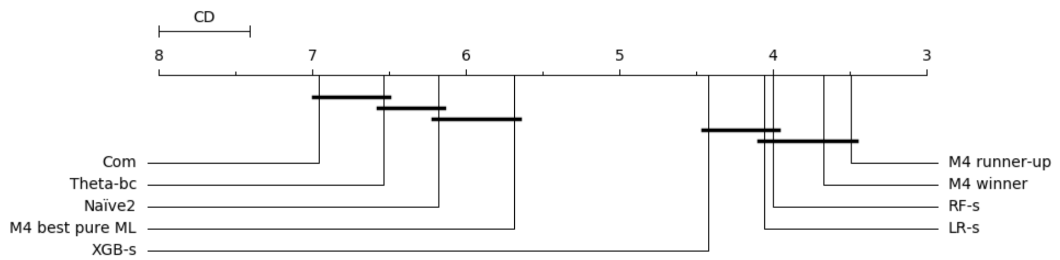
Notes: {regressor} is a placeholder for the cross-sectional regression algorithms, including Linear regression (LR), K-nearest-neighbors (KNN), random forest (RF), and gradient boosted trees (XGB), described in Table 9.7.

Table 9.7: Cross-sectional regression algorithms used to extend the M4 competition

Name	Description	Hyper-parameters
LR	Linear regression	fit_intercept=True
KNN	K-nearest neighbors	n_neighbors=1
RF	Random forest	n_estimators=500
XGB	Gradient boosted trees	n_estimators=500

Notes: For all algorithms except XGB, we use `scikit-learn` [205]. For XGB, we use `xgboost` [53]. For all other hyper-parameters, we use the libraries' default values.

Figure 9.1: Critical difference diagram based on sMAPE and the hourly data set for selected models



Notes: The diagram is based on sMAPE performance and the hourly data set. On the horizontal line, the diagram shows mean ranks for each forecaster. Forecaster grouped by a bar are not statistically significant based on pairwise post-hoc Nemenyi tests at the 5% level. Corresponding Wilcoxon-Holm test results are shown in the appendix in Table C.9.

Question 1: Can simple reduction approaches based on cross-sectional regression algorithms outperform the statistical models included in the M4 competition?

We present selected OWA results in Table 9.8. We also report some original M4 models as a reference for comparison, particularly the M4 winner [234, 235], the runner-up [192], the best pure ML model submitted by Trotta which we call "M4 best ML" (a convolutional neural network adapted to time series), and Theta-bc, the best performing statistical model submitted by Legaki & Koutsouri.

In line with previous results [180, 182], we find that over the whole of the M4 data set, the new ML models cannot on average achieve equal or better performance compared to the statistical models. However, on the hourly data set, ML models can perform better than statistical ones. Our best model, RF-s with an OWA of 0.493 comes even close to the cross-learning models of the M4 winner (0.44) and runner-up (0.484). XGB-s (0.496) and LR-s (0.501) achieve slightly worse, but still competitive performances.

To check if found performance differences are statistically significant, we use significance tests. We show the corresponding critical difference diagram in Figure 9.1. Results from Wilcoxon-Holm tests are shown in the appendix in Table C.9. For the hourly data set, the observed differences between RF-s and the statistical models are statistically significant. The differences between RF-s and the M4 winner and the runner-up are not statistically significant.

To answer question 1: No, simple reduction approaches based on cross-sectional regression algorithms cannot achieve equal or better performance compared to statistical methods on the whole of the M4 data set. But they can achieve equal or better performance on the hourly data set. Indeed, here they even perform on par with the the custom-built best performing M4 models, with no statistically significant difference between them.

It is worth emphasizing that compared to the best performing M4 models, our models are considerably simpler. They only need a single series for training. They only rely on simple cross-sectional regression algorithms. And with `sktime`, they only use out-of-the-box functionality without requiring any custom-built components.

Table 9.8: Performance of ML models used extend M4 competition (OWA)

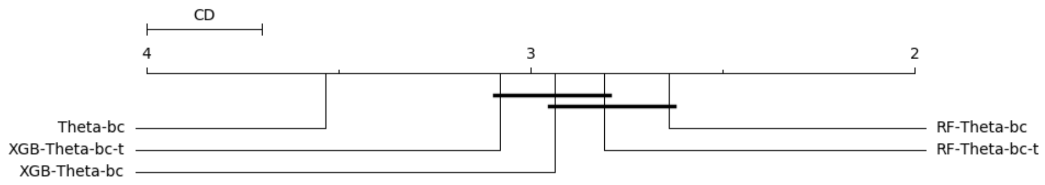
	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly	Total
M4 winner	0.778	**0.847**	**0.836**	0.851	1.046	**0.44**	**0.833**
M4 runner-up	0.799	**0.847**	0.858	**0.796**	1.019	0.484	0.847
Theta-bc	**0.776**	0.893	0.904	0.964	0.996	1.009	0.876
Com	0.886	0.885	0.93	0.911	**0.982**	1.506	0.914
M4 best ML	0.859	0.939	0.941	0.996	1.071	0.634	0.926
RF-s	0.967	1.014	0.994	1.015	1.078	0.493	0.994
Naïve2	1.0	1.0	1.0	1.0	1.0	1.0	1.0
XGB-s	1.022	1.091	1.118	1.113	1.149	0.496	1.088
LR-s	-	1.037	2.16	0.964	1.07	0.501	-

Notes: Rows show forecasters described in Table 9.6. Columns show M4 data sets grouped by sampling frequency. The entry "M4 best ML" refers to the M4 submission by Trotta which was the best pure ML model. We exclude results for LR model when generated forecasts were unstable/exploding due the little available data and linear extrapolation.

Question 2: Can one-step residual boosting with simple cross-sectional regression algorithms enhance the predictive performance of Theta-bc, the best statistical model in the M4 competition?

To investigate the second question, we compare results for Theta-bc with their one-step residual boosted variants based on the RF and XGB regression algorithms. We also include the versions of the boosted models with tuning for the window length hyper-parameter. Results are shown in Table 9.9.

Figure 9.2: Critical difference diagram based on sMAPE and the hourly data set for Theta models



Notes: The diagram is based on sMAPE performance and the hourly data set. On the horizontal line, the diagram shows mean ranks for each forecaster. Forecaster grouped by a bar are not statistically significant based on pairwise post-hoc Nemenyi tests at the 5% level. Corresponding Wilcoxon-Holm test results are shown in the appendix in Table C.10.

Our main findings are as follows: Over the whole of the M4 data set, one-step boosting does not improve the performance of Theta-bc. Boosting even leads to a performance loss on the yearly, monthly and quarterly data sets. On the other hand, it leads to slight performance gains on the weekly, daily and hourly data set. In particular, on the daily data set, boosting with RF improves the OWA of Theta-bc from 0.996 to 0.988, and on the hourly data set from 1.009 to 0.985. For the weekly data set, tuning is needed to improve accuracy beyond that of Theta-bc.

Again, we test whether found performance differences on the hourly data set are significant. As shown in Figure 9.2, we find that the boosted variants perform significantly better than Theta-bc without boosting on the hourly data set, with no significant difference between the different regression algorithms.

Table 9.9: Performance of one-step residual boosted Theta-bc models (OWA)

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly	Total
Theta-bc	**0.776**	**0.893**	**0.904**	0.964	0.996	1.009	**0.876**
RF-Theta-bc-t	0.854	0.938	0.933	**0.959**	0.999	0.987	0.919
RF-Theta-bc	0.864	0.957	0.932	1.052	**0.988**	**0.985**	0.925
XGB-Theta-bc	0.898	1.017	0.971	1.153	1.023	0.993	0.968
XGB-Theta-bc-t	0.906	1.009	0.976	1.006	1.04	0.998	0.971

Notes: Rows show forecasters described in Table 9.6. Columns show M4 data sets grouped by sampling frequency.

Question 3: Does tuning of the window length hyper-parameter in reduction approaches based on cross-sectional regression algorithms improve performance?

To investigate the last question, we compare the performance of each regressor, once with a tuned value for window length and once with the default value (see Table 9.6). Results are shown in Table 9.10. Our main findings are as follows:

Over the whole of the M4 data set, tuning of the window length does not improve performance. Exceptions are the weekly data set and the KNN regressor. On the weekly data set, all tried out regressors benefit from tuning, with the biggest OWA improvement being 0.082 for XGB. KNN additionally benefits from tuning on the yearly and hourly data set.

Note that other temporal cross-validation schemes than the ones tried out here are possible and may prove to be more beneficial to overall performance (e.g. a sliding window validation). In addition, one could try to optimize other hyper-parameters (e.g. the strategy to generate forecasts as discussed in 3.4, or the hyper-parameters of regressor using a nested cross-sectional cross-validation scheme). However, note that tuning comes at the cost of a considerable increase in run time, as reported in the appendix in Table C.8.

Table 9.10: Performance of tuned ML models (OWA)

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly	Total
RF-s	0.967	1.014	0.994	1.015	1.078	0.493	0.994
RF-t-s	1.005	1.070	1.057	0.967	1.087	0.683	1.047
XGB-s	1.022	1.091	1.118	1.113	1.149	0.496	1.088
XGB-t-s	1.038	1.144	1.170	1.031	1.179	0.746	1.131
KNN-s	1.086	1.171	1.257	1.218	1.338	0.544	1.197
KNN-t-s	1.062	1.185	1.276	1.147	1.331	0.751	1.205
LR-s	-	1.037	2.160	0.964	1.070	0.501	-
LR-t-s	-	-	1.876	0.889	1.089	0.670	-

Notes: Rows show forecasters described in Table 9.6. Columns show M4 data sets grouped by sampling frequency. We exclude LR model results when generated forecasts were instable/exploding, likely due the little available data and linear extrapolation.

9.8 Concluding remarks

We used `sktime` to reproduce and extend the M4 forecasting competition. Reproducing the M4 competition allowed us to validate published results, test our framework against reference implementations, and illustrate its effectiveness and applicability. We found no or only small differences for the naive and statistical forecasting algorithms, and large improvements for the ML algorithms.

Extending the M4 competition allowed us to further investigate the potential of simple out-of-the-box ML approaches for forecasting. In particular, we found that simple reduction-based ML models using cross-sectional regression algorithms can achieve competitive performance on the hourly data set, outperforming the statistical algorithms and achieving comparable performances to the best M4 models.

It also appears that the characteristics of the series may be a critical factor determining the performance of ML models. This suggests that certain forecasting problems warrant a more systematic exploration – something which we hope `sktime` can facilitate in future research.

Chapter 10

Conclusion and Directions for Future Work

This thesis had four main goals. The overarching goal was to develop a unified software framework for ML with time series and implement it in an open-source Python library called `sktime`. A second goal was to create a formal taxonomy of time series data and learning tasks. A third goal was to derive ML-specific software design principles. While the taxonomy of tasks and software design principles were research contributions in their own right, they were necessary to achieve the first goal. The fourth and final goal was to use `sktime` to reproduce and extend the M4 competition, one of the major comparative benchmarking studies for forecasting.

Our approach to developing the unified framework for ML with time series largely fell under domain-driven design. The initial step in domain-driven design is to develop a conceptual model of the domain of interest. The software is then implemented to closely match the conceptual model. The focus of part I of this thesis was to develop a conceptual model for ML with time series.

We started in Chapter 2 by introducing ML with time series and defining the concept of a learning task. A learning task was defined as a combination of a (i) description of the available data, both in terms of its relational and statistical structure, (ii) a learning process specification detailing at what point what input data is given to the learning algorithm and what output it returns, and (iii) a definition of success for evaluating algorithm performance. From this, we developed in Chapter 3 a taxonomy of key time series data forms (univariate, multivariate and panel data) and key time series learning tasks, including time series classification, regression and forecasting. We focused on (deterministic) point-predictive tasks, leaving other tasks for future work. While the taxonomy highlighted the differences between learning tasks, we also discussed reduction as a key concept to understand their relation. We gave an overview of important reduction relations in the time series setting, described common examples and discussed general properties of reduction relations (modular, composable, adaptive and tunable).

With the taxonomy for time series tasks at hand, we needed a better understanding of the domain of ML more generally, and a set of software design principles that could guide the development of the proposed unified framework. This was the focus of part II. We started in Chapter 4 by reviewing key ideas from classical software design and object-oriented programming as the predominant programming paradigm for ML software. As is standard in software design, we then began in Chapter 5 to identify those workflow elements that are more likely to change than others. These elements would serve as sensible points of abstraction. Examples of such change-prone elements include learning tasks, as described above, and learning algorithms, among others. In order to further develop our conceptual model, we then formalized, abstracted and related these elements based on their common attributes and interface points. At the core of our conceptual model, we proposed a new type system, called "scientific types". We defined a scientific type as a structured data type combined with mathematical or statistical properties that all elements of that type must satisfy. Scientific types extend classical type system as found in programming languages and mathematical formalism to data science. Like classical types, they define a value space into which all objects of a given type must fall (e.g. integers) and a set of well-defined operations for objects in that space (e.g. addition). In addition, they ascribe a set of statistical properties to all objects of a given type. These properties reflect the particular kind of learning task in relation to which these objects have been defined. For example, we can define a scientific type for forecasting algorithms in correspondence to the forecasting learning task, or a cross-sectional classification algorithm corresponding to the cross-sectional classification task. As such, scientific types categorize objects according to their data scientific purpose. We illustrated that we cannot just express simple algorithms like forecasters or cross-sectional supervised learners with scientific types, but also higher-order composition algorithms such as pipelining, ensembling, tuning or reduction, among others.

Based on our conceptual model, and scientific types in particular, we were able to formulate general design principles for ML toolbox design. We derived in Chapter 6 a set of general guiding principles for ML toolbox design. We also proposed design patterns to describe solutions to recurring problems in ML toolbox design, extending the classical design patterns put forward by Gamma et al. [90]. These principles and patterns, together with scientific types, ensure that the conceptual domain model is tightly linked to the underlying mathematical and statistical formalism, but also easily implementable in software. Using `scikit-learn` [205] as an example, we illustrated that these principles and patterns can explain key aspects of existing toolboxes.

With these principles in mind, we were ready to address the goal of developing a unified framework for ML with time series. We described in Chapter 8 the design of our framework and its implementation in an open-source library in Python called `sktime`. The design was derived from combining our previous insights from the taxonomy of learning tasks, scientific types as well as our design principles and patterns. Specifically, we first identified relevant objects in the domain of time series, then abstracted them in the form of scientific types and finally implemented them as part of a consistent, modular and unified API for ML with time series.

In the last part of this thesis, we used `sktime` to reproduce and extend the M4 competition. Reproducing the M4 competition allowed us to validate the published results, test our framework against reference implementations, and illustrate its applicability. Extending the M4 competition allowed us to evaluate previously unstudied models and investigate new composite models that are easily implementable in `sktime`. We began in Chapter 9 by re-implementing key algorithms from the competition independently of the published code. We found that our reproduced results were at least as good as the published ones, with minor variations and some larger improvements for ML models. We then set out to investigate whether simple, out-of-the-box reduction-based ML models provided by `sktime` can achieve competitive performance compared to the custom-built, cross-learning ML models that performed best in the M4 competition. We found that on a subset of the series included in the M4 data set, our new models can achieve state-of-the-art performance, with no statistically significant difference to the M4 winner models.

Before we discuss limitations and directions for future research, we summarize the contributions of this thesis below.

10.1 Summary of contributions

In summary, the contributions of this thesis are as follows:

1. *Formalization of time series learning problems* as learning tasks and development of a formal *taxonomy for time series learning tasks*, with a focus on common (deterministic) point-predictive tasks such as time series classification, regression and forecasting, as well as the reduction relations between them.

2. *Development, formalization and motivation of a set of general, reusable design principles for ML toolboxes* based on the idea of a *scientific type system* that links the software implementation with the underlying mathematical and statistical concepts.

3. *Review of existing time series analysis software* and a discussion of the limitations of current toolbox capabilities, with a focus on the open-source

ecosystem in Python.

4. *Design and implementation of the first unified framework for ML with time series*, with the aim to provide a principled and modular object-oriented application programming interface (API) in Python for specifying, training and validating time series algorithms for medium-sized data. The unified framework has been implemented in an open-source project called `sktime`.[1]

5. *Reproduction and extension of the M4 forecasting competition*, one of the major comparative benchmarking studies for predictive performance of forecasting algorithms, with a focus on using `sktime` to evaluate and compare simple reduction-based ML algorithms. To our knowledge, this is the first complete reproduction of the M4 competition within a single framework independent of the published code.

10.2 Limitations and future work

In this section, we discuss limitations of this thesis and directions of future research.

10.2.1 Selection of learning tasks

Time series data give rise to a wide variety of learning tasks. This thesis focused on three key tasks – time series classification, regression and forecasting. All of these tasks were phrased as (deterministic) point-predictive tasks. In future work, we hope to extend our taxonomy beyond these tasks. Table 3.1 gave an informal overview of common time series tasks. We consider the following tasks as key directions for future research, both in terms of formalization and software development within `sktime`:

- *Additional predictive tasks.* Common predictive tasks not included in this thesis are time series annotation tasks such as change point detection [5],

[1] `https://github.com/alan-turing-institute/sktime`

anomaly detection [52] and outlier detection [110] as well as multivariate forecasting [175], panel forecasting [16, 267] and supervised forecasting [98].

- *Non-predictive tasks.* One of the most popular non-predictive tasks is time series clustering [2]. Other non-predictive tasks include time series segmentation [143] and motif discovery [194].

- *Probabilistic tasks.* Probabilistic tasks include probabilistic variants of the above tasks, including probabilistic time series classification, regression and forecasting, as well as inherently probabilistic tasks such as survival analysis (single event risk prediction under censoring) [60, 139, 146] and point process modeling (multiple event risk prediction) [64].

- *Online learning variants of tasks.* Most of the above tasks are also possible in an online learning setting in which data becomes available incrementally as time moves on (e.g. early time series classification [268]). In this setting, it is often preferable to update a fitted model when new data arrives, rather than to completely refit it. Note that, depending on the data generative setting, new data may consist of new instances, new time points or both.

The conceptual model for ML theory and the design principles presented in this thesis are generalizable to other tasks. In this thesis, we focused on key conceptual objects (e.g. tasks and algorithms). To support other tasks, the conceptual analysis may have to be extended to other common objects, for example loss functions and probability distributions. Similarly, some these tasks are straightforward extensions to existing APIs (e.g. time series clustering), whereas others will require the development of new API designs. In general, inclusion of these tasks will require formalizing them, developing an API design, in line with the general design principles and patterns derived in this thesis, and implementing core algorithms and related functionality.

10.2.2 Simplifying assumptions

Much of this thesis relied on common simplifying assumptions, both in the formalization of learning tasks and the implementation of `sktime`. These include the assumption of time-homogeneity, as discussed in Chapter 3. In particular, we assumed that time series data are equally spaced and that they share a common time index across variables and instances. In the taxonomy of tasks, these assumptions eased notation and allowed us to concentrate on the most important aspects. In `sktime`, these assumptions allowed us to rely on well-established data containers (e.g. `NumPy` and `pandas`).

However, these assumptions are often violated in real-world applications. Time series data tends to exhibit considerable complexity. For example, different variables are often recorded at different frequencies and different instances are often observed over different time periods. In future work, we hope to extend our analysis and the implemented functionality to handle such data formats. One of the main challenges is the development of a data container to deal with "ragged" temporal and hierarchical data. `awkward-array` [211] presents a promising solution in this context and we wish to further explore its compatibility with `sktime`.

10.2.3 Toolbox design: Towards a more declarative language for ML

The research presented in this thesis marks a first attempt at consolidating the science of toolbox design. We hope that our research will inspire more research on the underlying design principles of ML software.

In particular, the conceptual analysis presented in this thesis, and scientific types in particular, mark a first step towards a more declarative design for ML toolboxes. The current API design of `sktime`, much like that of other ML toolboxes, falls under the imperative programming paradigm. It uses syntax statements such as `fit` or `predict` that express commands for the computer to perform. It focuses on how the program operates, in contrast to a more declarative language, which focuses on what the program should accomplish. A more declarative syntax would use a precise description of the task to be solved and properties of the preferred solution, without requiring a specification how the program should achieve the solution. To achieve this level of abstraction, it is necessary to clearly capture and delineate task descriptions in the program and to categorize functionality according to their ability to address these tasks.

In future work, we want to develop a layered API by adding a more declarative layer for task specification in `sktime`. Currently, `sktime` creates the domain layer tightly linked to our conceptual model of time series analysis. The task specification layer would add a new user interaction layer on top that makes it easier for users to run the most common workflows from specifying a task, to fitting a model and generating predictions, without having to specify details of the learning process itself.

Author's Contribution

Markus Löning (ML) is solely responsible for content and exposition of this thesis.

`sktime` is an community-driven open-source project with contributions from other community members. ML has been `sktime`'s lead developers since its beginning, contributing to almost all parts of it, most importantly, the design and implementation of the core interfaces for classification, forecasting, transformation, composition and model evaluation, as well as the implementation of specific algorithms.

An outline of `sktime`, its design and the taxonomy of learning tasks first appeared in Löning et al. [169]. Chapter 3 and 8 are partly based on this paper. The paper was written by ML, with contributions from Franz Király (FK) and small contributions from the other authors. FK initially conceived the research project. ML then made key contributions in this thesis to the conceptualization, design and implementation.

An overview of `sktime`'s forecasting interface first appeared in Löning and Király [167]. Chapter 8 is partly based on this paper. The paper was written by ML, with contributions from FK. In this thesis, ML refines some points, adds detail and aligns it with the overall structure and terminology of the thesis.

The discussion on design principles for machine learning toolboxes first appeared in Király et al. [145]. Chapters 5 and 6 are partly based on this paper. The paper was jointly written by ML and FK, with small contributions from the other authors. The research project was initiated by FK. Anthony Blaom first introduced the idea and the term "scitype" in the context of data containers and the `MLJ` software package. FK conceived of scitypes of objects beyond data containers. ML made key contributions by further formalizing the idea, applying it to the time series setting and implementing it in `sktime`. In this thesis, ML further refines some points, adds detail and aligns it with its overall structure and terminology.

The reproduction and extension of the M4 competition first appeared in Löning and Király [167]. Chapter 9 is largely based on this paper. The benchmarking experiments and description of the results was done by ML, with small contributions from FK. All code was written by ML. In this thesis, ML refines some points and links them to the rest of the thesis.

Appendix A

Object-Oriented Programming

In this chapter, we will review key concepts from object-oriented programming, the predominant paradigm for ML toolboxes. We will refer to these concepts frequently throughout the next chapters, including in the conceptual analysis in Chapter 5, the discussion of our design principles in Chapter 6 and in the design of `sktime` in Chapter 8.

In our review below, we largely rely on the exposition in Gamma et al. [90], but prefer our own terminology for some of the concepts, in line with common usage in data science software in Python.

Object. Object-oriented programs are made up of objects. An “object”, sometimes also called a “structured object”, packages data and methods that operate on that data.

Attribute. The data is typically represented as “attributes” that are accessible by the object’s methods. Attributes are sometimes also called “properties”, “variables” or “fields”.

Method. A “method”, sometimes also called “operation”, is essentially a function associated with an object that may operate on the object’s data.

Function. A “function”, sometimes also called “routine”, “subroutine” or “procedure”, is a sequence of instructions that perform a specific task packaged as a unit.

The function can then be used in programs wherever that particular task should be performed. Functions are usually “side effect” free, meaning that they do not affect the state of other objects. Methods, on the other hand, might not be side effect free, as they modify the state of objects.

Call. An object performs a certain method when it is “called” from a “client”. This is sometimes expressed as “when it receives a request or message from a client”. When a call is sent to an object, the particular method that is performed depends on both the call and the receiving object. Different objects that support identical calls may have different implementations of the method that fulfills the call.

Client. A “client” may be a user or another object, which in responding to a call, calls a method from another object.

Encapsulation. Calls are the only way to get an object to execute a method. Methods are the only way to change an objects internal data. Because of these restrictions, the object’s internal state is said to be “encapsulated”: it cannot be accessed directly, and its representation is hidden form the outside of the object. This means that changes to the internal representation can affect only a small, easily identifiable region of the program. This constraint greatly improves the modularity and maintainability of larger libraries. Note that different programming languages have different ways to enable encapsulation, some more, other less strict.[1]

Signature. Every method declared by an object specifies the method’s name, the input arguments it takes and the method’s output, jointly known as the method’s “signature”.

Interface. The set of all signatures defined by an object is called the object’s “interface”. An object’s interface characterizes the complete set of methods that can be called on the object. Note that the object’s internal representation does

[1]For example, some language allow fields and methods to be marked “public” or “private”, where private methods and fields restrict their call sites to other methods of the same object. Other languages, such as Python, do not have mechanisms for enforcing encapsulation, but rely on conventions among developers to ensure encapsulation.

not appear in its interface since it does not affect the methods we can perform with the object. Objects are known through their interfaces: there is no way to know anything about an object or to ask it to do anything without going through its interface. An object's interface says nothing about its implementation, that is, different objects are free to implement calls differently. That means that two objects having completely different implementations can have identical interfaces. Two objects responding to the same set of methods, i.e. with the same interface, may use entirely different implementation of the methods, as long as each carries with it an implementation of methods that works with its particular internal representation. Focusing on the object interface rather than its implementation or internal structure has two advantages: First, clients – be it users or other objects – can remain unaware of the specific objects they use as long as the objects adhere to the interface that clients expect. Second, clients can remain unaware of the classes that implement these objects, clients only need to know about the interface. Consequently, we can write programs that expect an object with a particular interface knowing that any object that has the expected interface will accept the call.

Application programming interface. The set of all objects, their interfaces and any stand-alone functions not associated with any object that can be called by the user is called the "application programming interface" or "API" for short.

Polymorphism. Encapsulation allows us to substitute objects that have identical interfaces for each other at run time. This substitutability is known as "polymorphism". Polymorphism lets a client object make few assumptions about other objects except that they support a particular interface. Polymorphism simplifies the definition of client objects, decouples objects from each other, and lets them vary their relationship to each other at run time.

Types. Object with the same interface are considered to have the same "type". An object may also have many types. Parts of an object's interface can be characterized by one type, and other parts can be characterized by other types.

Two objects of the same type need only share parts of their interfaces. Interfaces can contain other interfaces as subsets, we say that a type is a "subtype" of another if its interface contains the interface of its subtype. We often speak of a subtype inheriting the interface of its "supertype". We will discuss types in more detail in Chapter 5.

Class. Let us consider in more detail how objects are implemented. An object's implementation is defined by its "class". The class specifies the object's internal data and defines the methods the object can perform. Objects are created by instantiating a class. The object is said to be an "instance" of the class. So a class can be understood as a template from which we can create an object. For example, we can have multiple instantiations (objects) of the same template (class). It is important to differentiate between a class and its type: An object's class defines how the object is implemented. The class defines the object's internal state and the implementation of its methods. By contrast, an object's type only refers to its interface, that is, the set of calls which it can perform. An object can have many types and objects of different classes can have the same type. Of course, there is a close relationship between an object's class and type: because a class defines the methods an object can perform, it also defines the object's type. When we say that an object is an instance of a class, we imply that the object supports the interface defined by the class.

Inheritance. New classes can be defined in terms of existing classes. When a new class "inherits" from an old class, the new class includes the definitions of all the data and methods that the old class defines. We say the "subclass" "inherits" or "extends" the "parent" class. A subclass may "override" an method defined by its parent class. Overriding gives a subclass a chance to handle calls instead of relying on the behavior of the parent class. Inheritance has two advantages: First, inheritance is a mechanism for extending functionality by re-using functionality in parent classes. It makes it easy to define new classes simply by extending other classes. Subclasses can add or change behavior of parent classes. It is a mechanism for code and representation sharing. Second, inheritance lets you define families of objects with identical interfaces by inheriting from a common parent class. When inheritance is used carefully, all classes derived from the parent class will share its interface. This implies that a subclass merely adds or overrides methods but does not hide methods of the parent class. All subclasses can then respond to the call in the interface defined by the parent class, making them all subtypes of the parent class. Some programming languages also support "multiple inheritance", meaning that a subclass can have more than one parent class.

Abstract class. An abstract class is one whose main purpose is to define a common interface for its subclasses. An abstract class will defer some or all of its implementation to methods defined in subclasses, hence an abstract class cannot be instantiated. The methods that an abstract class declares but does not implement are called "abstract methods".

Concrete class. A class that is not abstract is called a "concrete class".

Mixin class. A mixin class is one that is intended to provide an optional interface or functionality to other classes It is similar to an abstract class in that it is not intended to be instantiated. Mixin classes require multiple inheritance.

Composition. Composition is an alternative to class inheritance. Here new functionality is acquired by assembling or "composing" objects to get more complex functionality. Object composition requires that the objects being composed have well-defined interfaces. Whereas class inheritance is statically defined at compile time, object composition is dynamically defined at run time through objects obtaining references to other objects.

Delegation. In composition, there are more than one object involved in handling a call. A receiving object may "delegate" a call to another object in the composition. This is analogous to subclasses deferring calls to parent classes in class inheritance.

Appendix B

Classical Software Design

Having described the developer's problem and key quality criteria of ML toolbox design in Chapter 4, we here give an overview of classical software design.

Various software design approach have been proposed.[1] Our approach to addressing the developer's problem largely falls under "domain-driven design" as put forward by Evans [78], but also draws on ideas by "pattern-oriented design" [48, 82, 90] and other approaches such as "responsibility-driven design" [264–266], "design by contract" [188] as well as best practices as discussed by Larman [152]. What all of these approaches have in common is the idea of taking those elements that are likely to vary jn typical practitioners' workflows as sensible points for abstraction.

B.1 Abstraction

The basic idea of abstraction is simple: to abstract is to set aside what is less relevant and to focus on what we judge more important for the purpose in question. We focus on what persists and abstract away what varies from one instance to another. For example, in typical ML workflows, some elements (e.g. the training set or learning algorithm) will vary more often than others. To keep our code general, we hide those aspects that vary from one instance to another, while

[1]For an overview of software design, see e.g. Budgen [45].

exposing those aspects that remain constant between instances.[2]

In object-oriented programming, we can encapsulate the change-prone parts of the code in objects and define a set of common, less change-prone interface points around the change-prone internals. This will hide the change-prone parts inside the object's internal representation and prevent external code from referring to the change-prone details within the object. We discuss object-oriented programming and key ideas such as encapsulation and interfaces in appendix A.

B.2 Domain-driven design

The central idea of domain-driven design is to first create a conceptual model of the domain of interest [78, 257]. Every software program relates to some sphere of knowledge or activity of its user. That subject area to which the user applies the program is called the "domain" of the software. In this thesis, the domain of interest is the body of mathematical, statistical and computer science methodology that underlies much of ML methodology, which we will simply refer to as "ML theory".

The conceptual model is a system of abstractions that describe selected aspects of the domain and can be used to solve problems related to software design in that domain. The aim of the conceptual model is to identify and select key elements in the domain, to describe and abstract them, and to relate and organize them. In practice, this involves analyzing common ML techniques in their mathematical or algorithmic formulation, common workflows used by practitioners to solve particular problems, and typical user journeys for existing ML toolboxes. For example, in ML workflows, key elements include predictive models that are mathematical or statistical at their core (e.g. linear regression), inspection these models (e.g. "return coefficients of a fitted linear model"), and generating predictions from fitted models and assessing their predictive performance (e.g.

[2]The main reason for this "change-proneness" principle is that changes to software code account for much of its cost [142]. So, when following this principle, abstraction decision are always taken according to some expectation of about the cost of software development.

"estimate the mean squared error"). The software is then implemented to closely match the structure and language of the conceptual model. The key challenges then of course becomes finding an incisive conceptual model. We will address this challenge in the next chapter.

Below we highlight a key themes of domain-driven design as described by Evans [78] and their relevance for ML toolbox design:

- *Drawing on established formalism.* Through the conceptual analysis, domain-driven design can leverage existing formalism in the domain. In the context of ML, this include especially mathematical, statistical and computer science formalism. ML algorithms, interfaces and workflows are inextricably linked to their underlying mathematical and statistical methodology. As we will see in the next chapter, our conceptual analysis leverages existing mathematical formalism to drive software design.

- *Ubiquitous language.* Conceptual analysis helps clarify and distill knowledge from the domain of interest and creates a shared language for developers and practitioners. Because of the close correspondence between conceptual model and software implementation, developers can communicate with practitioners in this language without translation. The shared language allows developers and practitioners to collaborate and communicate more effectively. Discussions can then be used to further refine the conceptual model itself.

- *Intention-revealing interfaces.* If a user has to understand the implementation of a method in order to use it, much of the value of abstraction and encapsulation has been lost. Instead, methods and interfaces should describe their effect and purpose, without reference to their implementation details. Naming of methods should therefore following the ubiquitous language, so that users can understand implementations based on the shared conceptual model. We argue in the next chapter that our conceptual analysis, and scientific types for ML algorithms in particular, result in intention-revealing interfaces. This is because they are defined in correspondence to the learning tasks that they are meant to solve.

- *Cohesive mechanisms.* Computations sometimes reach a level of complexity that begins to obscure their purpose. The conceptual “what” is blotted out by the algorithmic “how” – the variety of ML algorithms and their use for distinct but related tasks being one example. Finding categories for these algorithms and exposing their capabilities as intention-revealing interfaces can create a cohesive mechanism, allowing practitioners to focus on the problem (“what”) and delegating the intricacies of the solution (“how”) to the algorithms. We hope to show in the next chapter that scientific types, especially when applied to ML algorithms, can become such a cohesive mechanism in ML toolbox design.

- *Declarative design.* Declarative design indicates a way to writing or using an application as a kind of executable specification. Making the learning task specification more explicit and developing intention-revealing interfaces for algorithms can be seen as a first step towards a more declarative language for specifying and solving ML problems.

- *Documenting design.* The conceptual model has the added advantage of providing a rationale for the software design after the fact. Deriving a design from a conceptual domain model records the design decisions that were made, and more importantly, why they were made that way. As a result, communicating the design choices to practitioners and developers will be easier.

B.3 Object-oriented design patterns

In this section, we review key object-oriented design patterns from classical software design as put forward in Gamma et al. [90]. We will refer to these patterns frequently throughout the next chapters of this thesis. We review key concepts from object-oriented programming itself in the next section.

Table B.1: An overview of key design patterns from classical software design

Name	Category	Description
Strategy	Behavioral	Define a family of algorithms, encapsulate each one and make them interchangeable. The strategy pattern lets algorithms of the same family vary independently from clients that use it.
Template	Behavioral	Define the skeleton of an algorithm in a method, deferring some steps to subclasses. The emplate pattern lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Composite	Structural	Compose objects to represent part-whole hierarchies. The composite pattern lets clients treat individual objects and compositions of objects uniformly.
Adapter	Structural	Adapt the interface of a class into another interface clients expect. Adapters let classes work together that could not otherwise because of incompatible interfaces.
Decorator	Structural	Assign additional methods to an object at run time. Decorators provide a flexible alternative to subclassing for extending functionality.
Facade	Structural	Provide a unified interface to a set of interfaces in a subsystem. The facade pattern defines a higher-level interface that makes the subsystem easier to use.
Factory	Creational	Define an interface for creating an object, but let subclasses decide which class to instantiate. The factory pattern lets a class defer instantiation to subclasses.
Prototype	Creational	Specify the kind of object(s) to create using a prototypical instance, and create new objects by copying this prototype.

Notes: A more detailed discussion of these design patterns can be found in Gamma et al. [90].

A design pattern describes a reusable solution to a recurring design problem within a given context. It is not a prefabricated piece of code as provided by a toolbox or a finished design as provided by a framework, but rather a description of how to solve the problem in many different situation. It describes a general solution in terms of classes, objects and their interactions, without referring to application specifics. Table B.1 gives an overview of the relevant patterns for our purpose.

Based on these generic software design patterns, we will formulate domain-specific patterns for ML toolbox design in Chapter 6.

Appendix C

Benchmarking Results from the M4 Competition

Table C.1: MASE percentage difference between reproduced and published results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Naïve	0.000	0.000	0.000	0.000	0.000	0.000
Naïve2	0.000	0.000	0.000	0.000	0.000	0.000
sNaïve	0.000	0.000	0.000	0.000	0.000	0.000
SES	−0.008	0.015	0.038	−0.002	0.004	0.000
Holt	5.586	0.001	2.805	−1.599	0.200	−3.927
Damped	3.696	−0.458	1.507	−2.887	0.814	−1.154
Com	2.751	−0.425	1.035	−1.900	0.334	−4.332
ARIMA	0.141	3.329	0.991	−11.172	−4.873	4.054
Theta	−3.058	−1.153	−0.139	−0.016	−0.041	0.095
Theta-bc	−1.922	−1.013	−0.202	−0.723	−0.035	−3.606
MLP	−14.658	−30.207	−41.370	−80.142	−70.993	−9.859
RNN	−24.522	−34.106	−30.284	−38.831	−36.420	−20.987

Notes: Rows show forecasters described in table 9.3. Columns show M4 data sets grouped by sampling frequency. Values show the percentage difference between reproduced and published MASE values relative to the published values. Negative values indicate that reproduced results are lower (better) than published ones.

Table C.2: OWA percentage difference between reproduced and published results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Naïve	0.000	0.000	0.000	0.000	0.000	0.000
Naïve2	0.000	0.000	0.000	0.000	0.000	0.000
sNaïve	0.000	0.000	0.000	0.000	0.000	0.000
SES	−0.006	0.042	0.027	−0.003	0.007	0.000
Holt	4.782	−0.812	3.382	−2.568	0.244	−4.048
Damped	2.594	−0.753	0.751	−1.729	0.926	−0.984
Com	2.179	−0.524	1.080	−1.646	0.416	−3.256
ARIMA	0.909	3.984	1.727	−5.082	−3.502	0.053
Theta	−2.267	−0.541	−0.031	0.081	0.008	0.053
Theta-bc	−1.416	−0.542	−0.147	−0.372	0.008	−0.308
MLP	−13.251	−27.165	−34.713	−71.246	−66.951	−7.691
RNN	−23.582	−31.657	−30.563	−32.746	−34.685	−16.490

Notes: Rows show forecasters described in table 9.3. Columns show M4 data sets grouped by sampling frequency. Values show the percentage difference between reproduced and published OWA values relative to the published values. Negative values indicate that reproduced results are lower (better) than published ones.

Table C.3: sMAPE difference between reproduced and published results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Naïve	-0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Naïve2	-0.0 ± 0.0	-0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
sNaïve	-0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
SES	-0.001 ± 0.006	0.007 ± 0.004	0.002 ± 0.001	-0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Holt	0.665 ± 0.083	-0.167 ± 0.034	0.580 ± 0.044	-0.327 ± 0.107	0.009 ± 0.013	-1.271 ± 0.463
Damped	0.241 ± 0.056	-0.105 ± 0.023	0.001 ± 0.020	-0.062 ± 0.042	0.032 ± 0.021	-0.151 ± 0.151
Com	0.246 ± 0.042	-0.063 ± 0.016	0.151 ± 0.016	-0.127 ± 0.037	0.015 ± 0.007	-0.339 ± 0.171
ARIMA	0.245 ± 0.058	0.477 ± 0.038	0.325 ± 0.035	0.074 ± 0.285	-0.068 ± 0.023	-0.282 ± 0.389
Theta	-0.221 ± 0.017	0.005 ± 0.007	0.010 ± 0.004	0.016 ± 0.006	0.002 ± 0.002	0.001 ± 0.000
Theta-bc	-0.127 ± 0.017	-0.010 ± 0.005	-0.012 ± 0.004	-0.004 ± 0.006	0.002 ± 0.001	0.593 ± 0.242
MLP	-2.598 ± 0.048	-4.460 ± 0.062	-6.708 ± 0.062	-11.229 ± 1.046	-5.754 ± 0.179	-0.631 ± 0.127
RNN	-5.091 ± 0.091	-4.994 ± 0.073	-7.413 ± 0.089	-3.954 ± 0.785	-1.968 ± 0.113	-1.372 ± 0.383

Notes: Rows show forecasters described in table 9.3. Columns show M4 data sets grouped by sampling frequency. Values show the difference between reproduced and published mean sMAPE values, together with the standard error of the difference in means between paired samples. Values in bold indicate that the difference is statistically significant at the 95% level based on a two-sided paired t-test. Negative values indicate that reproduced results are lower than published ones.

Table C.4: MASE difference between reproduced and published results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Naïve	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Naïve2	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
sNaïve	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
SES	−0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0	−0.0 ± 0.0	0.0 ± 0.0	0.0 ± 0.0
Holt	0.198 ± 0.012	0.00 ± 0.02	0.028 ± 0.002	−0.039 ± 0.012	0.006 ± 0.009	−0.367 ± 0.168
Damped	0.125 ± 0.008	−0.005 ± 0.002	0.015 ± 0.005	−0.069 ± 0.023	0.026 ± 0.024	−0.034 ± 0.116
Com	0.090 ± 0.006	−0.005 ± 0.001	0.010 ± 0.002	−0.046 ± 0.012	0.011 ± 0.007	−0.199 ± 0.062
ARIMA	0.005 ± 0.010	0.039 ± 0.003	0.009 ± 0.002	−0.286 ± 0.115	−0.166 ± 0.024	0.038 ± 0.024
Theta	−0.103 ± 0.002	−0.014 ± 0.000	−0.001 ± 0.000	−0.00 ± 0.03	−0.001 ± 0.001	0.002 ± 0.000
Theta-bc	−0.058 ± 0.005	−0.012 ± 0.001	−0.002 ± 0.000	−0.019 ± 0.008	−0.001 ± 0.001	−0.092 ± 0.050
MLP	−0.725 ± 0.027	−0.699 ± 0.019	−0.796 ± 0.045	−10.874 ± 5.053	−9.210 ± 1.916	−0.257 ± 0.033
RNN	−1.213 ± 0.028	−0.688 ± 0.008	−0.485 ± 0.005	−1.993 ± 0.263	−2.270 ± 0.114	−0.640 ± 0.248

Notes: Rows show forecasters described in table 9.3. Columns show M4 data sets grouped by sampling frequency. Values show the difference between reproduced and published mean MASE values, together with the standard error of the difference in means between paired samples. Values in bold indicate that the difference is statistically significant at the 95% level based on a two-sided paired t-test. Negative values indicate that reproduced results are lower than published ones.

Table C.5: sMAPE results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Theta-bc	**13.239**	10.145	**12.99**	9.144	3.042	18.16
Theta	14.372	10.316	13.012	9.109	3.055	18.14
Com	15.094	**10.112**	13.585	8.817	**2.995**	21.714
RF-Theta-bc-t	14.552	10.788	13.543	9.014	3.056	18.136
Damped	15.439	10.132	13.475	8.804	3.096	19.114
RF-Theta-bc	14.738	11.006	13.527	9.801	3.061	18.061
ARIMA	15.413	10.908	13.768	8.727	3.124	13.698
SES	16.395	10.607	13.62	9.011	3.045	18.094
XGB-Theta-bc-t	15.334	11.535	14.166	9.484	3.134	18.303
XGB-Theta-bc	15.395	11.651	14.109	10.808	3.16	18.199
Naïve2	16.342	11.012	14.427	9.161	3.045	18.383
KNN-Theta-bc	15.503	12.046	14.805	11.133	3.189	19.32
KNN-Theta-bc-t	15.507	12.036	14.877	10.17	3.189	19.361
RF-s	16.655	11.674	14.899	9.327	3.291	10.978
RF	16.656	11.793	15.01	9.31	3.289	13.663
Holt	17.018	10.74	15.393	9.381	3.075	27.978
Naïve	16.342	11.61	15.256	9.161	3.045	43.003
sNaïve	16.342	12.521	15.988	9.161	3.045	13.912
RF-t-s	17.31	12.321	16.008	9.009	3.321	15.797
RNN	17.307	12.033	16.643	11.266	3.996	13.326
XGB-s	17.729	12.633	17.158	10.201	3.512	**10.912**
XGB	17.729	12.671	17.259	10.201	3.512	14.288
LR-s	-	12.235	17.768	9.519	3.32	11.235
LR	-	12.238	17.797	9.519	3.32	16.342
XGB-t-s	17.954	13.21	18.102	9.462	3.608	17.195
MLP	19.166	14.04	17.625	10.12	3.568	13.211
KNN-t-s	18.37	13.701	19.711	10.482	4.062	16.333
KNN-s	18.758	13.548	19.764	11.167	4.08	11.988
KNN	18.758	13.606	19.905	11.167	4.08	14.61
LR-t-s	-	-	18.619	**8.567**	3.363	16.76

Notes: Rows show forecasters described in Table 9.6. Columns show M4 data sets grouped by sampling frequency. We exclude LR model results when generated forecasts were instable/exploding, likely due the little available data and linear extrapolation.

Table C.6: MASE results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Theta-bc	**2.951**	1.186	0.964	2.582	3.252	2.465
Theta	3.279	1.218	0.968	2.637	3.261	2.457
RF-Theta-bc-t	3.246	1.229	0.985	2.592	3.257	2.363
ARIMA	3.407	1.204	**0.939**	**2.27**	3.244	0.981
RF-Theta-bc	3.282	1.253	0.984	2.87	**3.18**	2.365
Com	3.371	**1.168**	0.976	2.386	3.213	4.383
Damped	3.504	**1.168**	0.987	2.334	3.262	2.922
XGB-Theta-bc	3.392	1.338	1.025	3.127	3.309	2.387
XGB-Theta-bc-t	3.476	1.332	1.032	2.713	3.446	2.398
KNN-Theta-bc	3.394	1.366	1.069	3.01	3.365	2.52
KNN-Theta-bc-t	3.396	1.368	1.083	2.842	3.4	2.529
RF-s	3.639	1.327	1.016	2.809	3.523	**0.93**
RF	3.64	1.341	1.037	2.823	3.521	1.032
Holt	3.748	1.198	1.038	2.381	3.23	8.988
RF-t-s	3.779	1.399	1.068	2.639	3.553	1.212
SES	3.98	1.34	1.02	2.684	3.281	2.385
RNN	3.733	1.329	1.116	3.139	3.962	2.408
Naïve2	3.974	1.371	1.063	2.777	3.278	2.395
XGB-s	3.814	1.42	1.113	3.091	3.754	0.954
XGB	3.814	1.431	1.129	3.091	3.754	1.063
XGB-t-s	3.883	1.493	1.154	2.857	3.847	1.335
Naïve	3.974	1.477	1.205	2.777	3.278	11.608
sNaïve	3.974	1.602	1.26	2.777	3.278	1.193
MLP	4.221	1.615	1.129	2.694	3.763	2.35
KNN-s	4.072	1.524	1.217	3.378	4.38	1.044
KNN-t-s	3.977	1.543	1.261	3.192	4.355	1.471
KNN	4.072	1.534	1.239	3.378	4.38	1.112
LR-s	-	1.319	3.284	2.467	3.442	0.934
LR	-	1.327	3.301	2.467	3.442	1.001
LR-t-s	-	-	2.618	2.338	3.518	1.027

Notes: Rows show forecasters described in Table 9.6. Columns show M4 data sets grouped by sampling frequency. We exclude LR model results when generated forecasts were instable/exploding, likely due the little available data and linear extrapolation.

Table C.7: OWA results

	Yearly	Quarterly	Monthly	Weekly	Daily	Hourly
Theta-bc	**0.776**	0.893	**0.904**	0.964	0.996	1.009
Theta	0.852	0.912	0.906	0.972	0.999	1.006
Com	0.886	**0.885**	0.93	0.911	**0.982**	1.506
RF-Theta-bc-t	0.854	0.938	0.933	0.959	0.999	0.987
ARIMA	0.9	0.934	0.919	**0.885**	1.008	0.577
Damped	0.913	0.886	0.931	0.901	1.006	1.13
RF-Theta-bc	0.864	0.957	0.932	1.052	0.988	0.985
XGB-Theta-bc	0.898	1.017	0.971	1.153	1.023	0.993
SES	1.002	0.97	0.952	0.975	1.0	0.99
XGB-Theta-bc-t	0.906	1.009	0.976	1.006	1.04	0.998
RF-s	0.967	1.014	0.994	1.015	1.078	**0.493**
Holt	0.992	0.924	1.021	0.941	0.997	2.637
KNN-Theta-bc	0.901	1.045	1.016	1.149	1.037	1.052
Naïve2	1.0	1.0	1.0	1.0	1.0	1.0
KNN-Theta-bc-t	0.902	1.045	1.025	1.067	1.042	1.055
RF	0.967	1.025	1.008	1.016	1.077	0.587
RF-t-s	1.005	1.07	1.057	0.967	1.087	0.683
RNN	0.999	1.031	1.102	1.18	1.26	0.865
Naïve	1.0	1.066	1.095	1.0	1.0	3.593
XGB-s	1.022	1.091	1.118	1.113	1.149	0.496
XGB	1.022	1.097	1.129	1.113	1.149	0.611
sNaïve	1.0	1.153	1.146	1.0	1.0	0.628
XGB-t-s	1.038	1.144	1.17	1.031	1.179	0.746
MLP	1.117	1.226	1.142	1.037	1.16	0.85
KNN-s	1.086	1.171	1.257	1.218	1.338	0.544
KNN-t-s	1.062	1.185	1.276	1.147	1.331	0.751
KNN	1.086	1.177	1.272	1.218	1.338	0.63
LR-s	-	1.037	2.16	0.964	1.07	0.501
LR	-	1.039	2.169	0.964	1.07	0.654
LR-t-s	-	-	1.876	0.889	1.089	0.67

Notes: Rows show forecasters described in Table 9.6. Columns show M4 data sets grouped by sampling frequency. We exclude LR model results when generated forecasts were instable/exploding, likely due the little available data and linear extrapolation.

Table C.8: Summary results of new machine learning models

	sMAPE	MASE	OWA	Running time (min)
Theta-bc	**11.952**	**1.583**	**0.876**	8.1
Theta	12.264	1.669	0.9	6.27
Com	12.668	1.687	0.914	69.47
RF-Theta-bc-t	12.673	1.671	0.919	14539.77
ARIMA	12.992	1.673	0.92	14992.88
Damped	12.692	1.718	0.92	53.45
RF-Theta-bc	12.763	1.682	0.925	1189.37
XGB-Theta-bc	13.357	1.754	0.968	122.94
SES	13.09	1.885	0.97	5.9
XGB-Theta-bc-t	13.337	1.78	0.971	1826.44
RF-s	14.002	1.806	0.994	1168.56
Holt	14.16	1.83	0.997	11.72
KNN-Theta-bc	13.818	1.785	0.998	64.36
Naïve2	13.564	1.912	1.0	3.66
KNN-Theta-bc-t	13.848	1.794	1.003	656.04
RF	14.095	1.82	1.004	1163.64
RF-t-s	14.86	1.882	1.047	14165.87
RNN	15.122	1.902	1.067	38941.68
Naïve	14.208	2.044	1.072	1.04
XGB-s	15.576	1.926	1.088	116.95
XGB	15.647	1.937	1.096	115.76
sNaïve	14.657	2.057	1.105	**1.03**
XGB-t-s	16.246	1.983	1.131	1718.15
MLP	16.48	2.079	1.156	157.88
KNN-s	17.315	2.088	1.197	58.71
KNN-t-s	17.252	2.092	1.205	634.12
KNN	17.407	2.101	1.207	56.99
LR-s	-	-	-	55.36
LR	-	-	-	53.37
LR-t-s	-	-	-	590.87

Notes: Rows show forecasters described in Table 9.6. Columns show aggregate values for sMAPE, MASE and OWA metrics, as well as the total running time in minutes scaled to the number of CPUs used in the original M4 competition, as described in Chapter 9.6. We exclude LR model results when generated forecasts were instable/exploding, likely due the little available data and linear extrapolation.

Table C.9: Post-hoc Wilcoxon signed rank tests results for pairwise comparisons on the hourly data set

	forecaster A	forecaster B	p-value
0	LR-s	RF-s	0.657
4	LR-s	XGB-s	0.243
3	M4 runner-up	M4 winner	0.317
2	M4 runner-up	RF-s	0.468
1	RF-s	XGB-s	0.010

Notes: This table corresponds to the results presented in Figure 9.1. The results are based on sMAPE and the hourly data set. We only show pairs of forecasters which are not significantly different. Forecasters are described in Table 9.6. We use Wilcoxon signed rank tests to compute p-values. Significance at the 5% is established using Holm's procedure for correcting for multiple testing, as discussed in Section 9.3.

Table C.10: Post-hoc Wilcoxon signed rank tests results for pairwise comparisons on the hourly data set

	forecaster A	forecaster B	p-value
2	RF-Theta-bc	RF-Theta-bc-t	0.096
1	RF-Theta-bc-t	XGB-Theta-bc	0.092
3	Theta-bc	XGB-Theta-bc-t	0.527
0	XGB-Theta-bc	XGB-Theta-bc-t	0.028

Notes: This table corresponds to the results presented in Figure 9.2. The results are based on sMAPE and the hourly data set. We only show pairs of forecasters which are not significantly different. Forecasters are described in Table 9.6. We use Wilcoxon signed rank tests to compute p-values. Significance at the 5% is established using Holm's procedure for correcting for multiple testing, as discussed in Section 9.3.

Bibliography

[1] Martin Abadi et al. ‘TensorFlow: A system for large-scale machine learning’. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 2016, pp. 265–283.

[2] Saeed Aghabozorgi, Ali Seyed Shirkhorshidi and Teh Ying Wah. ‘Time-series clustering - A decade review’. In: *Information Systems* (2015). ISSN: 03064379.

[3] Nesreen K. Ahmed, Amir F Atiya, Neamat El Gayar and Hisham El-Shishiny. ‘An empirical comparison of machine learning models for time series forecasting’. In: *Econometric Reviews* 29.5-6 (2010), pp. 594–621. ISSN: 07474938.

[4] Alexander Alexandrov et al. ‘GluonTS: Probabilistic and Neural Time Series Modeling in Python’. In: *Journal of Machine Learning Research* 21.116 (2020), pp. 1–6.

[5] Samaneh Aminikhanghahi and Diane J Cook. ‘A survey of methods for time series change point detection’. In: *Knowledge and information systems* 51.2 (2017), pp. 339–367.

[6] Oren Anava, Elad Hazan, Shie Mannor and Ohad Shamir. ‘Online learning for time series prediction’. In: *Journal of Machine Learning Research.* 2013.

[7] Becky Arnold et al. *The Turing Way: A Handbook for Reproducible Data Science.* Mar. 2019. URL: `10.5281/zenodo.3233853`.

[8] *arundo/adtk v0.6.2.* 2021. URL: `https://github.com/arundo/adtk`.

[9] Vassilis Assimakopoulos and Konstantinos Nikolopoulos. ‘The theta model: a decomposition approach to forecasting’. In: *International journal of forecasting* 16.4 (2000), pp. 521–530.

[10] George Athanasopoulos, Rob J Hyndman, Haiyan Song and Doris C Wu. 'The tourism forecasting competition'. In: *International Journal of Forecasting* 27.3 (2011), pp. 822–844.

[11] Jono Bacon. *The art of community: Building the new age of participation.* O'Reilly Media, Inc., 2012.

[12] Anthony Bagnall, Michael Flynn, James Large, Jason Lines and Matthew Middlehurst. 'A tale of two toolkits, report the third: on the usage and performance of HIVE-COTE v1.0'. In: *arXiv preprint* (Apr. 2020). URL: `http://arxiv.org/abs/2004.06069`.

[13] Anthony Bagnall, James Large and Matthew Middlehurst. 'A tale of two toolkits, report the second: bake off redux. Chapter 1. dictionary based classifiers'. In: *arXiv preprint* (2019).

[14] Anthony Bagnall, Jason Lines, Aaron Bostrom, James Large and Eamonn Keogh. 'The great time series classification bake off: a review and experimental evaluation of recent algorithmic advances'. In: *Data Mining and Knowledge Discovery* 31.3 (May 2017), pp. 606–660. ISSN: 1384-5810. URL: `http://link.springer.com/10.1007/s10618-016-0483-9`.

[15] Anthony Bagnall et al. 'A tale of two toolkits, report the first: benchmarking time series classification algorithms for correctness and efficiency'. In: *arXiv preprint arXiv:1909.05738* (2019).

[16] Badi H. Baltagi. *Econometric Analysis of Panel Data.* 4th ed. chichester: John Wiley & Sons, 2008, p. 351. ISBN: 9780470518861.

[17] *bashtage/linearmodels v4.19.* URL: `https://github.com/bashtage/linearmodels/`.

[18] Len Bass, Paul Clements and Rick Kazman. *Software architecture in practice.* Addison-Wesley Professional, 2003.

[19] Kent Beck and Ralph Johnson. 'Patterns generate architectures'. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* 1994. ISBN: 9783540582021.

[20] Kent Beck et al. 'Manifesto for agile software development'. In: (2001).

[21] Stefan Behnel et al. 'Cython: The best of both worlds'. In: *Computing in Science & Engineering* 13.2 (2011), pp. 31–39.

[22] Alessio Benavoli, Giorgio Corani and Francesca Mangili. 'Should we really use post-hoc tests based on mean-ranks?' In: *Journal of Machine Learning Research* (2016). ISSN: 15337928.

[23] Christoph Bergmeir and José M. Benítez. 'On the use of cross-validation for time series predictor evaluation'. In: *Information Sciences* (2012). ISSN: 00200255.

[24] Alina Beygelzimer, Hal Daumé, John Langford and Paul Mineiro. 'Learning reductions that really work'. In: *Proceedings of the IEEE* 104.1 (2015), pp. 136–147.

[25] Alina Beygelzimer, John Langford and Bianca Zadrozny. 'Weighted one-against-all'. In: *American Association for Artificial Intelligence (AAAI)*. 2005, pp. 720–725.

[26] Alina Beygelzimer, John Langford and Bianca Zadrozny. 'Machine learning techniques—reductions between prediction quality metrics'. In: *Performance Modeling and Engineering*. Springer, 2008, pp. 3–28.

[27] Jeff Bezanson, Alan Edelman, Stefan Karpinski and Viral B Shah. 'Julia: A fresh approach to numerical computing'. In: *SIAM review* 59.1 (2017), pp. 65–98.

[28] Bernd Bischl et al. 'mlr: Machine Learning in R'. In: *Journal of Machine Learning Research* 17.170 (2016), pp. 1–5. URL: `http://jmlr.org/papers/v17/15-066.html`.

[29] Christopher M Bishop. *Pattern Recognition and Machine Learning*. Vol. 4. 4. 2006, p. 738. ISBN: 9780387310732. URL: `http://www.library.wisc.edu/selectedtocs/bg0137.pdf`.

[30] Anthony Blaom et al. *MLJ: A Machine Learning Framework for Julia*. Apr. 2020. URL: `https://zenodo.org/record/3765808`.

[31] Anthony D Blaom et al. 'MLJ: A Julia package for composable machine learning'. In: *Journal of Open Source Software* 5.55 (2020), p. 2704. URL: `https://doi.org/10.21105/joss.02704`.

[32] Tim Bollerslev. 'Generalized autoregressive conditional heteroskedasticity'. In: *Journal of econometrics* 31.3 (1986), pp. 307–327.

[33] Gianluca Bontempi. *Comments on M4 competition.* 2020.

[34] Gianluca Bontempi, Souhaib Ben Taieb and Yann-Aël Le Borgne. 'Machine Learning Strategies for Time Series Forecasting'. In: *Business Intelligence.* Springer, Berlin, Heidelberg, 2013, pp. 62–77.

[35] A Bostrom and Anthony Bagnall. 'Binary Shapelet Transform for Multiclass Time Series Classification'. In: *Proc. 17th International Conference on Big Data Analytics and Knowledge Discovery {(DAWAK)}* 32 (2015), pp. 24–46.

[36] George E. P. Box. 'Science and Statistics'. In: *Journal of the American Statistical Association* 71.356 (Dec. 1976), p. 791. ISSN: 01621459. URL: `https://www.jstor.org/stable/2286841?origin=crossref`.

[37] George E. P. Box and D. R. Cox. 'An Analysis of Transformations'. In: *Journal of the Royal Statistical Society: Series B (Methodological)* (1964).

[38] George E. P. Box, Gwilym M. Jenkins and Gregory C. Reinsel. *Time series analysis: Forecasting and control: Fourth edition.* 2013. ISBN: 9781118619193.

[39] George E. P. Box, Gwilym M. Jenkins, Gregory C. Reinsel and Greta M. Ljung. *Time series analysis: forecasting and control.* John Wiley & Sons, 2015. ISBN: 9781118674925.

[40] John E. Boylan, Paul Goodwin, Maryam Mohammadipour and Aris A. Syntetos. 'Reproducibility in forecasting research'. In: *International Journal of Forecasting* (2015). ISSN: 01692070.

[41] Leo Breiman. 'Statistical Modeling: The Two Cultures'. In: *Statistical Science* 16.3 (2001), pp. 199–231. ISSN: 08834237.

[42] *Tackling Hidden Hunger through Soils.* The Alan Turing Institute & Rothamsted Research. 2020. URL: `https://doi.org/10.5281/zenodo.3775489`.

[43] Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting.* Springer Texts in Statistics. Springer International Publishing, 2016. ISBN: 978-3-319-29852-8. URL: `http://link.springer.com/10.1007/978-3-319-29854-2`.

[44] William J Brown, Raphael C Malveau, Thomas J Mowbray and John Wiley. *AntiPatterns: Refactoring Software , Architectures, and Projects in Crisis.* Wiley, 1998. ISBN: 0849329949.

[45] David Budgen. *Software design.* Pearson Education, 2003.

[46] Lars Buitinck et al. 'API design for machine learning software: experiences from the scikit-learn project'. In: *ArXiv e-prints* (2013). URL: `https://github.com/scikit-learn`.

[47] David M Burns and Cari M Whyne. 'Seglearn: a python package for learning sequences and time series'. In: *The Journal of Machine Learning Research* 19.1 (2018), pp. 3238–3244.

[48] Frank Buschmann, Kelvin Henney and Douglas Schimdt. *Pattern-Oriented Software Architecture: On Patterns And Pattern Language, Volume 5.* Vol. 5. John wiley & sons, 2007.

[49] Matthias Bussonnier et al. 'Binder 2.0 - Reproducible, interactive, sharable environments for science at scale'. In: *Proceedings of the 17th Python in Science Conference.* Ed. by Fatih Akici, David Lippa, Dillon Niederhut and M Pacer. 2018, pp. 113–120.

[50] John M Chambers et al. 'Object-oriented programming, functional programming and R'. In: *Statistical Science* 29.2 (2014), pp. 167–180.

[51] Ngai Hang Chan. *Time series: applications to finance.* Vol. 487. John Wiley & Sons, 2004.

[52] Varun Chandola, Arindam Banerjee and Vipin Kumar. 'Anomaly detection: A survey'. In: *ACM computing surveys (CSUR)* 41.3 (2009), pp. 1–58.

[53] Tianqi Chen and Carlos Guestrin. 'XGBoost: A Scalable Tree Boosting System'. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining.* KDD '16. New York, NY, USA: ACM, 2016, pp. 785–794. ISBN: 978-1-4503-4232-2. URL: `http://doi.acm.org/10.1145/2939672.2939785`.

[54] François Chollet et al. *Keras.* https://keras.io. 2015.

[55] Maximilian Christ, Nils Braun, Julius Neuffer and Andreas W. Kempa-Liehr. 'Time Series FeatuRe Extraction on basis of Scalable Hypothesis tests (tsfresh – A Python package)'. In: *Neurocomputing* 307 (2018), pp. 72–77. ISSN: 0925-2312. URL: `https://www.sciencedirect.com/science/article/pii/S0925231218304843?via%7B%5C%%7D3Dihub`.

[56] Maximilian Christ, Andreas W. Kempa-Liehr and Michael Feindt. 'Distributed and parallel time series feature extraction for industrial big data applications'. In: *arXiv preprint* (Oct. 2016). URL: `http://arxiv.org/abs/1610.07717`.

[57] Robert B Cleveland, William S Cleveland, Jean E McRae and Irma Terpenning. 'STL: A seasonal-trend decomposition'. In: *Journal of official statistics* 6.1 (1990), pp. 3–73.

[58] Alistair Cockburn. *Agile software development: the cooperative game*. Pearson Education, 2006.

[59] D. R. Cox. 'Applied Statistics: A Review'. In: *The Annals of Applied Statistics* 1.1 (2007), pp. 1–16.

[60] D. R. Cox and David Oakes. *Analysis of Survival Data*. Chapman and Hall, 1984, p. 201. ISBN: 9780412244902.

[61] David Roxbee Cox. *Planning of Experiments*. Wiley, 1992.

[62] Drew Creal, Siem Jan Koopman and André Lucas. 'Generalized autoregressive score models with applications'. In: *Journal of Applied Econometrics* 28.5 (2013), pp. 777–795.

[63] Sven F Crone, Michele Hibon and Konstantinos Nikolopoulos. 'Advances in forecasting with neural networks? Empirical evidence from the NN3 competition on time series prediction'. In: *International Journal of forecasting* 27.3 (2011), pp. 635–660.

[64] Daryl J. Daley and David Vere-Jones. *An Introduction to the Theory of Point Processes*. 2nd. New York: Springer, 2003, pp. xviii+573. ISBN: 0-387-95541-0. URL: `http://link.springer.com/10.1007/b97277`.

[65] Jan G De Gooijer and Rob J Hyndman. '25 years of time series forecasting'. In: *International journal of forecasting* 22.3 (2006), pp. 443–473.

[66] Marcos Lopez De Prado. *Advances in financial machine learning.* John Wiley & Sons, 2018.

[67] Peter DeGrace and Leslie Hulet Stahl. *Wicked problems, righteous solutions.* Yourdon Press, 1990.

[68] Angus Dempster, François Petitjean and Geoffrey I Webb. 'ROCKET: exceptionally fast and accurate time series classification using random convolutional kernels'. In: *Data Mining and Knowledge Discovery* 34.5 (2020), pp. 1454–1495.

[69] Angus Dempster, Daniel F Schmidt and Geoffrey I Webb. 'MINIROCKET: A Very Fast (Almost) Deterministic Transform for Time Series Classification'. In: *arXiv preprint arXiv:2012.08791* (2020).

[70] Janez Demšar. 'Statistical Comparisons of Classifiers over Multiple Data Sets'. In: *Journal of Machine Learning Research* 7 (2006), pp. 1–30. ISSN: 1532-4435.

[71] Houtao Deng, George Runger, Eugene Tuv and Martyanov Vladimir. 'A time series forest for classification and feature extraction'. In: *Information Sciences* 239 (2013), pp. 142–153.

[72] Thomas G. Dietterich. 'Approximate Statistical Tests for Comparing Supervised Classification Learning Algorithms'. In: *Neural Computation* 10.7 (1998), pp. 1895–1923. URL: `http://www.ncbi.nlm.nih.gov/pubmed/9744903`.

[73] Thomas G. Dietterich. 'Machine Learning for Sequential Data: A Review'. In: *Structural, Syntactic, and Statistical Pattern Recognition.* Ed. by Caelli T., Amin A., Duin R.P.W., de Ridder D. and Kamel M. Springer, Berlin, Heidelberg, 2002, pp. 15–30. URL: `http://link.springer.com/10.1007/3-540-70659-3%7B%5C_%7D2`.

[74] Peter Diggle, Patrick Heagerty, Kung-Yee Liang and Scott Zeger. *Analysis of longitudinal data.* 2nd. Oxford University Press, 2013. ISBN: 0199676755. URL: `https://books.google.co.uk/books?id=ur0BlXPu0ukC%7B%5C&%7Dprintsec=frontcover%7B%5C&%7Ddq=Diggle,+P.,+P.+J.+Diggle,+P.+Heagerty,+P.+J.+Heagerty,+K.-Y.+Liang,+S.+Zeger,+et+al.+(2013).+Analysis+of+Longitudinal+Data+(2+ed.).+Oxford+University+Press.%7B%5C&%7Dhl=en%7B%5C&%7Dsa=X%7B%5C&%7Dved=0ahUKEwisyM`.

[75] David Donoho. '50 Years of Data Science'. In: *Journal of Computational and Graphical Statistics* 26.4 (Oct. 2017), pp. 745–766. ISSN: 1061-8600. URL: `https://www.tandfonline.com/doi/full/10.1080/10618600.2017.1384734`.

[76] Chris Drummond. 'Replicability is not reproducibility: nor is it good science'. In: (2009).

[77] Philippe Esling and Carlos Agon. 'Time-series data mining'. In: *ACM Computing Surveys* 45.1 (Nov. 2012), pp. 1–34. ISSN: 03600300. URL: `http://dl.acm.org/citation.cfm?doid=2379776.2379788`.

[78] Eric Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004.

[79] Amin Fakhrazari and Hamid Vakilzadian. 'A survey on time series data mining'. In: *2017 IEEE International Conference on Electro Information Technology (EIT).* IEEE. 2017, pp. 476–481.

[80] Johann Faouzi and Hicham Janati. 'pyts: A Python Package for Time Series Classification'. In: *Journal of Machine Learning Research* 21.46 (2020), pp. 1–6.

[81] Jose A. Fiorucci, Tiago R. Pellegrini, Francisco Louzada, Fotios Petropoulos and Anne B. Koehler. 'Models for aoptimising the theta method and their relationship to state space models'. In: *International Journal of Forecasting* (2016). ISSN: 01692070.

[82] Eric Freeman, Elisabeth Robson, Bert Bates and Kathy Sierra. *Head first design patterns.* " O'Reilly Media, Inc.", 2008.

[83] Jerome H Friedman. 'Data Mining and Statistics: What's the connection?' In: *Computing science and statistics* 29.1 (1998), pp. 3–9.

[84] Milton Friedman. 'A comparison of alternative tests of significance for the problem of m rankings'. In: *The Annals of Mathematical Statistics* 11.1 (1940), pp. 86–92.

[85] Chris Fry and Michael Brundage. *The M4 forecasting competition – A practitioner's view.* 2020.

[86] Ben Fulcher et al. 'benfulcher/hctsa: v1.05'. In: (Jan. 2021). URL: `https://doi.org/10.5281/zenodo.4426985%7B%5C#%7D.YBf6ly3-qWY.mendeley`.

[87] Ben D Fulcher and Nick S Jones. 'Highly Comparative Feature-Based Time-Series Classification'. In: *IEEE Transactions on Knowledge and Data Engineering* 26.12 (Dec. 2014), pp. 3026–3037. ISSN: 1041-4347. URL: `http://ieeexplore.ieee.org/document/6786425/`.

[88] Ben D Fulcher and Nick S Jones. 'hctsa: A Computational Framework for Automated Time-Series Phenotyping Using Massive Feature Extraction.' In: *Cell systems* 5.5 (Nov. 2017), 527–531.e3. ISSN: 2405-4712. URL: `http://www.ncbi.nlm.nih.gov/pubmed/29102608`.

[89] Cristina Gacek and Budi Arief. 'The many meanings of open source'. In: *IEEE software* 21.1 (2004), pp. 34–40.

[90] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison Wesley Professional, 1997.

[91] Salvador García and Francisco Herrera. 'An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons'. In: *Journal of Machine Learning Research* (2008). ISSN: 15324435.

[92] Everette S Gardner and E D McKenzie. 'Forecasting trends in time series'. In: *Management Science* 31.10 (1985), pp. 1237–1246.

[93] Andrew Gelman and Aki Vehtari. 'What are the most important statistical ideas of the past 50 years?' In: *arXiv preprint arXiv:2012.00174* (2020).

[94] Michael Gilliland. *The value added by machine learning approaches in forecasting*. 2020.

[95] Leilani H Gilpin et al. 'Explaining explanations: An overview of interpretability of machine learning'. In: *2018 IEEE 5th International Conference on data science and advanced analytics (DSAA)*. IEEE. 2018, pp. 80–89.

[96] Steven N. Goodman, Daniele Fanelli and John P.A. Ioannidis. 'What does research reproducibility mean?' In: *Getting to Good: Research Integrity in the Biomedical Sciences*. 2018. ISBN: 9783319513584.

[97] Christian Gourieroux, Michael Wickens, Eric Ghysels and Richard J Smith. *Applied time series econometrics*. Cambridge university press, 2004.

[98] Ahmed Guecioueur. *pysf: Supervised forecasting of sequential data in Python.* 2018. URL: `https://pypi.org/project/pysf/`.

[99] Isabelle Guyon. *A Practical Guide to Model Selection.* 2009. URL: `http://eprints.pascal-network.org/archive/00005768/`.

[100] V. Haenel, E. Gouillart and Gaël Varoquaux. *Python scientific lecture notes.* 2013. URL: `http://scipy-lectures.github.io/`.

[101] Mark Hall et al. 'The WEKA data mining software'. In: *ACM SIGKDD Explorations Newsletter* 11.1 (Nov. 2009), p. 10. ISSN: 19310145. URL: `http://portal.acm.org/citation.cfm?doid=1656274.1656278`.

[102] Mark Hamilton. *tseries: a library for time series analysis with sklearn.* Sept. 2017. URL: `https://zenodo.org/record/897193`.

[103] David J Hand. 'Data mining: statistics and more?' In: *The American Statistician* 52.2 (1998), pp. 112–118.

[104] Charles R. Harris et al. *Array programming with NumPy.* 2020.

[105] Trevor Hastie, Robert T. Tibshirani and Jerome Friedman. *The Elements of Statistical Learning.* 2nd ed. Vol. 1. Springer, 2009, pp. 1–694. ISBN: 978-0-387-84857-0. URL: `http://www.springerlink.com/index/10.1007/b94608`.

[106] *heidelbergcement/hcrystalball v0.1.10.* 2021. URL: `https://github.com/heidelbergcement/hcrystalball`.

[107] Hansika Hewamalage, Christoph Bergmeir and Kasun Bandara. 'Recurrent neural networks for time series forecasting: Current status and future directions'. In: *arXiv preprint arXiv:1909.00590* (2019).

[108] James A Highsmith and Jim Highsmith. *Agile software development ecosystems.* Addison-Wesley Professional, 2002.

[109] Sepp Hochreiter and Jürgen Schmidhuber. 'Long Short-Term Memory'. In: *Neural Computation* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. URL: `http://www.mitpressjournals.org/doi/10.1162/neco.1997.9.8.1735`.

[110] Victoria Hodge and Jim Austin. 'A survey of outlier detection methodologies'. In: *Artificial intelligence review* 22.2 (2004), pp. 85–126.

[111] S Holm. 'A simple sequentially rejective multiple test procedure'. In: *Scandinavian journal of statistics* (1979).

[112] G. Holmes, A. Donkin and I.H. Witten. 'WEKA: a machine learning workbench'. In: *Proceedings of ANZIIS '94 - Australian New Zealnd Intelligent Information Systems Conference*. IEEE, 1994, pp. 357–361. ISBN: 0-7803-2404-8. URL: `http://ieeexplore.ieee.org/document/396988/`.

[113] Charles C Holt. 'Forecasting trends and seasonal by exponentially weighted moving averages'. In: *ONR Memorandum* 52 (1957).

[114] Charles C Holt. 'Forecasting seasonals and trends by exponentially weighted moving averages'. In: *International journal of forecasting* 20.1 (2004), pp. 5–10.

[115] Tao Hong, Rob J Hyndman, Roman A. Ahmed, Han Lin Shang and Earo Wang. 'hts : An R Package for Forecasting Hierarchical or Grouped Time Series'. In: *Iranian Journal of Electrical and Electronic Engineering* 55.June (2014), pp. 146–166. ISSN: 01679473. URL: `http://dx.doi.org/10.1016/j.csda.2011.03.006%7B%5C%%7D5Cnhttp://www.lancaster.ac.uk/%7B~%7Dmorganle/images/HierarchicalModels.pdf%7B%5C%%7D5Cnhttp://robjhyndman.com/papers/hgts6.pdf%7B%5C%%7D5Cnhttp://dx.doi.org/10.1016/j.ijforecast.2008.07.004`.

[116] Tao Hong et al. *Probabilistic energy forecasting: Global Energy Forecasting Competition 2014 and beyond*. 2016.

[117] Jeremy Howard and Sylvain Gugger. 'Fastai: A layered api for deep learning'. In: *Information (Switzerland)* (2020). ISSN: 20782489.

[118] Stephan Hoyer and Joseph J. Hamman. 'xarray: N-D labeled Arrays and Datasets in Python'. In: *Journal of Open Research Software* 5.1 (Apr. 2017). URL: `http://openresearchsoftware.metajnl.com/articles/10.5334/jors.148/`.

[119] J. D. Hunter. 'Matplotlib: A 2D graphics environment'. In: *Computing In Science & Engineering* 9.3 (2007), pp. 90–95.

[120] Rob Hyndman, Anne B Koehler, J Keith Ord and Ralph D Snyder. *Forecasting with exponential smoothing: the state space approach*. Springer Science & Business Media, 2008.

[121] Rob J Hyndman, Roman A. Ahmed, George Athanasopoulos and Han Lin Shang. 'Optimal combination forecasts for hierarchical time series'. In: *Computational Statistics & Data Analysis* 55.9 (Sept. 2011), pp. 2579–2589. ISSN: 01679473. URL: `http://linkinghub.elsevier.com/retrieve/pii/S0167947311000971`.

[122] Rob J Hyndman and George Athanasopoulos. *Forecasting: principles and practice.* OTexts, 2018.

[123] Rob J Hyndman and Baki Billah. 'Unmasking the Theta method'. In: *International Journal of Forecasting* 19.2 (2003), pp. 287–290.

[124] Rob J Hyndman and Yeasmin Khandakar. 'Automatic time series forecasting: the forecast package for {R}'. In: *Journal of Statistical Software* 26.3 (2008), pp. 1–22. URL: `http://www.jstatsoft.org/article/view/v027i03`.

[125] Rob J Hyndman and Anne B Koehler. 'Another look at measures of forecast accuracy'. In: *International journal of forecasting* 22.4 (2006), pp. 679–688.

[126] Rob J Hyndman, Anne B Koehler, Ralph D Snyder and Simone Grose. 'A state space framework for automatic forecasting using exponential smoothing methods'. In: *International Journal of forecasting* 18.3 (2002), pp. 439–454.

[127] Rob J Hyndman et al. *forecast: Forecasting functions for time series and linear models.* Apr. 2018. URL: `https://researchportal.bath.ac.uk/en/publications/forecast-forecasting-functions-for-time-series-and-linear-models`.

[128] Rob J. Hyndman. *Encouraging replication and reproducible research.* 2010.

[129] Rob J. Hyndman. 'A brief history of forecasting competitions'. In: *International Journal of Forecasting* (2020). ISSN: 01692070.

[130] *intive-DataScience/tbats v1.1.0.* 2021. URL: `https://github.com/intive-DataScience/tbats`.

[131] Hassan Ismail Fawaz et al. 'Deep learning for time series classification: a review'. In: *Data Mining and Knowledge Discovery* 33.4 (2019), pp. 917–963. ISSN: 1573756X.

[132] Gareth James, Daniela Witten, Trevor Hastie and Robert T. Tibshirani. *An Introduction to Statistical Learning.* Vol. 102. New York: Springer, 2006, p. 618. ISBN: 9780387781884. URL: `http://books.google.com/books?id=9tv0taI8l6YC`.

[133] Kevin Jamieson and Ameet Talwalkar. 'Non-stochastic best arm identification and hyperparameter optimization'. In: *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016*. 2016.

[134] Tim Januschowski, Jan Gasthaus and Yuyang Wang. 'Open-Source Forecasting Tools in Python.' In: *Foresight: The International Journal of Applied Forecasting* 55 (2019).

[135] Tim Januschowski et al. *Criteria for classifying forecasting methods*. 2020.

[136] Rafael C. Jiménez et al. 'Four simple recommendations to encourage best practices in research software'. In: *F1000Research* (2017). ISSN: 1759796X.

[137] Eric Jones, Travis E. Oliphant, Pearu Peterson et al. *SciPy: Open source scientific tools for Python*. 2001. URL: `http://www.scipy.org/`.

[138] *Augmenting Clinical Decision-Making in Intensive Care*. The Alan Turing Institute & Great Ormond Street Hospital. 2020. URL: `https://doi.org/10.5281/zenodo.3670726`.

[139] J. D. Kalbfleisch and Ross L. Prentice. *The statistical analysis of failure time data*. J. Wiley, 2002, p. 439. ISBN: 9781118031230. URL: `https://www.wiley.com/en-us/The+Statistical+Analysis+of+Failure+Time+Data%7B%5C%%7D2C+2nd+Edition-p-9781118031230`.

[140] Viktor Kazakov and Franz J Király. 'Machine Learning Automation Toolbox (MLaut)'. In: *arXiv preprint arXiv:1901.03678* (2019).

[141] Guolin Ke et al. 'LightGBM: A highly efficient gradient boosting decision tree'. In: *Advances in Neural Information Processing Systems*. 2017.

[142] Stephen Kell. 'In search of types'. In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 2014, pp. 227–241.

[143] Eamonn Keogh, Selina Chu, David Hart and Michael Pazzani. 'Segmenting time series: A survey and novel approach'. In: *Data mining in time series databases*. World Scientific, 2004, pp. 1–21.

[144] Eamonn Keogh, Li Wei, Xiaopeng Xi, Sang-Hee Lee and Michail Vlachos. 'LB_-Keogh supports exact indexing of shapes under rotation invariance with arbitrary representations and distance measures'. In: *Proceedings of the 32nd international conference on Very large data bases*. Citeseer. 2006, pp. 882–893.

[145] Franz J. Király, Markus Löning, Anthony Blaom, Ahmed Guecioueur and Raphael Sonabend. 'Designing Machine Learning Toolboxes: Concepts, Principles and Patterns'. In: *arXiv preprint* (2021).

[146] David G. Kleinbaum and Mitchel. Klein. *Survival analysis: A Self-Learning Text*. Springer, 2012, p. 700. ISBN: 9781441966469.

[147] Alex J. Koning, Philip Hans Franses, Michèle Hibon and H. O. Stekler. 'The M3 competition: Statistical tests of the results'. In: *International Journal of Forecasting* (2005). ISSN: 01692070.

[148] Max Kuhn. 'Building Predictive Models in R: Using the caret Package'. In: *Journal of Statistical Software* 28.5 (2008). URL: `http://www.jstatsoft.org/v28/i05/`.

[149] Max Kuhn et al. *caret: Classification and Regression Training*. 2018. URL: `https://cran.r-project.org/web/packages/caret/index.html`.

[150] Siu Kwan Lam, Antoine Pitrou and Stanley Seibert. 'Numba: A LLVM-based python JIT compiler'. In: *Proceedings of the Second Workshop on the LLVM Compiler Infrastructure in HPC - LLVM '15* (2015).

[151] Michel Lang et al. 'mlr3: A modern object-oriented machine learning framework in R'. In: *Journal of Open Source Software* 4.44 (2019), p. 1903.

[152] Craig Larman. *Applying UML and patterns: an introduction to object oriented analysis and design and interactive development*. Pearson Education India, 2012.

[153] Sean Law. 'STUMPY: A Powerful and Scalable Python Library for Time Series Data Mining'. In: *Journal of Open Source Software* (2019). ISSN: 2475-9066.

[154] Thach Le Nguyen, Severin Gsponer, Iulia Ilie, Martin O'Reilly and Georgiana Ifrim. 'Interpretable time series classification using linear models and multi-resolution multi-domain symbolic representations'. In: *Data Mining and Knowledge Discovery* (2019). ISSN: 1573756X.

[155] Yann LeCun, Léon Bottou, Yoshua Bengio and Patrick Haffner. 'Gradient-based learning applied to document recognition'. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

[156] Sabina Leonelli, Daniel Spichtinger and Barbara Prainsack. *Sticks and carrots: encouraging open science at its source.* 2015.

[157] Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh and Ameet Talwalkar. 'Hyperband: A novel bandit-based approach to hyperparameter optimization'. In: *Journal of Machine Learning Research* (2018). ISSN: 15337928.

[158] Yi-Hsuan Lin, Tung-Mei Ko, Tyng-Ruey Chuang and Kwei-Jay Lin. 'Open source licenses and the creative commons framework: License selection and comparison'. In: *Journal of information science and engineering* 22.1 (2006), pp. 1–17.

[159] J Lin, E Keogh, L Wei and S Lonardi. 'Experiencing {SAX}: a novel symbolic representation of time series'. In: *Data Mining and Knowledge Discovery* 15.2 (2007).

[160] J Lin, R Khade and Y Li. 'Rotation-invariant similarity in time series using bag-of-patterns representation'. In: *Journal of Intelligent Information Systems* 39.2 (2012), pp. 287–315.

[161] Xihong Lin et al. *Past, present, and future of statistical science.* CRC Press, 2014.

[162] J Lines and Anthony Bagnall. 'Time Series Classification with Ensembles of Elastic Distance Measures'. In: *Data Mining and Knowledge Discovery* 29.3 (2015), pp. 565–592.

[163] J Lines, S Taylor and Anthony Bagnall. 'Time Series Classification with {HIVE-COTE}: The Hierarchical Vote Collective of Transformation-based Ensembles'. In: *ACM Trans. Knowledge Discovery from Data.* Vol. 12. 5. ACM, 2016.

[164] Markus List, Peter Ebert and Felipe Albrecht. *Ten Simple Rules for Developing Usable Software in Computational Biology.* 2017.

[165] Chenghao Liu, Steven C.H. Hoi, Peilin Zhao and Jianling Sun. 'Online ARIMA algorithms for time series prediction'. In: *30th AAAI Conference on Artificial Intelligence, AAAI 2016.* 2016. ISBN: 9781577357605.

[166] Alysha M. de Livera, Rob J. Hyndman and Ralph D. Snyder. 'Forecasting time series with complex seasonal patterns using exponential smoothing'. In: *Journal of the American Statistical Association* (2011). ISSN: 01621459.

[167] Markus Löning and Franz J. Király. 'Forecasting with sktime: Designing sktime's New Forecasting API and Applying It to Replicate and Extend the M4 Study'. In: *arXiv preprint* (2020).

[168] *Machine Learning for Enhanced Understanding of Cell Culture Bioprocess Development*. The Alan Turing Institute & AstraZeneca. 2019. URL: `https://doi.org/10.5281/zenodo.3367412`.

[169] Markus Löning et al. 'sktime: A Unified Interface for Machine Learning with Time Series'. In: *Workshop on Systems for ML at NeurIPS 2019* (2019).

[170] Markus Löning et al. *alan-turing-institute/sktime*. Oct. 2020. URL: `https://doi.org/10.5281/zenodo.3749000`.

[171] Markus Löning et al. *sktime: A Unified Framework for Machine Learning with Time Series*. Nov. 2020. URL: `https://doi.org/10.5281/zenodo.4061880`.

[172] Carl H Lubba, Ben Fulcher and Olivier-tl. 'chlubba/catch22: v0.2.1'. In: (Jan. 2021). URL: `https://doi.org/10.5281/zenodo.4431166%7B%5C#%7D.YDypdjVsBLc.mendeley`.

[173] Carl H Lubba et al. 'catch22: CAnonical Time-series CHaracteristics'. In: *Data Mining and Knowledge Discovery* 33.6 (2019), pp. 1821–1852.

[174] B Lucas et al. 'Proximity Forest: an effective and scalable distance-based classifier for time series'. In: *Data Mining and Knowledge Discovery* 33.3 (2019), pp. 607–635.

[175] Helmut Lütkepohl. *New introduction to multiple time series analysis*. Springer Science & Business Media, 2005.

[176] M4 Team et al. 'M4 competitor's guide: Prizes and rules'. In: (2018). URL: `https://www.m4.unic.ac.cy/wp-content/uploads/2018/03/M4-Competitors-Guide.pdf`.

[177] M4 Team et al. 'M4 dataset'. In: (2018). URL: `https://github.com/Mcompetitions/M4-methods/tree/master/Dataset`.

[178] Spyros Makridakis, Vassilios Assimakopoulos and Evangelos Spiliotis. *Objectivity, reproducibility and replicability in forecasting research.* 2018.

[179] Spyros Makridakis and Michèle Hibon. 'The M3-competition: Results, conclusions and implications'. In: *International Journal of Forecasting* 16.4 (2000), pp. 451–476. ISSN: 01692070.

[180] Spyros Makridakis, Evangelos Spiliotis and Vassilios Assimakopoulos. 'Statistical and Machine Learning forecasting methods: Concerns and ways forward'. In: *PLoS one* 13.3 (2018). ISSN: 19326203.

[181] Spyros Makridakis, Evangelos Spiliotis and Vassilios Assimakopoulos. 'The M4 Competition: Results, findings, conclusion and way forward'. In: *International Journal of Forecasting* 34.4 (2018), pp. 802–808. ISSN: 01692070.

[182] Spyros Makridakis, Evangelos Spiliotis and Vassilios Assimakopoulos. 'The M4 Competition: 100,000 time series and 61 forecasting methods'. In: *International Journal of Forecasting* (2019).

[183] Spyros Makridakis et al. 'The accuracy of extrapolation (time series) methods: Results of a forecasting competition'. In: *Journal of forecasting* 1.2 (1982), pp. 111–153.

[184] Wes McKinney. 'Data Structures for Statistical Computing in Python'. In: *Proceedings of the 9th Python in Science Conference.* Ed. by Stéfan van der Walt and Jarrod Millman. 2010, pp. 51–56.

[185] Wes McKinney. 'pandas: a Foundational Python Library for Data Analysis and Statistics'. In: *Python for High Performance and Scientific Computing.* 2011.

[186] Wannes Meert, Kilian Hendrickx and Toon Van Craenendonck. *wannesm/dtaidistance v2.0.0.* Aug. 2020. URL: `https://doi.org/10.5281/zenodo.3981067`.

[187] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman and Aram Galstyan. 'A survey on bias and fairness in machine learning'. In: *arXiv preprint arXiv:1908.09635* (2019).

[188] Bertrand Meyer. *Object-oriented software construction.* Vol. 2. Prentice hall Englewood Cliffs, 1997.

[189] Matthew Middlehurst, James Large and Anthony Bagnall. 'The canonical interval forest (CIF) classifier for time series classification'. In: *arXiv preprint arXiv:2008.09172* (2020).

[190] K. Jarrod Millman and Michael Aivazis. 'Python for Scientists and Engineers'. In: *Computing in Science & Engineering* 13.2 (Mar. 2011), pp. 9–12. ISSN: 1521-9615. URL: `http://ieeexplore.ieee.org/document/5725235/`.

[191] Margaret Mitchell et al. 'Model cards for model reporting'. In: *Proceedings of the conference on fairness, accountability, and transparency*. 2019, pp. 220–229.

[192] Pablo Montero-Manso, George Athanasopoulos, Rob J Hyndman and Thiyanga S Talagala. 'FFORMA: Feature-based forecast model averaging'. In: *International Journal of Forecasting* 36.1 (2020), pp. 86–92.

[193] Douglas C Montgomery. *Design and analysis of experiments*. Wiley, 2017.

[194] Abdullah Mueen. 'Time series motif discovery: dimensions and applications'. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 4.2 (2014), pp. 152–159.

[195] Andreas C Müller. *amueller/dabl*. 2021. URL: `https://github.com/amueller/dabl`.

[196] Soroosh Nalchigar et al. 'Solution patterns for machine learning'. In: *International Conference on Advanced Information Systems Engineering*. Springer. 2019, pp. 627–642.

[197] Elizamary Nascimento, Anh Nguyen-Duc, Ingrid Sundbø and Tayana Conte. 'Software engineering for artificial intelligence and machine learning software: A systematic literature review'. In: *arXiv preprint arXiv:2011.03751* (2020).

[198] Brett Naul, Stéfan van der Walt, Arien Crellin-Quick, Joshua S Bloom and Fernando Pérez. 'Cesium: open-source platform for time-series inference'. In: *arXiv preprint arXiv:1609.04504* (2016).

[199] P B Nemenyi. 'Distribution-free multiple comparisons'. In: 25.2 (1963), p. 1233.

[200] Travis E. Oliphant. 'Python for Scientific Computing'. In: *Computing in Science & Engineering* 9.3 (May 2007), pp. 10–20. ISSN: 1521-9615. URL: `http://ieeexplore.ieee.org/document/4160250/`.

[201] Travis E. Oliphant. *Guide to NumPy*. 2nd. CreateSpace Independent Publishing Platform, 2015, p. 364. ISBN: 151730007X.

[202] David Lorge Parnas, John E Shore and David Weiss. 'Abstract types defined as classes of variables'. In: *ACM SIGPLAN Notices* 11.SI (1976), pp. 149–154.

[203] Adam Paszke et al. 'Pytorch: An imperative style, high-performance deep learning library'. In: *Advances in neural information processing systems*. 2019, pp. 8026–8037.

[204] Judea Pearl, Madelyn Glymour and Nicholas P Jewell. *Causal inference in statistics: A primer*. John Wiley & Sons, 2016.

[205] Fabian Pedregosa et al. 'Scikit-learn: Machine Learning in Python'. In: *The Journal of Machine Learning Research* 12 (2011), pp. 2825–2830. URL: `https://dl.acm.org/citation.cfm?id=2078195`.

[206] Fernando Perez and Brian E. Granger. 'IPython: A System for Interactive Scientific Computing'. In: *Computing in Science & Engineering* 9.3 (2007), pp. 21–29. ISSN: 1521-9615. URL: `http://ieeexplore.ieee.org/document/4160251/`.

[207] Josef Perktold and Skipper Seabold. 'Statsmodels: Econometric and Statistical Modeling with Python Quantitative histology of aorta View project Statsmodels: Econometric and Statistical Modeling with Python'. In: *Proceedings of the 9th Python in Science Conference*. 2010. URL: `https://www.researchgate.net/publication/264891066`.

[208] M Hashem Pesaran. *Time series and panel data econometrics*. Oxford University Press, 2015.

[209] Fotios Petropoulos et al. 'Forecasting: theory and practice'. In: (Dec. 2020). URL: `https://arxiv.org/abs/2012.03854`.

[210] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT Press, 2002.

[211] Jim Pivarski, Peter Elmer and David Lange. 'Awkward Arrays in Python, C++, and Numba'. In: *arXiv preprint arXiv:2001.06307* (2020).

[212] Jim Pivarski, Ianna Osborne, Pratyush Das, Anish Biswas and Peter Elmer. 'Awkward Array: JSON-like data, NumPy-like idioms'. In: *Scipy Confernce 2020* (2020).

[213] Hans E. Plesser. 'Reproducibility vs. Replicability: A brief history of a confused terminology'. In: *Frontiers in Neuroinformatics* (2018). ISSN: 16625196.

[214] Roger S Pressman. *Software engineering: a practitioner's approach.* Palgrave macmillan, 2005.

[215] R Core Team. *R: A Language and Environment for Statistical Computing.* Tech. rep. Vienna, Austria: R Foundation for Statistical Computing, 2014.

[216] Sebastian Raschka. 'MLxtend: Providing machine learning and data science utilities and extensions to Python's scientific computing stack'. In: *The Journal of Open Source Software* 3.24 (Apr. 2018). URL: `http://joss.theoj.org/papers/10.21105/joss.00638`.

[217] Sebastian Raschka, Joshua Patterson and Corey Nolet. 'Machine learning in Python: Main developments and technology trends in data science, machine learning, and artificial intelligence'. In: *Information* (2020). ISSN: 20782489.

[218] *robjhyndman/tsfeatures.* URL: `https://github.com/robjhyndman/tsfeatures`.

[219] Matthew Rocklin. 'Dask: Parallel computation with blocked algorithms and task scheduling'. In: *Proceedings of the 14th Python in Science Conference.* 2015, pp. 130–136. URL: `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.825.5314%7B%5C&%7Drep=rep1%7B%5C&%7Dtype=pdf`.

[220] Winston W Royce. 'Managing the development of large software systems: concepts and techniques'. In: *Proceedings of the 9th international conference on Software Engineering.* 1987, pp. 328–338.

[221] David E. Rumelhart, Geoffrey E. Hinton and Ronald J. Williams. 'Learning representations by back-propagating errors'. In: *Nature* 323.6088 (Oct. 1986), pp. 533–536. ISSN: 0028-0836. URL: `http://www.nature.com/doifinder/10.1038/323533a0`.

[222] Stuart J Russell and Peter Norvig. *Artificial Intelligence-A Modern Approach.* 4th. London: Pearson Education, 2021.

[223] P Schäfer. 'The {BOSS} is concerned with time series classification in the presence of noise'. In: *Data Mining and Knowledge Discovery* 29.6 (2015), pp. 1505–1530.

[224] P Schäfer and M Högqvist. 'SFA: a symbolic {Fourier} approximation and index for similarity search in high dimensional datasets'. In: *Proceedings of the 15th International Conference on Extending Database Technology.* 2012, pp. 516–527.

[225] M. Schwab, N. Karrenbach and J. Claerbout. 'Making scientific computations reproducible'. In: *Computing in Science & Engineering* (2000). ISSN: 15219615.

[226] David Sculley et al. 'Hidden technical debt in machine learning systems'. In: *Advances in neural information processing systems.* 2015, pp. 2503–2511.

[227] Artemios-Anargyros Semenoglou, Evangelos Spiliotis, Spyros Makridakis and Vassilios Assimakopoulos. 'Investigating the accuracy of cross-learning time series forecasting methods'. In: *International Journal of Forecasting* (2020). ISSN: 0169-2070. URL: `https://www.sciencedirect.com/science/article/pii/S0169207020301850`.

[228] Shai Shalev-Shwartz and Shai Ben-David. *Understanding machine learning : from theory to algorithms.* Cambridge University Press, 2014, p. 397. ISBN: 1107057132. URL: `https://books.google.co.uk/books?hl=en%7B%5C&%7Dlr=%7B%5C&%7Did=Hf6QAwAAQBAJ%7B%5C&%7Doi=fnd%7B%5C&%7Dpg=PR15%7B%5C&%7Ddq=Understanding+Machine+Learning:+From+Theory+to+Algorithms%7B%5C&%7Dots=2HtfQloLL0%7B%5C&%7Dsig=s%7B%5C_%7DmD-JFAuUbYLYWo2gK5yAlcbcg%7B%5C#%7Dv=onepage%7B%5C&%7Dq=Understanding%20Machine%20Learning%7B%5C%%7D3A%20From%20Theory%20to`.

[229] Galit Shmueli. 'To explain or to predict?' In: *Statistical Science* (2010). ISSN: 08834237.

[230] Julien Siebert, Janek Groß and Christof Schroth. 'A systematic review of Python packages for time series analysis'. In: *arXiv preprint arXiv:2104.07406* (2021).

[231] *skpref/skpref.* 2021. URL: `https://github.com/skpref/skpref`.

[232] *sktime/sktime-dl.* 2021. URL: `https://github.com/sktime/sktime-dl`.

[233] Taylor G. Smith et al. *pmdarima: ARIMA estimators for Python.* 2019. URL: `http://www.alkaline-ml.com/pmdarima`.

[234] Slawek Smyl. 'A hybrid method of exponential smoothing and recurrent neural networks for time series forecasting'. In: *International Journal of Forecasting* 36.1 (2020), pp. 75–85.

[235] Slawek Smyl, Jai Ranganathan and Andrea Pasqua. 'M4 forecasting competition: Introducing a new hybrid ES-RNN model'. In: *URL: https://eng. uber. com/m4-forecasting-competition* (2018).

[236] Derek Snow. *AtsPy: Automated Time Series Models in Python*. 2020. URL: `https://github.com/firmai/atspy/`.

[237] Raphael Sonabend, Franz J Király, Andreas Bender, Bernd Bischl and Michel Lang. 'mlr3proba: Machine Learning Survival Analysis in R'. In: *arXiv preprint arXiv:2008.08080* (2020).

[238] Sören Sonnenburg et al. 'The need for open source software in machine learning'. In: *Journal of Machine Learning Research* 8.Oct (2007), pp. 2443–2466.

[239] William D Stanley, Gary R Dougherty, Ray Dougherty and H Saunders. 'Digital signal processing'. In: (1988).

[240] Souhaib Ben Taieb. 'Machine learning strategies for multi-step-ahead time series forecasting'. PhD thesis. Universit Libre de Bruxelles, Belgium, 2014.

[241] Souhaib Ben Taieb and Rob J. Hyndman. 'Boosting multi-step autoregressive forecasts'. In: *31st International Conference on Machine Learning, ICML 2014*. 2014. ISBN: 9781634393973.

[242] Yoon-Sik Tak and Eenjun Hwang. 'A leaf image retrieval scheme based on partial dynamic time warping and two-level filtering'. In: *7th IEEE International Conference on Computer and Information Technology (CIT 2007)*. IEEE. 2007, pp. 633–638.

[243] Chang Wei Tan, Christoph Bergmeir, Francois Petitjean and Geoffrey I. Webb. 'Time Series Extrinsic Regression'. In: (June 2020). URL: `http://arxiv.org/abs/2006.12672`.

[244] Romain Tavenard. *tslearn: A machine learning toolkit dedicated to time-series data*. 2017. URL: `https://tslearn.readthedocs.io/en/latest/index.html`.

[245] Romain Tavenard et al. 'Tslearn, A Machine Learning Toolkit for Time Series Data'. In: *Journal of Machine Learning Research* 21.118 (2020), pp. 1–6. URL: http://jmlr.org/papers/v21/20-091.html.

[246] Ross Taylor. *PyFlux: An open source time series library for Python.* 2016. URL: http://www.pyflux.com.

[247] Sean J Taylor and Benjamin Letham. 'Forecasting at scale'. In: *The American Statistician* 72.1 (Sept. 2018), pp. 37–45. URL: https://peerj.com/preprints/3190/.

[248] Sean J. Taylor and Benjamin Letham. 'Forecasting at Scale'. In: *American Statistician* (2018). ISSN: 15372731.

[249] *tidyverts/fable v0.2.1.* URL: https://github.com/tidyverts/fable.

[250] Charles Truong, Laurent Oudre and Nicolas Vayatis. 'Selective review of offline change point detection methods'. In: *Signal Processing* 167 (2020), p. 107299.

[251] *unit8co/darts.* 2021. URL: https://github.com/unit8co/darts.

[252] Joaquin Vanschoren, Jan N Van Rijn, Bernd Bischl and Luis Torgo. 'OpenML: networked science in machine learning'. In: *ACM SIGKDD Explorations Newsletter* 15.2 (2014), pp. 49–60.

[253] Vladimir Naumovich Vapnik. 'Principles of Risk Minimization for Learning Theory'. In: *Advances in Neural Information Processing Systems 4*. Ed. by R. P. Lippmann, J. E. Moody and S.J. Hanson. Morgan Kaufmann, 1992, pp. 831–838. URL: http://papers.nips.cc/paper/506-principles-of-risk-minimization-for-learning-theory.pdf.

[254] Vladimir Naumovich Vapnik. *The Nature of Statistical Learning Theory.* Springer, 1995. ISBN: 1475732643. URL: https://books.google.co.uk/books?hl=en%7B%5C&%7Dlr=%7B%5C&%7Did=EqgACAAAQBAJ%7B%5C&%7Doi=fnd%7B%5C&%7Dpg=PR7%7B%5C&%7Ddq=%7B%5C%%7D09The+Nature+of+Statistical+Learning+Theory%7B%5C&%7Dots=g3HZjxcX19%7B%5C&%7Dsig=k9dqrqWjGNxE%7B%5C_%7D451rb35aKUCGNA%7B%5C#%7Dv=onepage%7B%5C&%7Dq=The%20Nature%20of%20Statistical%20Learning%20Theory%7B%5C&%7Df=false.

[255] Vladimir Naumovich Vapnik. *Statistical Learning Theory.* Wiley, 1998.

[256] Gaël Varoquaux et al. 'Scikit-learn: Machine Learning Without Learning the Machinery'. In: *GetMobile: Mobile Computing and Communications* 19.1 (June 2015), pp. 29–33. URL: `http://dl.acm.org/citation.cfm?doid=2786984.2786995`.

[257] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013.

[258] Stéfan van der Walt, S Chris Colbert and Gaël Varoquaux. 'The NumPy Array: A Structure for Efficient Numerical Computation'. In: *Computing in Science & Engineering* 13.2 (2011), pp. 22–30. ISSN: 1521-9615. URL: `http://ieeexplore.ieee.org/document/5725236/`.

[259] Hironori Washizaki, Hiromu Uchida, Foutse Khomh and Yann-Gaël Guéhéneuc. 'Studying software engineering patterns for designing machine learning systems'. In: *2019 10th International Workshop on Empirical Software Engineering in Practice (IWESEP)*. IEEE. 2019, pp. 49–495.

[260] F Wilcoxon. 'Individual comparisons by ranking methods'. In: *Biometrics* 1 (1945), pp. 80–83.

[261] Greg Wilson et al. 'Best Practices for Scientific Computing'. In: *PLoS Biology* (2014). ISSN: 15449173.

[262] Greg Wilson et al. 'Good enough practices in scientific computing'. In: *PLoS Computational Biology* (2017). ISSN: 15537358.

[263] Peter R Winters. 'Forecasting sales by exponentially weighted moving averages'. In: *Management science* 6.3 (1960), pp. 324–342.

[264] Rebecca Wirfs-Brock and Alan McKean. *Object design: roles, responsibilities, and collaborations*. Addison-Wesley Professional, 2003.

[265] Rebecca Wirfs-Brock and Brian Wilkerson. 'Object-oriented design: a responsibility-driven approach'. In: *ACM SIGPLAN Notices* 24.10 (1989), pp. 71–75.

[266] Rebecca Wirfs-Brock, Brian Wilkerson and Lauren Wiener. 'Designing object-oriented software'. In: (1990).

[267] Jeffrey M Wooldridge. *Econometric analysis of cross section and panel data*. MIT Press, 2010, p. 1064. ISBN: 0262232588. URL: `https://books.google.co.uk/books?id=yov6AQAAQBAJ%7B%5C&%7Dprintsec=frontcover%7B%5C&%7Ddq=wooldridge+panel+data%7B%5C&%7Dhl=en%7B%5C&%7Dsa=X%7B%5C&%7Dved=0ahUKEwie5JH8u7PeAhVFTcAKHR3qA70Q6AEIKzAA%7B%5C#%7Dv=onepage%7B%5C&%7Dq=wooldridge%20panel%20data%7B%5C&%7Df=false`.

[268] Zhengzheng Xing, Jian Pei and S Yu Philip. 'Early classification on time series'. In: *Knowledge and information systems* 31.1 (2012), pp. 105–127.

[269] M Yeh et al. 'Matrix Profile I: All Pairs Similarity Joins for Time Series: A Unifying View that Includes Motifs, Discords and Shapelets'. In: *Proc. 16th {IEEE} International Conference on Data Mining*. 2016.

[270] G. Peter Zhang. 'Neural networks for time-series forecasting'. In: *Handbook of Natural Computing*. Springer, 2012. ISBN: 9783540929109.

[271] Guoqiang Zhang, B. Eddy Patuwo and Michael Y. Hu. 'Forecasting with artificial neural networks: The state of the art'. In: *International Journal of Forecasting* (1998). ISSN: 01692070.

[272] A Zheng. 'The challenges of building machine learning tools for the masses'. In: *SE4ML:SoftwareEngineering for Machine Learning (NeurIPS 2014 Workshop)*. 2014.