Course Book

MACHINE LEARNING—SUPERVISED LEARNING

DLBDSMLSL01

MACHINE LEARNING—SUPERVISED LEARNING

MASTHEAD

Publisher:
IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing address:
Albert-Proeller-Straße 15-19
D-86675 Buchdorf
media@iu.org
www.iu.de

DLBDSMLSL01
Version No.: 001-2023-0810
N.N.

© 2023 IU Internationale Hochschule GmbH
This course book is protected by copyright. All rights reserved.
This course book may not be reproduced and/or electronically edited, duplicated, or distributed in any kind of form without written permission by the IU Internationale Hochschule GmbH (hereinafter referred to as IU).
The authors/publishers have identified the authors and sources of all graphics to the best of their abilities. However, if any erroneous information has been provided, please notify us accordingly.

MODULE DIRECTOR

PROF. DR. CHRISTIAN MÜLLER-KETT

Mr. Müller-Kett has been a professor of data science at IU International University of Applied Sciences since 2020. He studied biology and integrated geography with a focus on system modeling and geodata processing at Radboud University Nijmegen, Albert-Ludwigs-University Freiburg, and Humboldt University Berlin. He received his doctorate from Ruhr Univeristy Bochum upon completion of his thesis concerning system-oriented spatial simulations for sustainable urban planning in the field of urban analytics and geoinformatics.

In his professional work, he acts in a consultative capacity, focusing on the management and analysis of spatial data, predictive analysis, the Internet of Things, big data, and distributed cloud computing.

His research primarily concerns machine learning techniques in sustainable spatial planning, limnology, and remote sensing. He has completed such research during his stays abroad in the United States, Argentina, and Japan.

TABLE OF CONTENTS

MACHINE LEARNING—SUPERVISED LEARNING

Module Director . . . 3

Introduction
Signposts Throughout the Course Book . . . 8
Suggested Readings . . . 9
Learning Objectives . . . 10

Unit 1
Introduction to Machine Learning 11

1.1 Pattern Recognition Systems . . . 12
1.2 The Machine Learning Design Cycle . . . 16
1.3 Technical Notions of Learning and Adaption . . . 18
1.4 Under- and Overfitting . . . 23

Unit 2
Regression 27

2.1 Linear Regression . . . 28
2.2 Lasso and Ridge Regularization . . . 35
2.3 Generalized Linear Models . . . 39
2.4 Logistic Regression . . . 43

Unit 3
Basic Classification Techniques 51

3.1 K-Nearest Neighbor . . . 53
3.2 Naïve Bayes . . . 62

Unit 4
Support Vector Machines 67

4.1 Large Margin Classification . . . 69
4.2 The Kernel Trick . . . 73

Unit 5
Decision & Regression Trees 79

5.1 Decision & Regression Trees . . . 80
5.2 Random Forest . . . 86
5.3 Gradient Boosting . . . 90

Appendix

List of References 94
List of Tables and Figures 96

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

This course book contains the core content for this course. Additional learning materials can be found on the learning platform, but this course book should form the basis for your learning.

The content of this course book is divided into units, which are divided further into sections. Each section contains only one new key concept to allow you to quickly and efficiently add new learning material to your existing knowledge.

At the end of each section of the digital course book, you will find self-check questions. These questions are designed to help you check whether you have understood the concepts in each section.

For all modules with a final exam, you must complete the knowledge tests on the learning platform. You will pass the knowledge test for each unit when you answer at least 80% of the questions correctly.

When you have passed the knowledge tests for all the units, the course is considered finished and you will be able to register for the final assessment. Please ensure that you complete the evaluation prior to registering for the assessment.

Good luck!

SUGGESTED READINGS

GENERAL SUGGESTIONS

Bishop, C. M. (2006). *Pattern recognition and machine learning.* Springer.

Grus, J. (2019). *Data science from scratch: First principles with Python* (2nd ed.). O'Reilly.

Mitchell, T. M. (1997). *Machine learning*. McGraw-Hill.

UNIT 1

Hingane, M. C., Matkar, S. B., Mane, A. B., & Shirsat, A. M. (2015). Classification of MRI brain image using SVM classifier. *International Journal of Science Technology & Engineering*, *1*(9), 24—28.

Wirth, R., & Hipp, J. (2000). CRISP-DM: Towards a standard process model for data mining. *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining* (pp. 29—39). Practical Application Company.

UNIT 2

Uyanıka, G. K., Güler, N. (2013). A study on multiple linear regression analysis. *Procedia—Social and Behavioral Sciences, 106*(2013), 234—240.

UNIT 3

Cunningham, P., & Delany, S. J. (2020). *K-nearest neighbor classifiers (2nd edition, with Python examples)* [Manuscript submitted for publication]. arXiv.

UNIT 4

Gunn, S. (1998). *Support vector machines for classification and regression.* University of Southampton. Available online.

UNIT 5

Loh, W. (2014). Fifty years of classification and regression trees. *International Statistical Review, 82*(3), 329—348.

LEARNING OBJECTIVES

The course **Machine Learning—Supervised Learning** focuses on one of the most popular topics in the world of computer science. In fact, machine learning can be credited with having influenced some of the most impactful technological achievements made in recent years, from recommendation algorithms used across internet platforms to the creation of self-driving vehicles. Machine learning plays an increasingly important role in our everyday lives. One of the most popular ways to teach a machine is to use supervised learning techniques. Here, a machine is taught using data that include both the inputs and outputs the machine should eventually learn to predict.

In this course, you will discover how machine learning relates to pattern recognition. You will then learn about the various design and implementation stages of a machine learning model and how to recognize and avoid any potential pitfalls. The many algorithms used in supervised learning, including support vector machines and decision trees, will be presented to you in an easy, straightforward, and practical way, gradually increasing in complexity to avoid confusion. You will be introduced to new concepts, such as regression and classification, and familiarize yourself with their techniques. By the end of the course, you will be able to understand and apply these algorithms and methods on your own.

UNIT 1

INTRODUCTION TO MACHINE LEARNING

STUDY GOALS

On completion of this unit, you will be able to …

- define machine learning and fully understand the concept of supervised learning.
- utilize the machine learning design cycle as an iterative process model for building machine learning systems.
- understand the technical notions of learning and adaption and how they relate to one other.
- detect and avoid problems of over- and underfitting.

1. INTRODUCTION TO MACHINE LEARNING

Introduction

With data being recorded in so many aspects of our lives, the amount collected in recent years has grown considerably. There is a wealth of valuable information hidden within this ever-increasing flood of data—data that come in many different formats, including numerical, textual, and even audiovisual. The complexity of these data, as well as the sheer amount amassed, makes identifying and understanding insightful patterns and relationships hidden therein a laborious task. We need algorithms that allow us to analyze these huge amounts of data, observe patterns, and propose rules so that hypotheses can be formulated and tested.

In this unit, we will find answers to the following questions:

- What is machine learning?
- How is pattern recognition related to machine learning?
- How can we design and implement a machine learning model?
- What should we be aware of during the machine learning process?

1.1 Pattern Recognition Systems

Pattern recognition and machine learning are two concepts that are very closely linked with each other. Bishop (2006) explains that pattern recognition has its roots in engineering and that machine learning has evolved from the field of computer science. Nevertheless, both of these activities can be viewed as facets of the same field. Although the focus in machine learning is on more computationally intensive methods, there is still a strong overlap between the two activities: both pattern recognition and machine learning aim to detect patterns hidden in data (Webb, 2002).

In this course, we understand pattern recognition as a process in which machine learning techniques are used to detect patterns hidden in data. More formally, pattern recognition is the process of using machine learning techniques to assign a **label** to a given **observation** based on its features (i.e., the observation's attributes). In this context, "to learn" is to capture the dependency structures between features and labels in order to generalize them and thereby predict labels for new, unseen observations. The following are two forms of pattern recognition that can be distinguished based on the type of machine learning used:

Label
A label is the attribute that we want to predict.

Observation
A single record in the data is called an observation.

1. With supervised learning, we provide both the features and the labels that we want to predict during the learning phase. The model therefore learns to assign labels based on provided data containing both of these elements.
2. With unsupervised learning, the training dataset contains only the features (and not the labels to be predicted). The goal here is to detect patterns and similarities in the data in order to assign the same label to similar observations.

In this course, we will focus on supervised machine learning, i.e., we will work with training data containing both features and labels. Supervised learning models can be further grouped into classification and regression models: classification models predict class membership (e.g., whether a client is a churner or a non-churner); regression models predict continuous numeric values (e.g., sales figures).

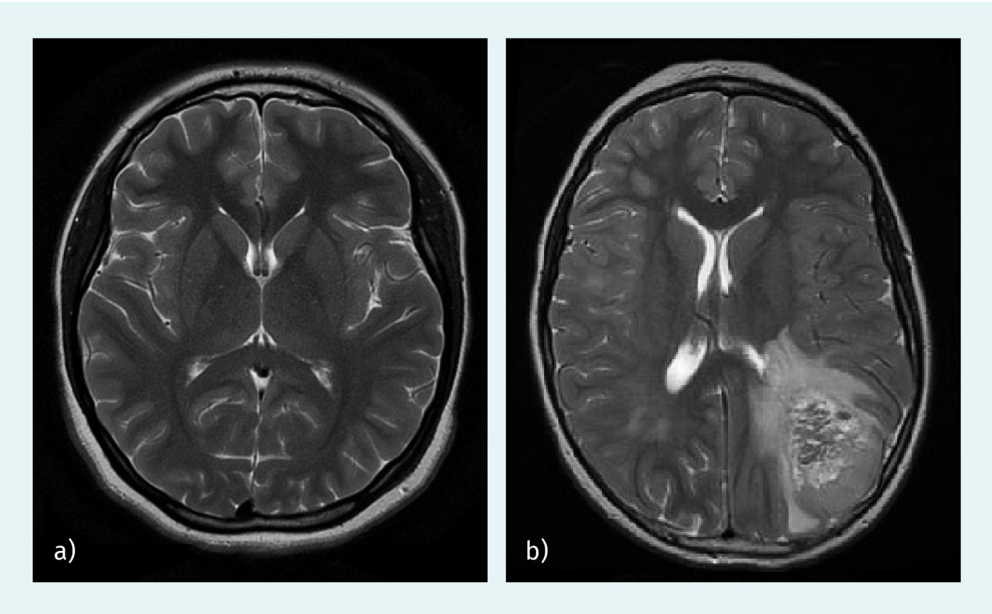
Pattern Recognition in the Real World

Let us now look at a real-world example of a pattern recognition system—more precisely, a classification model based on the work of Singh and Kaur (2012), Funmilola et al. (2015), and Hingane et al. (2015). The processing of image data plays a pivotal role in the medical field. Through the use of x-ray machines, magnetic resonance imaging (MRI), and ultrasound machines, a large number of invaluable images and data can be generated to aid in the diagnosis of inconspicuous diseases.

Suppose that a doctor is trying to detect abnormalities (including evidence of disease) present in an MRI scan of a patient's brain. Given the gravity of the decision this doctor needs to make, they may opt to consult an automated system to help identify and/or confirm the diagnosis. Therefore, in this example, we would like to automate the brain diagnosis process and simplify the associated task of determining the brain's current state of health (e.g., by using MRI images to detect any potential tumors).

For this process, we will need two sets of images, which will act as the input for the learning process of the pattern classification system. The first set contains images of healthy brains, and the second set contains images of brains with abnormalities. The following figure shows two example images: on the left, we see a healthy brain; on the right, we see that the MRI has revealed an abnormality.

Figure 1: Sample of Normal and Abnormal Brain Images

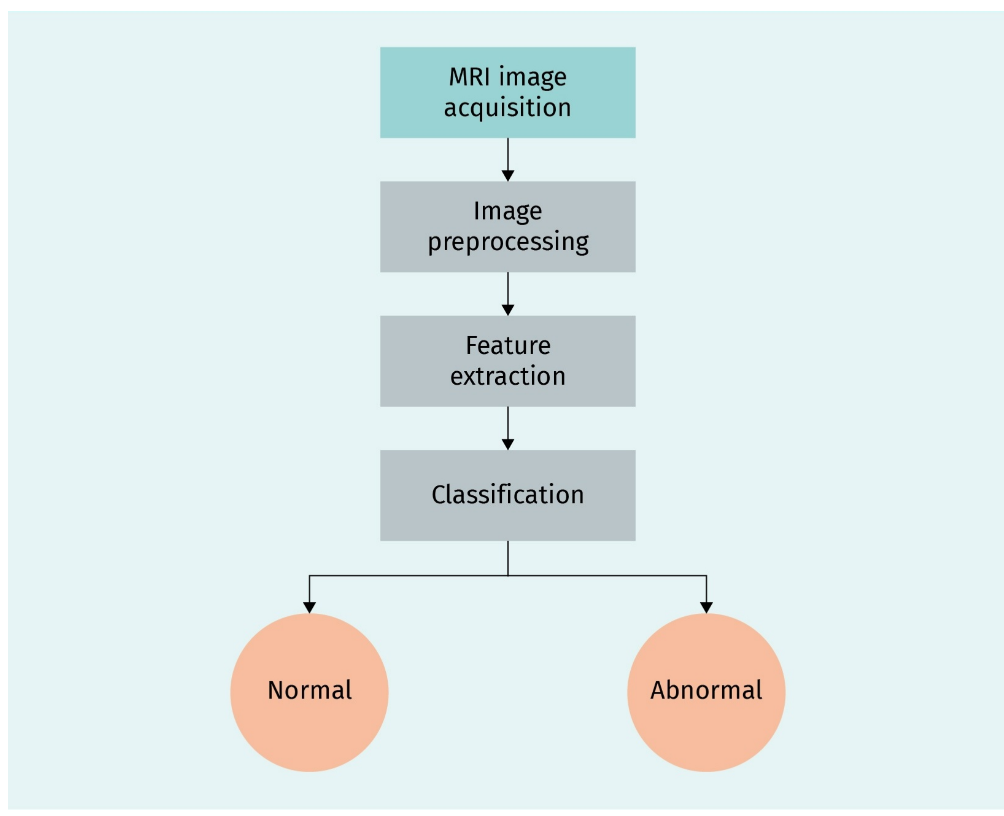


Source: Hashan (2021a; 2021b), CC BY-NC-SA 4.0.

Based on the training images, the system must learn to fully comprehend how the human brain is structured and what the differences between healthy brains and brains with abnormalities look like. We can then successfully make an automated diagnosis. Because every human brain is unique (and therefore no two images will be perfectly alike), this is a very challenging task. The complex composition of the brain, which is divided into white matter, gray matter, and cerebrospinal fluid, is another obstacle that must be overcome. Furthermore, the images are often affected by various confounding factors relating to the scanning process itself (e.g., the patient accidentally moving during capture).

The pattern classification system must manage this complexity by using the information that can be taken from the image (e.g., image density, shape structure, and color differences). These properties help to generate features used by the classifier, whose goal is to label the image (i.e., classify it) as “normal” or “abnormal.” It does this by predicting the probabilities of class membership for these two classes and then assigning the appropriate class label (i.e., the one with the highest rate of probability predicted) to the image. The pattern classification system is a process designed to break down the complexity of the data provided. It comprises a set of successive steps used for each image. These individual steps are shown in the following figure.

Figure 2: Brain MRI Image Classification



Source: Hofer (2021).

Before any legitimate analysis and classification of these images can take place, we must first go through the **preprocessing** stage, i.e., the images must be prepared for machine learning. In this first step, each MRI brain scan is converted into an array of pixels and then sent through a sequence of simple transformation steps. This may include, in this scenario, resizing each image so they have the same dimensions, filtering them to remove noise, using a thresholding algorithm to identify and highlight any areas of interest, and/or converting them to grayscale to reduce both complexity and the model's eventual computational costs. In some cases, it is also advisable to use augmentation techniques to artificially expand the training dataset, e.g., by intentionally skewing the image, shifting it, or adding extra noise. As a final preprocessing step, **image segmentation** is performed to separate the different parts of the brain. Feature extraction comes next in the overall process, where the preprocessed image is reduced to certain essential properties. By selecting and combining these properties, features can be derived that are suitable for distinguishing the classes from each other.

Preprocessing
Preparing data to make them usable for machine learning is called preprocessing.

Image segmentation
Grouping similar pixels to identify objects in an image is called image segmentation.

For feature extraction, the next stage, the models designed by Singh and Kaur (2012) and Hingane et al. (2015) use a statistical method that characterizes an image based on the position of pixels with similar gray values. It uses this information to transform the image into a set of features (e.g., texture or shape). After completing this step, each image is now reduced to a feature vector x in an n-dimensional feature space, where n is the number of features.

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

The last step in this process concerns the actual classification. This step is divided into two phases: the training phase and the testing phase. In the training phase, a large portion of the images are used to learn the decision boundary, which separates and isolates the two classes in the feature space. The decision boundary is then tested using the remaining images that were left unused during the training phase. This testing phase reveals the suitability of the decision boundary.

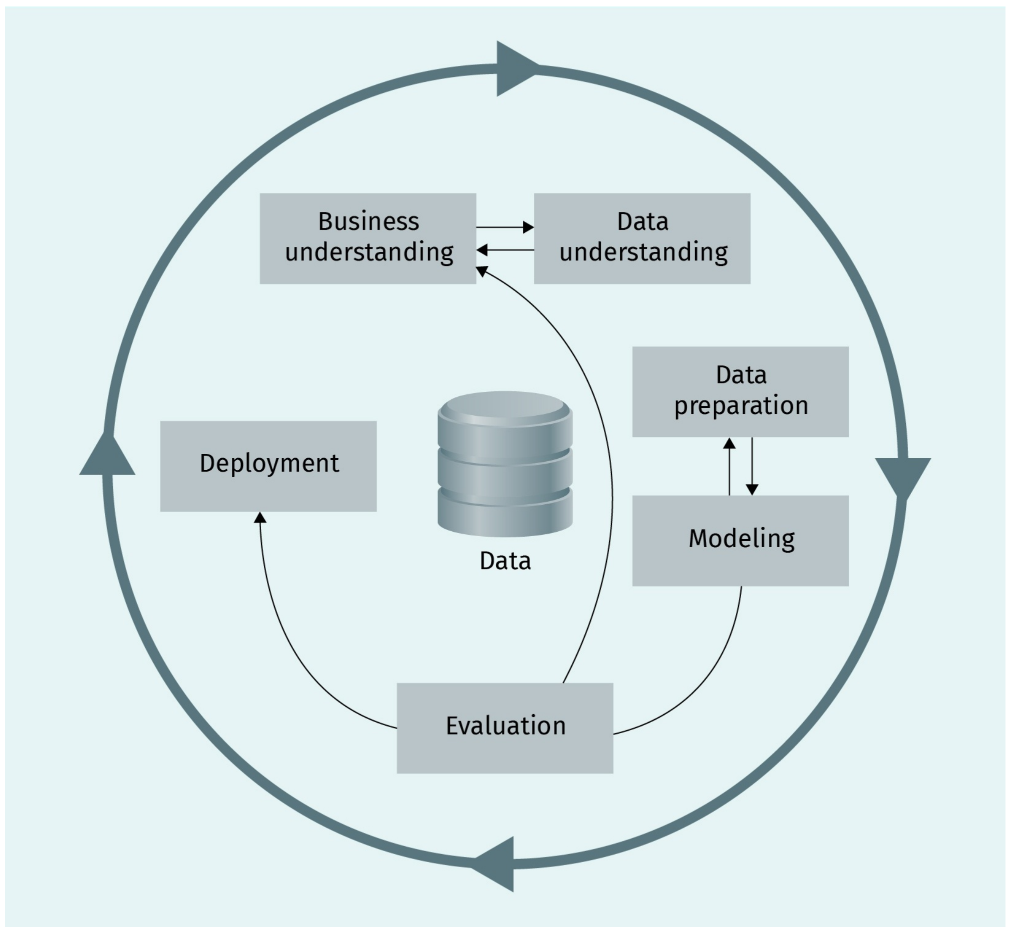
In order for the classifier to perform effectively, the selection of features representing the image must be taken into consideration. It is important to choose features that provide the optimal decision boundary. The more features taken into account, the more complex the classification model will be, which may not necessarily be beneficial to us. Some features could be redundant or entirely useless, others may be expensive to measure or extract, and some may result in less accurate classifications. But how do we know which features to choose? One possible approach is to obtain and utilize more comprehensive and diverse training data. Conversely, one could simplify the model and generalize its parameters while keeping the overall **model loss** in mind. As a rule, we want to keep our models as simple as possible.

Model loss
This term refers to the calculation of the probability of misclassification.

1.2 The Machine Learning Design Cycle

Although the development of a machine learning system can be structurally divided into specific, successive phases, the very nature of machine learning is typically characterized by iterative, recurrent phases. By going through these phases, the model can continuously improve. This is also referred to as the machine learning design cycle. The Cross-Industry Standard Process for Data Mining (CRISP-DM) is regarded as the blueprint for the development of machine learning models. The CRISP-DM cycle consists of six phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment. The individual phases of the CRISP-DM cycle and their iterative nature are illustrated in the following figure based on Wirth and Hipp (2000).

Figure 3: The Six Phases of the CRISP-DM Process Model



Source: Jensen (2012), CC BY-SA 3.0.

The Six Phases of the CRISP-DM Process Cycle

We will now take a closer look at each stage in this cycle to better understand the roles that they play in model construction.

Business understanding

This initial phase concerns the understanding of project objectives and requirements from a business perspective and their translation into a machine learning problem. A project plan is also designed at this stage in the process. The term “business” here should be understood as a broad concept, referring to organizations in a more general manner.

Data understanding

This phase begins with an initial collection of data and the appropriate actions taken to understand them. The goal is to examine the data quality, detect any possible errors and inconsistencies, identify interesting subsets, and form early insights. Because it is necessary to understand the data on, at least, a basic level when project planning, it is not uncommon for iterations between these first two phrases to occur.

Data preparation

After forming an initial picture of the existing data and their quality, the data are prepared for machine learning. This includes the selection and integration of the data chosen as machine learning input, feature extraction, data cleaning, and the transformation of the data into a suitable format.

Modeling

The actual machine learning begins during this phase. This phase can be quite enlightening, as it commonly reveals problems in the data. Here, we start constructing and training a wide variety of models based on data using a set of parameter constellations. These parameters, naturally, vary from model to model. It is also during the construction of the model that new features or ways of incorporating new data emerge. Modeling and data preparation are closely related, as the preparation steps needed for one model may not be suitable to another.

Evaluation

In the evaluation phase, the goal is to identify which model performs the best. To solve this, the different models and their various parameters are compared to each other.

Deployment

In this final phase, the model that has performed the best during trials is integrated into a business or organization, where it should achieve the desired effect. This is where the value added by the machine learning project comes into play.

1.3 Technical Notions of Learning and Adaption

In this section, we will briefly cover the various types of machine learning and look at the steps of the machine learning cycle. All supervised learning models will follow a specific process, involving the splitting of a dataset in order to evaluate its ability to make predictions.

Definition of Machine Learning

Now that we have seen a practical example of a machine learning system and how such a system can be constructed, we should focus on some fundamental concepts before continuing. Machine learning is a field of computer science that focuses on teaching computers to learn what humans do naturally: learn from data and past experience. Specifically, this is made possible by developing programs and algorithms. This process should result in a computer being able to automatically make predictions and perform certain tasks without relying on predetermined rules or formulae. American scientist Arthur L. Samuel (1959) defined machine learning as the field that gives computers learning abilities without them being explicitly programmed. Mitchell (1997) offers a more technical definition: "A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*" (p. 2).

Types of machine learning

As previously mentioned, machine learning algorithms are commonly divided into three different categories: supervised learning, unsupervised learning, and reinforcement learning. With supervised learning, the training data that we feed the algorithm include the desired solution (i.e., the label). The goal of supervised learning is to learn how to detect the label based on the features provided when applied to new, unlabeled data.

We focus on supervised machine learning, i.e., machine learning based on labeled data. Supervised learning can be further grouped into regression tasks (where we predict continuous numeric values) and classification tasks (where we predict pre-known classes). To perform these tasks, we can choose from a wide variety of algorithms, e.g., linear regression, logistic regression, k-nearest neighbors (KNNs), support vector machines (SVMs), and decision trees.

With unsupervised learning, the provided data are unlabeled, and the objective is to analyze and discover patterns therein. This type of machine learning is usually used for clustering, anomaly detection, dimension reduction, and **association rule learning** tasks. When the system observes the environment and selects and performs an action, this is called reinforcement learning, which is our third and final form of machine learning. If the chosen action results in the desired outcome, the system is rewarded. Otherwise, it experiences a penalty, i.e., negative rewards. Over time, it is expected that the system will devise a strategy in order to collect the most rewards; this is called a policy.

Association rule learning
Discovering and learning about relationships between features is called association rule learning.

Machine Learning Process

Regardless of whether we are working with regression or classification, the process of supervised machine learning will include the following steps:

- learning,
- evaluating the model (i.e., controlling the success of the learning stage), and
- optimizing the model (i.e., adapting the learning process further).

These three steps and how they relate to one other are shown in the following figure.

Figure 4: The Steps of the Machine Learning Cycle

Source: Hofer (2021).

Before the actual learning process can commence, the labeled dataset is first divided into a training set and a test set. One common division is 80 percent training data and 20 percent testing data, although the ratio to be applied is not defined *per se*. Once divided, the model is iteratively trained using the training data and tested using the test data. During this process, the labels of the test set are withheld from the model. The model makes predictions for the test data labels, which are then compared with the actual values of labels in the test set. The delta between the predicted and actual label values is evaluated, and the model is optimized based on this evaluation. By repeating these steps numerous times, the model is improved step-by-step, and prediction performance increases.

To achieve more robust results, it is common to divide the dataset into three parts: a training dataset, a test dataset, and an additional validation dataset. Just as before, the training dataset is still used to train the model, and the test dataset is used for the final evaluation after the model training is completed. This new validation dataset is used to evaluate the prediction performance during the actual learning process. It should be noted that the terms “test” and “validation set” are sometimes used interchangeably in the literature relevant to this topic.

Cross Validation for Evaluating Machine Learning Models

The predictive performance of a machine learning model can be evaluated using a variety of metrics specific to the task of regression or classification. Cross validation is designed to evaluate both regression and classification machine learning models. This resampling technique can be used in a variety of situations, including during model selection and

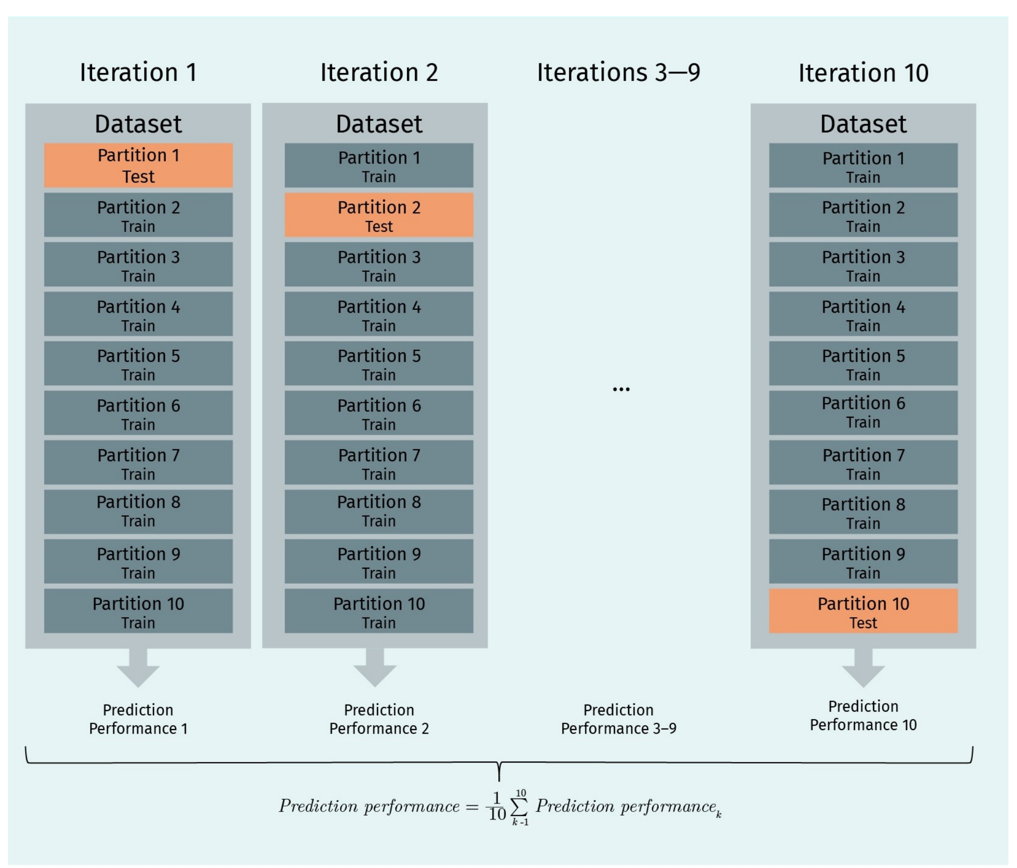
when comparing the various models to see which will produce the best prediction results. It can also be used during the comparison of **model parameters**, thus determining which parameter design yields the best results.

Model parameters
The parameters of a model are configuration variables that decide how the model will behave.

In cross validation, the provided dataset is not split merely into a training and a test set. Rather, these provided data are split k times. In each of these k splits, a different partition is used for testing, and the remaining data are used for training the model. Subsequently, the predictive quality is averaged over these k training and test runs, leading to a more robust conclusion concerning the model's quality.

Cross validation requires just a single parameter k, which indicates the number of splits to make. For this reason, this is also called k-fold cross validation. For example, $k=10$, a common value, reflects a ten-fold cross validation. For a better understanding, the following graphic shows visually how a ten-fold cross-validation works. Cross-validation is very computationally intensive. Instead of training and validating a model once, this is done k times. This disadvantage is kept within limits if the value of the parameter k is kept within acceptable ranges, typical values here are, for example, k=5 or k=10, or by having sufficiently large computing capacities, so that this disadvantage is not severe.

Figure 5: Ten-Fold Cross Validation



Source: Hofer (2021).

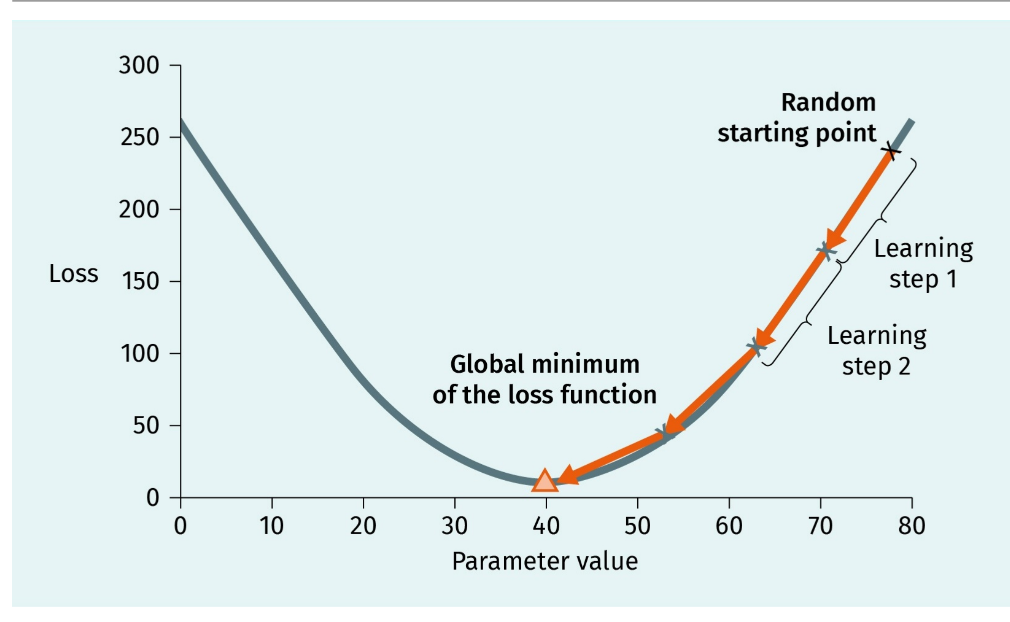
Optimization

The loss function, also called the objective or target function, is the computation of our model's prediction accuracy rate and how much these predictions differ from the true values of the labels. The model's accuracy improves based on the minimization of this loss function. It can be also used to assess the quality of the model when we calculate the accuracy of the model using the test data. The function that minimizes the loss function, the optimizer, iterates over the model's parameter values. During each training iteration, it updates these values, thus incrementally minimizing the loss function and devising a solution.

Gradient descent

Gradient descent is one of the most common optimization algorithms used in machine learning. With gradient descent, the model parameters are iteratively tweaked to minimize the loss function (and find its global minimum). At the start of the training phase, the model parameters are initialized by random values. From there, the goal is to work in the direction of the steepest descent of the loss function gradually, i.e., to take the step offering the largest possible loss reduction. The gradient descent locates the steepest descent by determining the local gradient of the loss function (i.e., the partial derivative of the loss function) and goes in the negative direction of the gradient. This is repeated until the function's minimum has been reached. This process is illustrated in the figure below.

Figure 6: Gradient Descent



Source: Hofer (2021).

An important aspect of gradient descent optimization is the size of the single optimization steps, which is influenced by the chosen learning rate. Multiplying the chosen learning rate with the calculated gradient will result in the size of the particular optimization step. If the learning rate is too low, the convergence of the function (i.e., the finding of the optimum)

may take a very long time. It could even end in a local minimum instead of the global minimum, as shown in the following figure. Conversely, a large learning rate could lead us to the global minimum much quicker. However, we would then run the risk of overshooting the global minimum entirely. The optimal learning rate quickly leads us to the convergence of the function at the global minimum.

Not all loss functions are convex, as can be seen in the following figure. In fact, they may have local minima and plateaus that render finding the global minimum difficult. Stochastic gradient descent can help us solve this problem by taking a subsample of the training observations and using it for the optimization process, making the algorithm faster. The stochastic nature of random sampling also adds a degree of randomness when descending the gradient of the loss function. Although this added randomness does not necessarily guarantee that the algorithm will find the absolute global minimum, it can help the algorithm get sufficiently close, i.e., by allowing it to jump away from any minima and plateaus (Boehmke & Greenwell, 2019).

Figure 7: Learning Rates

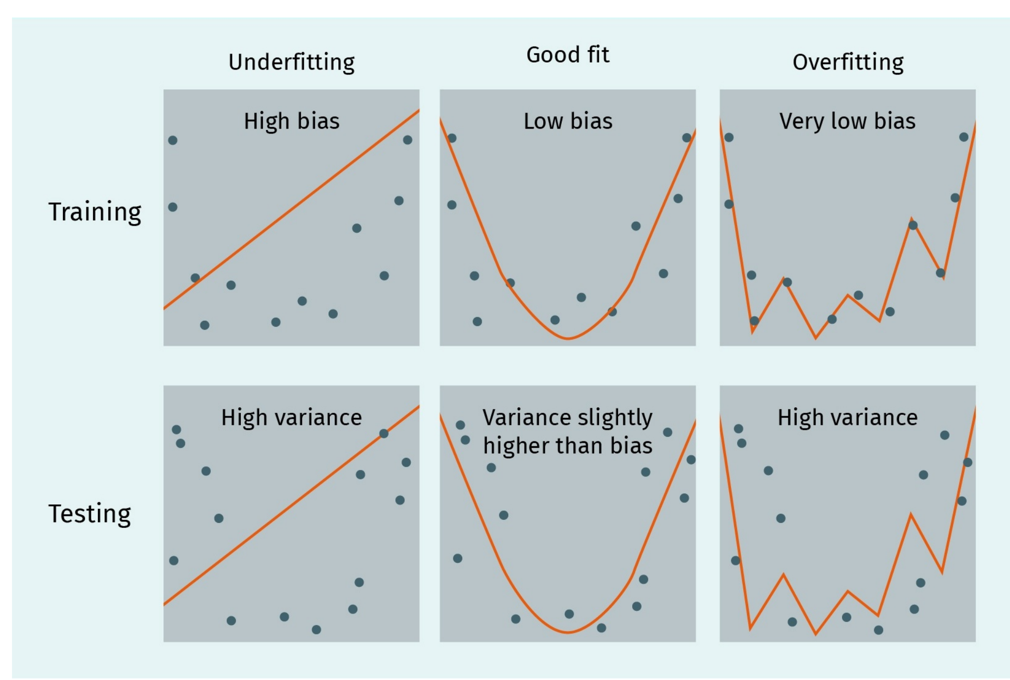
Source: Hofer (2021).

The major advantage of gradient descent optimization compared to other optimization methods is the varying step size. It varies depending on the distance from the minimum we are trying to find. We take larger steps when we are far from the optimum and smaller steps when we are closer. This flexibility is what makes gradient decent so powerful and fast.

1.4 Under- and Overfitting

Overfitting is a phenomenon where a model performs well with the training data and struggles with the test data left unseen during the learning phase. This means that the model fails to generalize the learned knowledge and is unable to apply it to new, previously unseen data. In reality, the model will have simply memorized the training data. To illustrate the problem of overfitting and underfitting, it makes sense to first look at the problem visually.

Figure 8: Overfitting and Underfitting in Regression



Source: Hofer (2021).

This figure shows three different regression models (the lines) and how they perform on the training and the test data (the dots). It is apparent that the first model, the linear model, does not manage to represent the pattern in the training data. Instead, the data have a nonlinear relationship, which this linear model simply does not capture. This results in a high degree of training error (i.e., a high bias). Bias is the inability of a machine learning model to capture the true relationship between the label and the features. It is caused by erroneous assumptions that are inherent to the learning algorithm. This model is too simple to properly capture the underlying relationships in the data. It has already performed poorly on the training data (as it is “underfitted”), which may lead it to make poor predictions when presented with the test dataset. This results in a high error with the test data, i.e., a high variance. By contrast, the third model performs outstandingly with the training data and shows a very low bias. However, it fails to generalize its knowledge to new, unseen data and results in high variance with the test dataset. The model is “overfitted” (i.e., fitted too closely) to the pattern in the training data.

Those who build the models are the ones responsible for avoiding overfitting and underfitting problems. The model must be constructed in such a way that it can recognize and understand the underlying relationships of the data well without losing the ability to generalize what it has learned. This is also called the variance-bias-tradeoff. An optimal balance between bias and variance is required to never underfit or overfit the model. For the sake of completeness, the following figure illustrates the problem of overfitting and underfitting in classification models. Generally, the principle is the same here as it is for regression models: classifiers that do not capture essential patterns in the data reflect underfitting, and classifiers fitted too closely to the training data reflect overfitting.

Figure 9: Overfitting and Underfitting in Classification

Source: Hofer (2021).

Now that we are aware of these problems, how can we avoid building a model that is overfitted or underfitted?

Avoiding Overfitting

The following is a list of measures one can take to prevent overfitting:

- choosing a less complex model
- collecting more training data
- reducing noise in the training data (e.g., fixing data errors and removing outliers)
- using **regularization**

Regularization
When a penalty term that targets higher model complexity is built into a model, this is called regularization.

Avoiding Underfitting

As previously stated, underfitting happens when a model is too simple to learn the underlying structure of the data. Predictions from an underfitting model are bound to be inaccurate, as they cannot precisely represent reality. The following is a list of possible measures that one can take to avoid underfitting:

- selecting a more powerful, mathematically complex model
- extracting better features that represent the real world and feeding it to the algorithms
- reducing constraints (e.g., reducing regularization) on the model

SUMMARY

Data collection and processing have become an integral part of today's modern world. Pattern recognition is a process in which machine learning techniques are used to find and understand underlying insights hidden in data. In this process, labels are assigned an observation based on its features. There are two types of pattern recognition: supervised learning, wherein the training dataset features both labels and features; and unsupervised learning, wherein the training dataset exclusively contains features. For supervised learning, there are many algorithms one can employ, including linear regression, logistic regression, and k-nearest neighbors (KNNs). The Cross-Industry Standard Process for Data Mining (CRISP-DM) is seen as the blueprint for the development of machine learning models and comprises six phases: business understanding, data understanding, data preparation, modeling, evaluation, and deployment.

To ensure the model's accuracy, it must be optimized. The loss function is the computation of the model's prediction accuracy and how far it differs from the true value of the labels. A model's accuracy improves based on the minimization of the loss function. One of the most common methods of model optimization is gradient descent, where the parameters are iteratively altered to minimize the loss function. Overfitting is when a model performs well with training data and poorly with unseen data; underfitting, by comparison, is when the model fails to reflect reality due to its lack of complexity, which causes a high rate of error on both sets of data.

UNIT 2

REGRESSION

STUDY GOALS

On completion of this unit, you will be able to …

- understand the concept of regression and when to use it.
- evaluate a regression model’s performance.
- utilize regularization techniques and understand where they are implemented.
- apply different well-known regression models with the use of Python.

2. REGRESSION

Introduction

Supervised machine learning models are trained using data that include both the inputs and the outputs the machine should learn to predict. These outputs (i.e., the respective labels) can be of categorical or numerical nature. The range of applications of regression models in practice is quite versatile. These are used when the labels are continuous numeric values. In the real world, we see examples of their implementation every day, including in weather forecasts, sales projections, the recording of users visiting a particular website, and individual income balancing. This unit will focus on the concept of regression, presenting and explaining some of the most common algorithms and methods. We will find answers to the following questions:

- How do regression models work, and what is the math behind them?
- How can we evaluate the performance of a regression model?
- How can we reduce bias and variance in regression models?
- How can we apply a regression model with Python?

2.1 Linear Regression

First, let us look at some fundamental regression-related terms widely used in relevant literature and their synonyms.

Table 1: Important Regression Terms and Their Synonyms

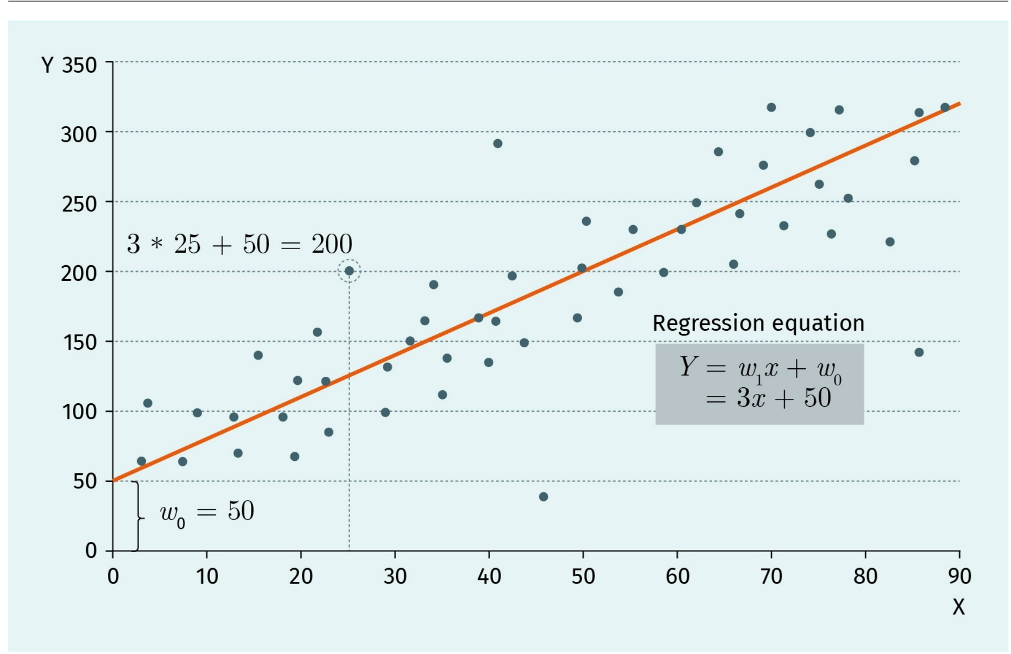
Term	Synonyms
Target variable (Y)	Label, y variable, dependent variable
Input (X)	X variables, independent variables, predictors, features
Coefficient (ω_i)	Weight, slope, regression coefficient
$(y$-axis) Intercept (ω_0)	Bias
Loss function	Cost function, target function, objective function, error function

Source: Hofer (2021).

Algorithm

To understand the concept of regression, we will first focus on linear regression (the simplest form) and then move to some more complex, efficient models. Linear regression tries to predict the output Y, based on an input X or even a larger number of n inputs X_i $(I=1, 2, ..., n)$, i.e., the features. This is done by fitting a straight line through the given cloud of data points that explains the relationship between Y and X in the best possible way. This is illustrated in the graph below, where the data points represent the observations in the training data and the line represents the regression line. These are used to predict the label Y.

Figure 10: Linear Regression Model



Source: Hofer (2021).

Formally, the linear regression model is defined as

$$Y = \omega_0 + \sum_{i=1}^{n} \omega_i X_i$$

where X_i $(I=1, 2, ..., n)$ refers to the n inputs or features, ω_i the unknown weights of the inputs $X_i$responsible for the slope of the regression line, and ω_0 the y-axis intercept. These latter two elements are determined during model training. After this formal introduction to the regression equation, we now address the question of how to determine the weights. As noted previously, the weights (or coefficients) are determined during the training phase of the model, in which the goal is to minimize the presence of residuals.

A residual is the difference between an actual value and its predicted value, i.e., the vertical difference between the data point and the regression line. If the predicted value is lower than the actual value, the residual is positive; if the predicted value is higher, the residual is negative; and if the prediction and the actual value are equal, the residual is zero. The definition of the residual R_I of observation I results from the difference between the actual value of the label y_{en} and its predicted value q_i can be formally expressed as

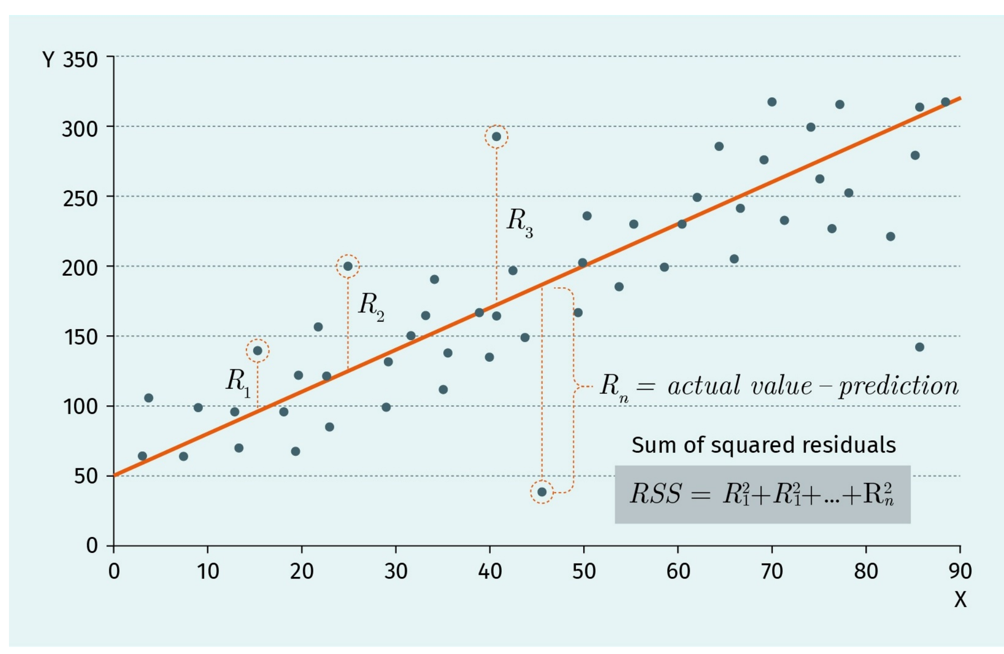
$$R_i = y_i - \widehat{y}_i$$

The goal of the training phase is to adjust the individual coefficients of the regression line so that the loss function is minimized. The loss function most commonly used in linear regression is called ordinary least squares (OLS), which minimizes the residual sum of squares (RSS), calculated as

$$RSS = \sum_{i=1}^{n} R_i^2$$

Squaring the residuals ensures both that positive and negative residuals do not cancel each other out and that larger deviations of the value predicted from the actual value are more significant than smaller deviations. This is illustrated in the figure below.

Figure 11: Plot of Residuals



Source: Hofer (2021).

The charm of linear regression lies in the previously noted understandability of the model and the ease with which the results can be interpreted. If we use the derived regression line from the previous example, we can easily interpret the coefficients determined by OLS and thus the influence of the input features X on the label Y. In our simplified example,

we had only one input feature x_1. For the regression line, we obtained $Y=3x+50$. Thus, for intercept w_0, OLS yielded $w_0=50$ and $w_1=3$ for the coefficient. Here, the intercept $w_0$is to be interpreted as a minimum value that label Y (to be predicted) does not fall below.

One example of this concept in action would be deducing the legal minimum salary when predicting monthly salaries that each observation (i.e., person) receives, regardless of the features. To give a second example, the intercept could be a minimum measurement when predicting an individual's height that is exceeded by each observation. The coefficient w_1 indicates how much the label Y changes when the feature x_1 changes. In our example, the regression line $Y=3x+50$ results in a predicted value $Y=80$ for $x=10$ and a prediction of $Y=110$ for $x=20$. Thus, increasing x_1 from 10 to 20 results in an increased prediction of Y by 40.

Metric for Measuring the Prediction Performance of a Regression Model

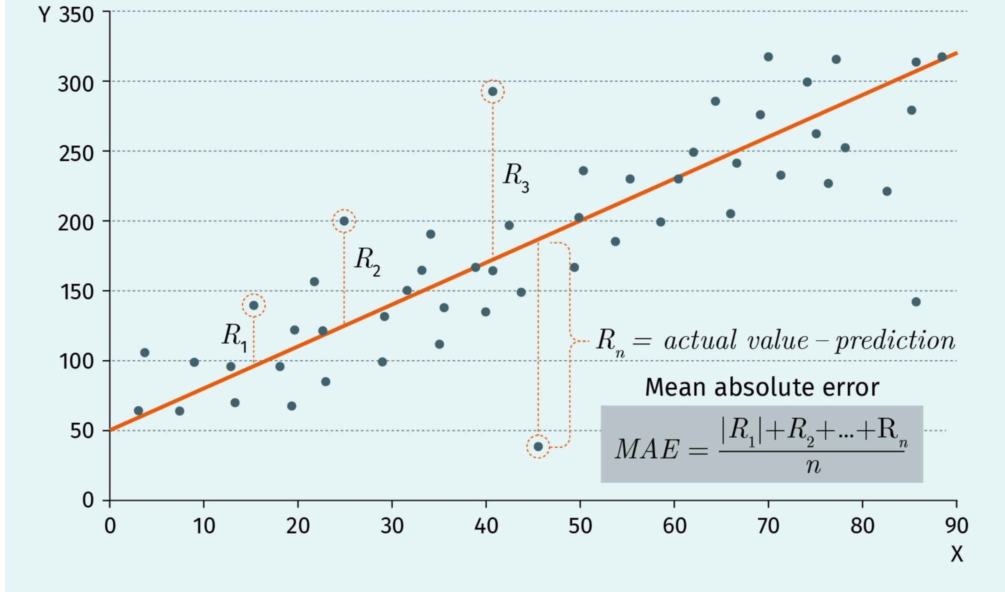
To measure how good the predictions of a regression model are, a wide variety of methods can be used, i.e., **metrics**. We will now focus on the three most popular metrics put into practice: the mean absolute error (MAE), the mean squared error (MSE), and the root mean squared error (RMSE).

Metric
A measure to evaluate the performance of machine learning model is called a metric.

Mean absolute error (MAE)

The MAE is the absolute difference between all predictions and the actual values (i.e., the absolute residuals), averaged over the number of predictions. This is illustrated in the following figure.

Figure 12: Illustration of Mean Absolute Error



Source: Hofer (2021).

The residuals are taken to the absolute value here so that positive and negative deviations do not cancel each other out. The MAE is helpful because it is reported in the variable unit being forecasted. This allows us to make more precise statements, e.g., “The regression model can predict the sales figures with a mean deviation of 50 EUR.” The compact, formal notation of MAE is as follows:

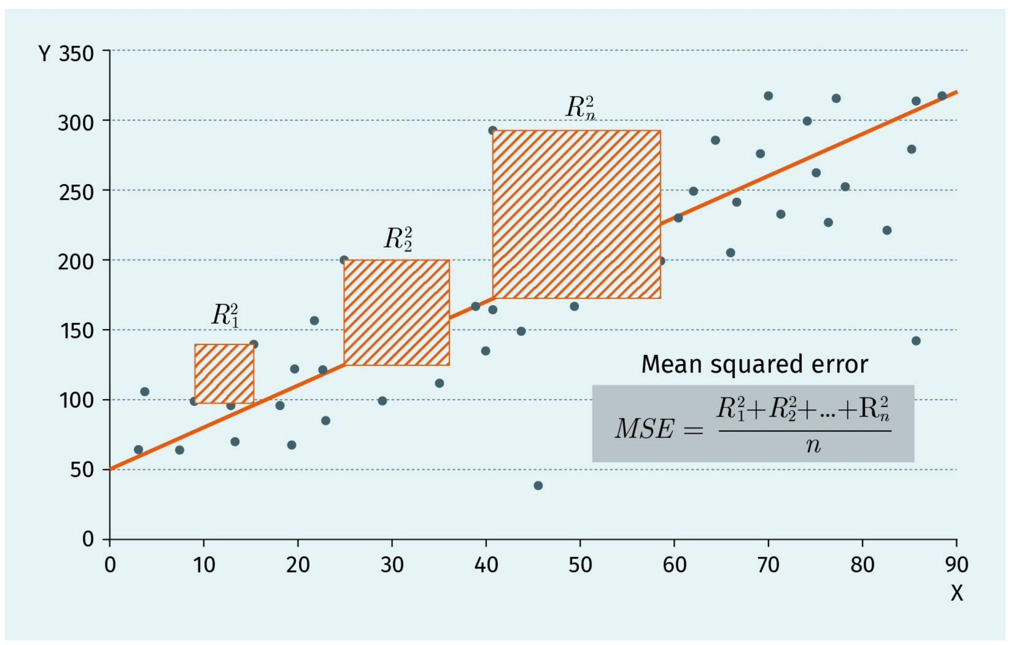
$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \widehat{y}_i|$$

Following this expression, the MAE results from the summed absolute differences between the actual value y_i and the predicted value $\hat{y}_i$ for each observation i, averaged over all predicted observations n. When dividing by n, we keep the error measure consistent as we move from a small collection of observations to a larger collection. This move alone would otherwise increase the error without taking the mean.

Mean squared error (MSE)

The MSE differs from the MAE in only one aspect: The deviations between the actual and the predicted values are taken to the square and not the absolute value. In contrast to the MAE, its results cannot be interpreted in a stand-alone manner. Here, the unit of the label is also squared, and statements such as “The regression model is able to predict the sales figures with a mean deviation of 50^2 EUR” are not particularly useful. Instead, the squaring provides another desired effect. While the MAE penalizes both smaller and larger deviations from the actual value in the same way, squaring the MSE makes deviations larger from the actual value more influential than smaller ones. The MSE, therefore, penalizes outliers more heavily than it does small deviations. This effect is shown in the figure below.

Figure 13: Illustration of Mean Squared Error



Source: Hofer (2021).

The compact, formal definition of MSE is

$$MSE = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - \widehat{y}_i\right)^2$$

Following this expression, the MSE results from the summed squared differences between the actual value y_{en} and the predicted value q_i for each observation I, averaged over all predicted observations n.

Root mean square error (RMSE)

The RMSE is the square root of the MSE, whereby the result is telegraphed in the unit of the label being predicted. This also provides increased interpretability as compared to the MSE alone, and the resulting scores yield smaller, more manageable numbers. The compact, formal definition of RMSE is

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \widehat{y}_i\right)^2}$$

Following this expression, the RMSE results from the root of the summed squared differences between the actual value $y_{en}$and the predicted value q_i for each observation I, averaged over all predicted observations n. The RMSE is one of the most common measures against errors when evaluating machine learning models. For presentation purposes, the MAE is often also reported due to its easy comprehensibility.

Implementation

When developing machine learning solutions, it is common to use libraries, which means that not every algorithm and procedure used has to be programmed by scratch itself. One of the most popular Python libraries for developing machine learning solutions is scikit-learn. This library supports several operations natively, like classification, regression, clustering, and includes a wide variety of machine learning algorithms. In the following, we will show a sample implementation of linear regression using the scikit-learn library. In addition, we will show how the results of the trained and applied model can be visualized using the matplotlib library.

Let's choose a dataset of person's professional experience in years and the task of predicting the salary based on this information.

We typically start by importing the libraries we want to use. We first import the matplotlib library to visualize our results and then the pandas library, probably the most popular Python library when it comes to loading, preparing, and storing data.

Code

```
import pandas as pd
import matplotlib.pyplot as plt
```

Next, we import the dataset and divide it to input features X and labels Y.

Code

```
Dataset = pd.read_csv('Salary_Data.csv')
X_train = dataset.drop(columns=[‘Salary’])
y_train = dataset.iloc[:,1].values
```

Now we import an instance of linear regression using the scikit-learn library and train the model based features X_train and labels y_train of the training dataset using the `fit()` method.

Code

```
from sklearn.linear_model import LinearRegression
regressor=LinearRegression()
regressor.fit(X_train,y_train)
```

Next, we plot the data and the trained model using the matplotlib library.

Code

```
plt.scatter(X_train,y_train,color='red')
plt.plot(X_train,regressor.predict(X_train),)
plt.title('Salary vs Experience(Train set)')
plt.xlabel('Experience in years')
plt.ylabel('Salary')
```

In the resulting plot, we can now look at the data points and the trained model and easily see the positive correlation between the professional experience in years and the salary at a glance.

Figure 14: Linear Regression Plot Generated by the Matplotlib Library

Source: Hofer (2021).

The intercept and the value of the coefficient of the trained model can also be shown.

Code

```
print(regressor.intercept_, regressor.coef_)
```

As explained before, this regression coefficient can be interpreted. If we assume that the coefficient is 1000, then one more year of professional experience would increase the salary by 1000 units according to the model. The trained model can now be used to generate predictions for new unseen data. We can do this with the `predict()` method.

Code

```
y_pred = regressor.predict(X_newdata)
```

2.2 Lasso and Ridge Regularization

Linear models reflect a simple, effective approach to building predictive models. Nevertheless, they have their limitations: on datasets with a high number of features, they tend to lose their generalizability and run into overfitting. For such datasets, one should make

use of regularization methods, which aim to prevent overfitting in predictive models. In this section, we will take a closer look at the most popular regularization techniques for reducing model complexity through the restriction or regulation of estimated coefficients: ridge regression, lasso regression, and elastic net (a combination of the first two).

Ridge Regression

Ridge regression regularizes the linear regression, putting constraints on the coefficients by adding a penalty term

$$\alpha \sum_{i=1}^{n} \omega_i^2$$

to the loss function. We introduce a penalty for larger numbers of regression coefficients ω_i, i.e., the more input variables we incorporate into our model, the higher the penalty. By doing this, we render uninformative variables superfluous, as they are unable to compensate for the loss that they introduce. Therefore, it is only worth inputting informative variables. The loss function takes the form

$$RSS + \alpha \sum_{i=1}^{n} \omega_i^2$$

Norm
A norm is a mapping that assigns a number to a mathematical object (e.g., a vector) intended to describe its size.

Multicollinearity
When a feature is highly correlated with one or more of the other features in a regression model, we speak of multicollinearity.

The penalty size, also known as the L2 **norm**, can take on a wide range of values and is controlled by the tuning parameter α. When $\alpha=0$, there is no effect. Our loss function equals the normal OLS regression loss function; as $\alpha \to \infty$**,** the penalty becomes large and forces the coefficients towards zero. However, we should note that the coefficients here do not actually take the value of zero, meaning that all features have a contributory role in the prediction result. Ridge regression is particularly useful for **multicollinear** datasets comprising a vast number of features, many of which we assume are relevant and, consequently, should be accounted for by the model. Essentially, ridge regression pushes correlated features toward each other, thereby preventing one from being strongly positive and the other strongly negative. It also pushes many of the less important features toward zero, which helps to highlight any important signals in the data (Boehmke & Greenwell, 2019).

Application

Using scikit-learn, an open source Python library, we can apply ridge regression very easily, here with a value of $\alpha=1$. We can then train the model with the help of the `fit()` method.

Lasso Regression

Lasso (least absolute shrinkage and selection operator) regression is an alternative regularization method. It differs from ridge regression only in that the L2 norm is exchanged for the L1 norm. We do this by introducing a penalty term

$$\alpha \sum_{i=1}^{n} |\omega_i|$$

to the loss function so that it is defined as follows:

$$RSS + \alpha \sum_{i=1}^{n} |\omega_i|$$

The lasso penalty has the ability to push coefficients to zero, thereby canceling out the influence certain features would have on the model. This is in stark contrast to the ridge penalty, which can only push the variables closer to zero. Consequently, lasso regression can also be used for feature selection (Boehmke & Greenwell, 2019).

Application

Lasso regression can be imported, initialized, and trained just as easily as ridge regression using the scikit-learn library.

Code

```
from sklearn.linear_model import Lasso
lasso_reg = Lasso(alpha=0.1)
lasso_reg.fit(X_train,y_train)
```

Elastic Net

Elastic net acts as a middle ground between the ridge and lasso regression concepts, uniting the advantages of both into one method. It selects important coefficients, as does lasso regression, and is effective in handling correlated features, as is ridge regression. The penalty term used here

$$\alpha \sum_{i=1}^{n} \left(r\omega_i^2 + (1-r)|\omega_i| \right)$$

is a combination of the ridge and lasso penalty terms. When $r=0$, it represents lasso regression; when $r=1$, it represents ridge regression. For all values $0<r<1$, elastic net affects both the ridge and the lasso regression penalties. The larger the value of r, the more weight is given to the ridge regression term.

Application

Much like ridge and lasso regression, elastic net can be applied using scikit-learn. During initialization, the `l1_ratio` parameter must be defined with a value in the range `0 ≤ l1_ratio ≤ 1`, analogous to the explanation of the parameter r above. If `l1_ratio=0` is

set, the penalty is a pure ridge penalty. If `l1_ratio=1` is set, the penalty is a pure lasso penalty. If values `0 < l1_ratio < 1` are chosen, the penalty is an appropriately weighted combination of both.

Code

```
From sklearn.linear_model import ElasticNet
elastic_net = ElasticNet(alpha=0.1, l1_ratio=0.5)
elastic_net.fit(X_train,y_train)
```

We will now examine the regression coefficients that result from adding an artificially generated `YearsExp/100` feature to the dataset. Linear, ridge, and lasso regression, as well as elastic net, will be considered. `YearsExp/100` is highly correlated with `YearsExp` and does not introduce any information when used for regression.

Code

```
dataset['YearsExp/100'] = dataset['YearsExperience']/100
```

When we initialize the models with the parameterizations seen earlier and train them on the artificially expanded dataset, we obtain the following coefficients using the `coef_` method.

Table 2: Coefficients of Features "YearsExperience" and "YearsExp/100"

	Model	Coefficients
0	Linear regression	[9449.017419713111, 94.4901741971311]
1	Ridge regression	[9444.97381418553, 94.4497381418512]
2	Lasso regression	[9449.94947649658, 0.0]
3	Elastic net (l1_ratio=0.01)	[9330.37945693112, 93.29375135668731]
4	Elastic net (l1_ratio=0.99)	[9448.735918738957, 0.0]

Source: Hofer (2021).

We note that lasso regression has set the coefficient of the `YearsExp/100` feature to zero because of its correlation with the `YearsExperience` feature. From the console output for the two elastic net models, we can see two very clear results: the model for an `l1_ratio` approaching zero nears ridge regression; and, an `l1_ratio` approaching one nears lasso regression.

2.3 Generalized Linear Models

Generalized linear models (GLMs) are a category of expanded linear regression models. Linear regression, as previously described, attempts to predict a continuous variable using the linear combination of descriptive features. Linear regression is built upon the following basic assumptions (Hardin & Hilbe, 2007):

- There is a linear relationship between the target variable Y and the features X.
- The residuals of the model are normally distributed.
- The residuals have a constant variance, i.e., there is homoscedasticity.

If, for example, the dependencies in the data do not follow a linear relationship, or the other assumptions made concerning linear regressions are not true, applying linear regression would simply yield poor predictions. GLMs address these shortcomings. Just like the linear models they generalize, the purpose of GLMs is to specify the relationship between the target variable Y and some number of features X. General models are developed by relaxing the assumptions of linear models. By restructuring the relationship between the linear predictor and the fit, we can "linearize" relationships thought to be strictly nonlinear (Hardin & Hilbe, 2007). GLMs all essentially comprise the following three components:

1. A linear predictor $\eta_i = \omega_0 + \omega_1 x_{1i} + \ldots + \omega_p x_{pi}$
2. A probability distribution that generates the target variable Y
3. A monotone differentiable **link function**$g(\mu_i) = \eta_i$ describing how the mean depends on the linear predictor $\eta_{i.}$

Link function
The link function specifies the link between the linear predictor and the probability distribution.

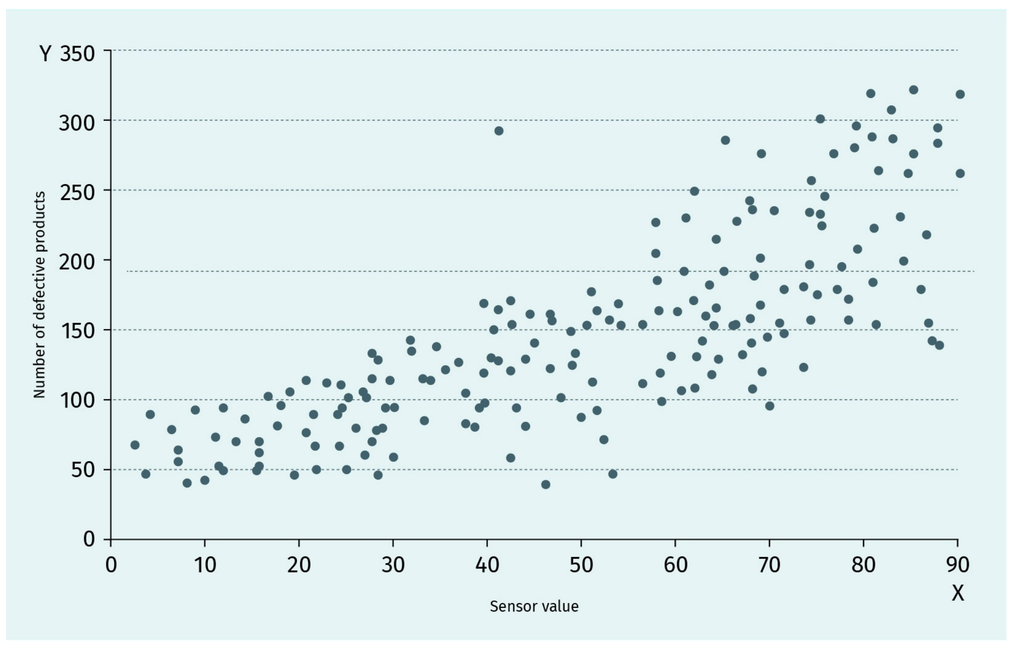
Because the link function is selected separately from the random component (i.e., the probability distribution), we have greater flexibility in our modeling. By introducing the link function and the **additive effects** it can represent, there is no need for a constant variance. By specifying these three GLM components, a wide variety of models can be built in a flexible way, i.e., to represent the underlying distribution and dependency structures of the provided data. Let us illustrate the concept of GLM with an example from Kida (2019).

Additive effect
This is a joint effect of two or more features on the target variable.

GLMs in action

Suppose we want to predict the number of defective products Y with a sensor value of the producing machine as explanatory feature X. The scatter plot can be seen in the following figure.

Figure 15: Number of Defective Products



Source: Hofer (2021).

Applying linear regression to such data would be problematic for two reasons in particular. First, there is an obvious nonlinear relationship between the number of defective devices Y and the sensor values (i.e., there is non-constant variance). Second, Y only takes on discrete values (as opposed to continuous ones). It would be much more appropriate here to use the Poisson regression, another type of GLM. With Poisson regression, our GLM is formulated as follows:

$$ln\eta_i = \omega_0 + \omega_1 x_{1i}$$
$$y_i \sim Poisson(\eta_i)$$

The first expression describes our linear predictor and expresses the link function in the form of the logarithm, which is typical in such scenarios. The second expression describes our probability distribution, which expresses that our target variable Y follows a **Poisson distribution**. According to Kida (2019), the Poisson distribution has only one parameter η, which specifies both the mean and the standard deviation. Thus, the larger the mean, the larger the standard deviation, as depicted in the following figure.

Poisson distribution
The Poisson distribution is used to model the count of events that occur in a given time interval.

Figure 16: The Poisson Distribution with Different Values for η

η = 1 η = 5 η = 10

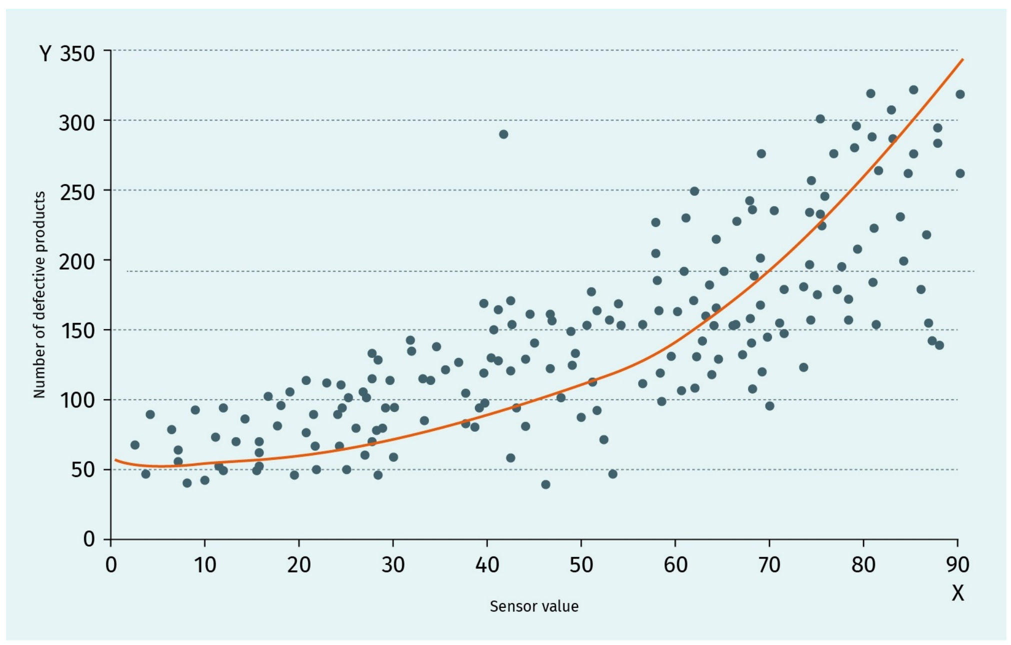
Source: Hofer (2021), based on Kida (2019).

To derive predictions, we resolve the expression shown previously to η_i by using the inverse of the logarithm, i.e., the exponential function. This gives us the following equation:

$$\eta_i = \exp(\omega_0 + \omega_1 x_{1i})$$

With an increase in inputs X, the predicted outputs Y increase exponentially. Consequently, this means that the model is capable of representing a nonlinear relationship between the target variable Y and the descriptive features X. When we apply Poisson regression to our data, the result appears as depicted in the figure below. The red curve represents the predictions generated by Poisson regression.

Figure 17: Poisson Regression Applied to Our Data



Source: Hofer (2021).

The mathematics required for this process is somewhat elaborate and beyond the scope of this unit. The interested reader is referred to McCulloch and Searle (2001, p. 135). At this point, let us remember that, in GLMs, we do not estimate y directly with a linear combination of features x. Rather, we estimate the mean of the link function by this linear combination. The link function connects the linear combination of features x with any distribution of the exponential family (not only the normal distribution), from which we assume the values of y to be predicted are drawn. In other words, we push a link function and probability distribution between our y and x values. As we do that, our data need not fulfill the somewhat strict assumptions made by regular linear regression. It is notable that, in regular linear regression, we have the choice between the ordinary least squares (OLS) and maximum likelihood methods for estimating the regression coefficients. By comparison, in GLMs, we only use maximum likelihood for this estimation.

In Python, GLMs can be used with the help of the statsmodels library. The code for initializing and fitting the Poisson regression is as follows:

Code

```
import statsmodels.api as sm
exog, endog = sm.add_constant(x), y
mod = sm.GLM(endog, exog,
       family=sm.families.Poisson(
            link=sm.families.links.log))
res = mod.fit()
```

The terms endog (endogenous) and exog (exogenous) describe the target variable Y and the features X, respectively,in the statsmodels library. We use the `add_constant` method now to add our constant ω_0 for the intercept. Otherwise, our linear predictor would be in the form of $\omega_1\ x_{1i}$. We should note that the link function does not have to be specified here, as it acts as the default logarithm for Poisson regression.

Link functions

Just as we used the logarithm as the link function for Poisson distribution, the link functions of other probability functions can also be taken by default. Here are a few of the common probability functions and the link function typically used with each (Ray, 2017).

Table 3: Some Probability Distributions and Their Link Functions

Distribution	Use	Notation	Link function
Gaussian	Linear repose	$N(\mu,\sigma 2)$	"Identity": μ
Poisson	Counts of events	$N(\mu)$	$\text{Log}(\mu)$
Bernoulli	Outcome of single yes/no occurrences	$\text{Bern}(p)$	$\text{Logit}(\mu)$
Binomial	Count of yes occurren-ces out of n yes/no events	$\text{Bin}(n,\ \mu)/n$	$\text{Logit}(\mu)$

Source: Hofer (2021).

For posterity, we should note that the logit is the natural logarithm of an eventuality, i.e., the probability p divided by the counter probability $1\text{-}p$. This can be formally expressed as follows:

$$logit\left(p\right) = \left(\frac{p}{1-p}\right)$$

2.4 Logistic Regression

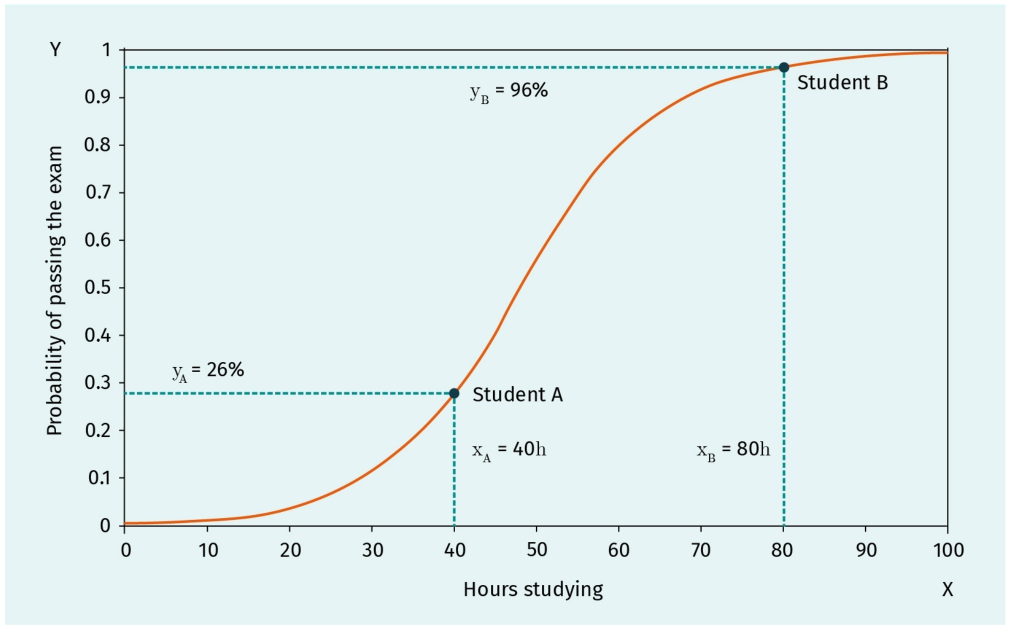
Logistic regression, a closely related method to linear regression, is a special form of GLM. Instead of fitting a line, logistic regression uses an S-shaped logistic function. This curve ranges between 0 and 1, as shown in the figure below.

Figure 18: Logistic Regression

Source: Hofer (2021).

There is a particular property of note here that has become popular for binary classification problems: the logistic function values are all in the range of values 0 and 1. This property also has a tendency to push the values against the boundaries of this range (i.e., toward 0 or 1). Here, the output value Y expresses the probability of class membership. The following figure illustrates this using a simplified example, where the number of learning hours is used to predict whether a student will pass an exam. The x-axis measures the "learning hours" feature (i.e., the time the student has spent studying the material), and the y-axis measures the probability that any given student will pass the exam.

Figure 19: Logistic Regression as a Binary Classifier



Source: Hofer (2021).

For student A, who spent 40 hours studying, the logistic regression model predicts a 26 percent chance of passing the exam. By comparison, student B, who has spent 80 hours studying, has a 96 percent chance of passing. Therefore, the classification output for student A is “will fail the exam” and “will pass the exam” for student B. Formally, the logistic function is defined as follows:

$$p\left(X\right) = \frac{e^{\omega_0 + \omega_1 X}}{1 + e^{\omega_0 + \omega_1 X}}$$

As in linear regression, the ω parameters stand for the regression coefficients and X for the input feature. $p(X)$ can be interpreted as the initial probability of class membership (i.e., the probability of passing the exam). We should note that this only defines simple logistic regression when considering one input feature. That being said, this definition can be expanded to include multiple features X for the prediction. This is called multiple logistic regression and is defined as follows:

$$p\left(X\right) = \frac{e^{\omega_0 + \omega_1 X_1 + \ldots + \omega_p X_p}}{1 + e^{\omega_0 + \omega_1 X_1 + \ldots + \omega_p X_p}}$$

As with other classification algorithms, logistic regression has a decision boundary, which is decisive for the class to which an observation is assigned. In the case of logistic regression, we also speak of the threshold probability. The threshold probability marks the class boundary. Although it is often 50 percent, it can vary depending on the purpose of the model. Formally, it can be expressed as follows:

$$y = \begin{cases} 0 & if \quad p \leq 0.5 \\ 1 & if \quad p > 0.5 \end{cases}$$

In this case, the threshold probability is 50 percent. If the output of the logistic function is less than or equal to 0.5, the observation is assigned to class 0 (e.g., the student fails the exam). By comparison, if the output of the function is greater than 0.5, the observation is assigned to class 1 (e.g., the student passes the exam).

Model Training via Maximum Likelihood

While linear regression uses ordinary least squares (OLS) to fit a line to the data, logistic regression does not, as it has no concept of residuals. Therefore, we have to use a different mechanism to fit the curve to the data, namely the maximum likelihood method. Using this method, we seek estimates for our regression coefficients ω such that the predicted probability $\mathrm{p}(X)$ for each observation matches the observed output value Y of the observation as closely as possible. In other words, we are trying to find ω such that plugging these estimates into the model for $p\ (X)$ yields one of the two following options: a number close to one for all observations actually belonging to the positive class and a number close to zero for all observations not belonging to this class (Boehmke & Greenwell, 2019). In mathematical terms, this can be formally expressed as a likelihood function:

$$l(\omega_0, \omega_1) = \prod_{i:y_i=1} p(X_i) \prod_{i':y'_i=0} [1 - p(x'_i)]$$

The likelihood function expresses the probability $\mathrm{p}(\mathrm{X_i})$ that an observation i belongs to the class based on its features $\mathrm{X_i}$.It also expresses the counter probability $1\text{-}\mathrm{p}(\mathrm{x_i})$ that it does not belong to the class. For the coefficients ω_0 and ω_1,the values are chosen that maximize the stated likelihood function by approximating the output probabilities to the actual output values.

Application

Let us look at an example from Géron (2019) using the popular Iris dataset, which comes bundled in the scikit-learn library. The dataset consists of 50 samples of three Iris flower species (*Iris setosa*, *Iris versicolor*, and *Iris virginica*) and contains four features (the respective lengths and widths for both the sepals and petals). Géron (2019) builds a binary classifier that detects one of the species, the *Iris virginica*, by using the logistic regression.

First, we load the Iris dataset using scikit-learn.

Code

```
from sklearn import datasets
iris = datasets.load_iris()
```

Then, we split it into features X and labels Y.

Code

```
import numpy as np
X = iris["data"][:, 3:] # petal width
y = (iris["target"] == 2).astype(np.int)
# 1 if Iris virginica, else 0
```

Next, we initialize and train an instance of the logistic regression.

Code

```
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(X, y)
```

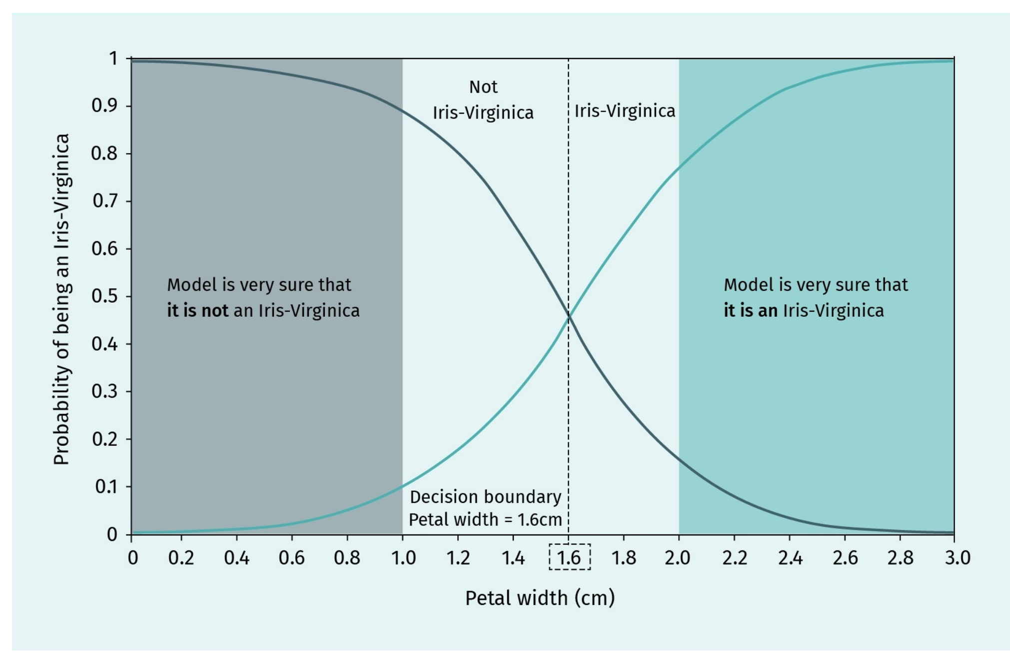
Finally, with the help of the matplotlib library, we let the model predict the class probabilities and plot the estimated probabilities for Iris flowers with petal widths ranging from 0 to 3 cm.

Code

```
from matplotlib import pyplot as plt
X_new = np.linspace(0, 3, 1000).reshape(-1, 1)
y_proba = log_reg.predict_proba(X_new)
plt.plot(X_new, y_proba[:, 1], "g-",
   label="Iris virginica")
plt.plot(X_new, y_proba[:, 0], "b--",
        label="Not Iris virginica")
```

Let us take a look at the resulting plot.

Figure 20: Plot of Estimated Probabilities with Decision Boundary



Source: Hofer (2021), based on Géron (2019).

We can see from the figure above that the classifier is very sure that a flower is of the *Iris virginica* variety when its petals are at least 2 cm in width (i.e., the model outputs a high probability for the class belonging). Comparatively, the classifier is also very sure that a flower is not of this variety when petal width is under 1 cm. If the petal width moves between these ranges (i.e., $1\text{cm} < \text{petal width} < 2\text{cm}$), the classifier is less sure that the flower in question is an *Iris virginica*. If we ask the model to output only the predicted class membership via the `predict()` method (instead of the predicted probability via the `predict_proba()` method used earlier), then only the class that the model considers to be more probable is shown. The model uses the default threshold probability of 50 percent, which is approximately 1.6 cm in this case. Thus, if the width of the flower's petals is greater than 1.6 cm, the classifier will assume that it is an *Iris virginica*.

SUMMARY

Regression models are a type of supervised machine learning where the label to be predicted is a continuous numerical value. In practice, the range of application for these models is very broad, and we see them implemented every day (e.g., in weather forecasts, sales projections, the recording of users visiting a particular website, and individual income balancing). Probably the simplest form of a regression model is linear regression, which tries to predict the label Y based on one or more fea-

tures X. This is done by fitting a straight line through the given cloud of data points that explains the relationship between Y and X in the best possible way.

Linear models are a very simple, yet effective approach to building predictive models. Nevertheless, with datasets with a high number of features, they tend to lose their generalizability and have overfitting problems. Regularization methods (e.g., ridge regression, lasso regression, and elastic net) can be used to counteract this.

If the dependencies in the data do not follow a linear relationship, applying a linear regression would result in poor predictions. Generalized linear models (GLMs) help to address this shortcoming. GLMs are developed by relaxing the assumptions of linear models so that we can "linearize" relationships thought to be strictly nonlinear. Logistic regression is a special form of GLM in which an S-shaped logistic function is fitted (instead of a line). The values of the logistic function are between 0 and 1, which also makes it popular for binary classification problems where the boundary values of the function represent the two classes.

UNIT 3

BASIC CLASSIFICATION TECHNIQUES

STUDY GOALS

On completion of this unit, you will be able to …

– understand the concept of classification and when to use it.
– evaluate the prediction performance of a classification model.
– apply two very popular classification models using Python.

3. BASIC CLASSIFICATION TECHNIQUES

Introduction

Classification is a supervised learning technique that uses a provided training dataset containing both inputs and outputs in the form of a pre-determined number of categorical class labels. The classification algorithm therefore learns the characteristics of the classes provided in the training dataset and can then categorize a previously unseen observation by assigning a class label to it based on its feature values. Classification algorithms are used in a wide range of practical situations. **Classifiers** can be used, for example, to predict whether a customer will buy a product or cancel a service. They can also be used in image recognition, e.g., to help identify objects appearing in a video stream. It is common to divide classification algorithms into binary and multi-class classifiers based on the number of output classes. With binary classifiers, the algorithm predicts the class membership for an observation and assigns it to one of two possible classes. An example of this is the prediction of whether a web store patron is actually a fraud. Multi-class classifiers, by contrast, predict a class based on a predefined set of classes, e.g., the classification of objects in an image or the language of a certain text.

Classifier
A model used for classification is also called a classifier.

A further categorization of classification algorithms can be made based on how they learn. Classifiers are trained using either a lazy learner or an eager learner. Lazy learners perform only minimal training, sometimes even none at all. According to Mitchell (1997), "we call these methods lazy because they defer the decision of how to generalize beyond the training data until each new query instance is encountered" (p. 244). Lazy learners usually store the training dataset until they receive the test dataset or the observation to be classified. Any additional training data received by the classifier are added to those already existing. Future predictions are then made using this expanded and larger training dataset. In comparison to eager algorithms, lazy classification algorithms need less time for training and more time to make a prediction. This is because a comparison is made to the training data each time a prediction is to be formed. Indeed, "we call this method eager because it generalizes beyond the training data before observing the new query" (p. 244). The training phase of eager learners lasts longer than that of lazy learners. Nevertheless, they need less computation time to make a prediction.

In this unit, we will familiarize ourselves with two well-known classification algorithms, each using a different learner. We will introduce the k-nearest neighbor classifier as an example of a lazy learner and naïve Bayes as an example of an eager learner. We will find answers to the following questions:

- How do the k-nearest neighbor and the naïve Bayes algorithm work?
- How can we evaluate a classification model's performance?
- How can we apply both algorithms using Python?

3.1 K-Nearest Neighbor

Let us assume we want to predict whether a person's monthly income will be above or below 4,000 EUR. In order to make an accurate prediction, we could ask this person's coworkers (i.e., neighbors) about their salary to determine if said person earns above or below the threshold. If we then collect further information about this person and their coworkers (e.g., age, level of education, time spent in the field, department, or job title), we should be able to make an even more accurate prediction.

The idea behind nearest neighbor classification is to derive a prediction for one observation by using a number of similar observations (i.e., neighbors) from the training dataset. The central task of the k-nearest neighbor algorithm is to choose the number of k-nearest neighbors by which the prediction for the observation is deduced. The following figure depicts the k-nearest neighbor algorithm by providing an example of a different number of neighbors ($\mathrm{k}{=}3,\ \mathrm{k}{=}6,\ \mathrm{k}{=}9$). Depending on the number of k-nearest neighbors chosen for the classification of the new unknown observation (represented below by the circle), the algorithm makes a different prediction about its class membership and thus determines whether to assign this observation the class label "rectangle" or "triangle."

If we decide to infer the class membership using the three most similar observations from the training dataset (i.e., if we set $\mathrm{k}{=}3$), the algorithm predicts the class membership "rectangle." If we take the six most similar neighbors into consideration and set $\mathrm{k}{=}6$, the algorithm makes the prediction "triangle." If we decide to set $\mathrm{k}{=}9$, we also receive the prediction that the new observation is a rectangle.

Figure 21: K-Nearest Neighbor Algorithm with Two Classes and k=3, k=6, and k=9

?

$k = 3$
Assign to class
“rectangle”

$k = 6$
Assign to class
“triangle”

$k = 9$
Assign to class
“rectangle”

Source: Hofer (2021).

In this example, the k-nearest neighbor acts as a binary classifier. It can be extended easily for multi-class classification, i.e., more than two classes. The k-nearest neighbor can also be tweaked to perform regression. This is where we simply take the mean value and output it as the predicted continuous value (as opposed to taking the vote of the neighbors).

Since the k-nearest neighbor algorithm is non-parametric in nature (which means there is no need to build any model or tune its parameters), it is easy to process, understand, and implement. However, the classification phase would prove to be more expensive in terms of computation. The k-nearest neighbor algorithm does not learn mathematical functions (i.e., decision boundaries) like other classification algorithms do. Other algorithms infer a prediction based on these decision boundaries. By comparison, k-nearest neighbor merely memorizes the training dataset. For this reason, the k-nearest neighbor algorithm is also called a lazy algorithm.

Algorithm

The lazy learner used as a training algorithm by the k-nearest neighbor classifier does not require much computation time. The classification algorithm that it uses, however, is computationally intensive. It can be broken down into the following three phases:

1. Loading the data
2. Selecting a value for k that specifies the number of neighbors
3. Classifying the required observations

The third and final phase of this algorithm comprises the following sequence of actions:

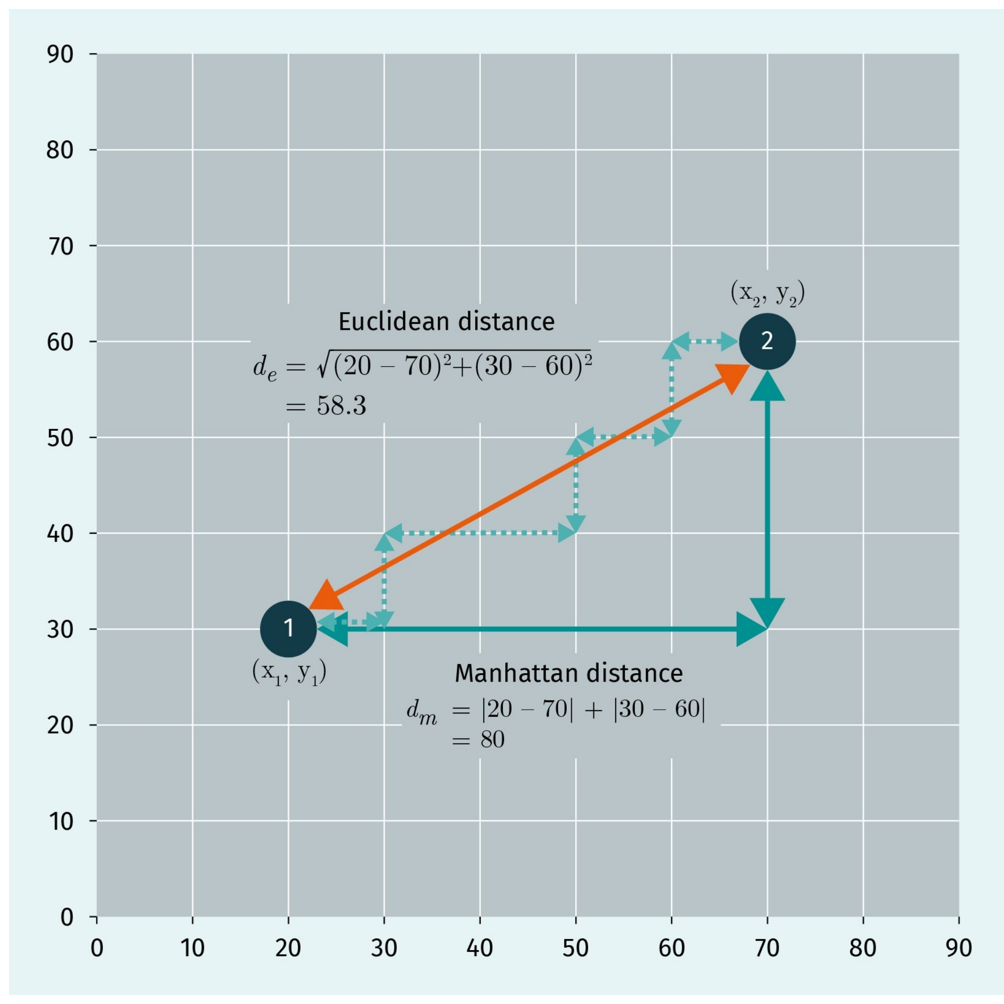
- calculating the distance to all observations of the training dataset
- obtaining the k-nearest neighbors from the training dataset
- obtaining the classes of these k-nearest neighbors
- assigning the dominant class label to the observation

The actual computational work in the k-nearest neighbor algorithm lies in calculating the distance between the observation to be classified and each point in the training data. Once this has been completed, the k-nearest neighbors can be identified. This is where distance measures come into play.

Distance Measures

In principle, the distance measure used in the k-nearest neighbor algorithm can be chosen freely. Nevertheless, the two most popular distance measures are the Euclidean distance and the Manhattan distance (Boehmke & Greenwell, 2019). The easiest way to show how these two distance measures are calculated and how they differ is to use an illustrative example, as shown in the figure below.

Figure 22: Calculation of Euclidean Distance and Manhattan Distance



Source: Hofer (2021).

In this example, we calculate both the Euclidean distance and the Manhattan distance between the first point with coordinates $(x_1 = 20, x_2 = 30)$ and the second point with coordinates $(x_1 = 70, x2 = 60)$. The Euclidean distance and its calculation are represented by the direct, diagonal line between the two points, shown above in orange. The Manhattan distance is composed of the distances added on the axes between the two points, shown above in green. As one could surmise from the figure, the Manhattan distance (also called the city block distance) gets its name from its composition. Much like the street layout in Manhattan (New York), only horizontal and vertical (i.e., not diagonal) paths can be followed. The figure above shows two different paths for the Manhattan distance, both leading to the same result.

Euclidean distance

Following the concepts introduced above, the Euclidean distance d_e between two observations x_i and x_j can be defined as follows:

$$d_e(x_i, x_j) = \sqrt{\sum_{i=1}^{n} (x_i - y_i)^2}$$

Manhattan distance

If we want to calculate the distance between two observations x_i and x_j, the Manhattan distance $d_m$can be defined as follows:

$$d_m(x_i, x_j) = \sum_{i=1}^{n} |x_i - y_i|$$

Choosing a Value for k

There is no fixed rule for determining the optimum k value. Rather, we try to arrive at the optimal value for k iteratively, i.e., by running the algorithm a few times with different values. We can then evaluate which k value yields the best prediction results on the test dataset. It is generally understood that low k values often overfit and large ones will underfit. According to Boehmke and Greenwell (2019), at the extremes, when $k=1$, we base our prediction on a single observation that has the closest distance; when $k=n$, we are simply using all training samples as our predicted value (p. 163).

Metrics for Measuring the Prediction Performance of a Classification Model

As with regression, there are many metrics for evaluating the prediction performance of classification algorithms. We will now take a closer look at the most popular metrics widely used in practice: the confusion matrix, accuracy, recall, and precision.

Confusion matrix

The confusion matrix is a standard tool for the presentation of classification results. The following figure shows the structure of a confusion matrix consisting of four fields and using two classes to be predicted: “cat” and “dog.” The rows of the matrix show the actual number of observations in each class; the columns show how many observations the classifier has predicted in each class. It should also be noted that this assignment can be inversed, i.e., the columns could list the actual observations per class and the rows could list the predicted observations per class. A perfect classifier that only gives correct predictions would only return true positives and true negatives, i.e., observations that are correctly classified as “cat” or “dog.” However, in practice, this result would likely be considered utopian. In reality, this is a matter of increasing the number of true positives and negatives in comparison to the false positives and negatives, i.e., observations incorrectly classified as “cat” or “dog.”

Table 4

	Predicted positive (cat)	**Predicted negative (dog)**
Actual positive (cat)	True positive (TP)	False negative (FN)
Actual negative (dog)	False positive (FP)	True negative (TN)

Source: Hofer (2021).

The matrix is a decent visual representation of the classification results. That being said, it is less suitable for the comparison of different classifiers. To perform such comparisons, it is much more practical to use metrics that express model performance in the form of single digits. Then, these results could be easily compared with the scores achieved by other classification models. This is where accuracy, recall, and precision come into play, which can all be calculated from the matrix.

Accuracy

By dividing the sum of correctly classified observations by the total number of observations, we arrive at an accuracy percentage. The accuracy is calculated based on the confusion matrix, as seen here:

$$\text{Accuracy}=\frac{TP+TN}{TP+TN+FP+FN}$$

Accuracy is a metric that should be used with caution when evaluating classification models. For example, we can look at a classifier designed to predict whether any prospective airline passenger is a dangerous criminal. As the dataset is highly unbalanced, and the proportion of non-criminals passing through the airport each day could be north of 99 percent, the classifier will have high accuracy. For this reason, it is advisable to additionally consider recall and precision when working with unbalanced datasets.

Recall

Recall is the true positive rate that shows how well the model prevents false negatives (i.e., failing to identify the class as positive when appropriate). Let us take the task of classifying objects present in an image as an example scenario. Failure to classify certain objects in said image would result in a lower recall. Recall is calculated using the following notation:

$$\text{Recall}=\frac{TP}{TP+FN}$$

Precision

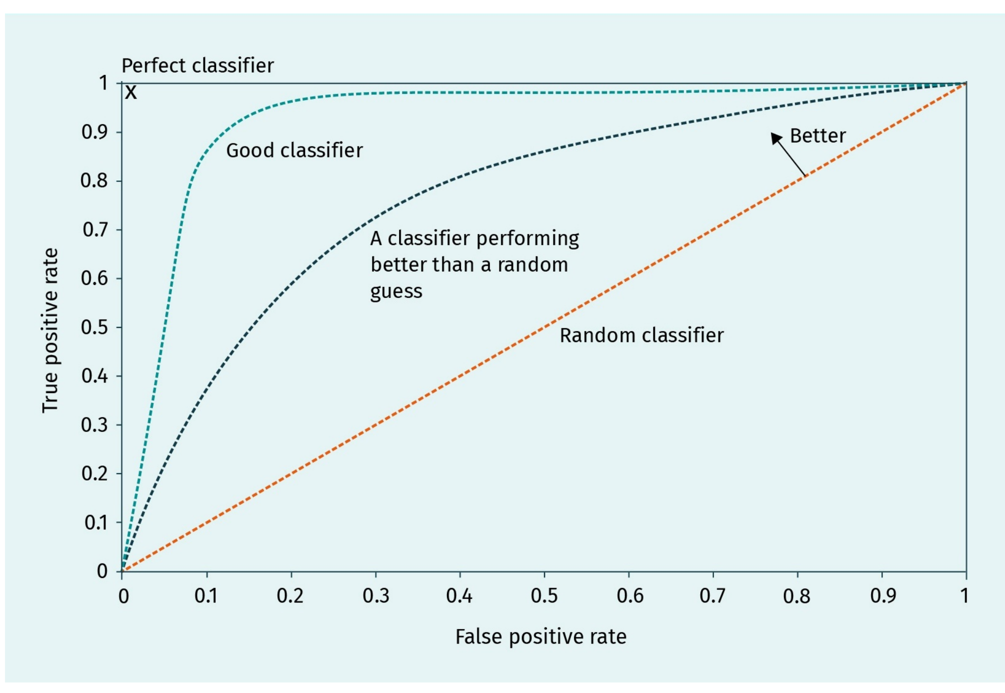
Precision refers to the accuracy of the positive predictions and indicates the reliability of a positive classification. It is typically used alongside the recall to ensure the model performs its classification tasks well. Let us briefly return to our example concerning the identification of objects in an image. If we assign an object to a specific class, the certainty and accuracy of this classification would reflect precision. Precision is expressed as follows:

$$\text{Precision}=\frac{TP}{TP+FP}$$

Receiver operating characteristic curve (ROC curve)

Another visual metric for evaluating classifiers is the receiver operating characteristic (ROC) curve. The ROC curve is the result of comparing of the true positive rate on the y-axis and the false positive rate on the x-axis. A diagonal line indicates a classifier that decides by random chance. In the figure below, we see different ROC curves for different classifiers. The closer a classifier appears to the upper left corner of the diagram, i.e., the higher its true positive rate and the lower its false positive rate, the better its prediction performance.

Figure 23: ROC Curve



Source: Hofer (2021).

Area under the curve (AUC)

Another metric that can be used to evaluate classification models is the area under the curve (AUC). As its name suggests, this metric covers the area beneath the ROC curve. The AUC is expressed as a number in the range of 0 to 1: an AUC of 0.5 describes a classifier acting randomly; an AUC of 1.0 (the maximum) represents a perfect classifier.

Figure 24: AUC Curve

Source: Hofer (2021).

Application

We will look at how the k-nearest neighbor algorithm can be used with the help of the scikit-learn library. This example uses the rather ubiquitous breast cancer dataset. First, we import the used libraries and modules.

Code

```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
```

The dataset contains two classes of tumors: 212 malignant tumors and 357 benign tumors. There are 30 features in the dataset. The label must be categorically coded so that it can be interpreted by the model (i.e., `malignant=0` and `benign=1`). For simplicity, we will only focus on three random features from those available.

Code

```
dataset = load_breast_cancer()
X = pd.DataFrame(dataset.data,columns=dataset.feature_names)
X = X[['mean smoothness', 'mean concavity', 'radius error']]
y = pd.Categorical.from_codes(dataset.target, dataset.target_names)
y = pd.get_dummies(y, drop_first=True)
```

We split the dataset into a training and a test set with the help of scikit-learns' `train_test_split` module. By default, the module takes 25 percent of the data for testing.

Code

```
_train, X_test, y_train, y_test = train_test_split(X, y)
```

We can now initialize and train an instance of the k-nearest neighbor algorithm with a value of $\mathrm{k{=}4}$. We choose the Manhattan distance as the distance metric.

Code

```
knn = KNeighborsClassifier(n_neighbors=4, +
      metric='manhattan')
knn.fit(X_train, y_train.values.ravel())
```

In the next step, we apply the trained model to the test data and generate predictions.

Code

```
y_pred = knn.predict(X_test)
```

To evaluate the model, we generate the confusion matrix.

Code

```
print(confusion_matrix(y_test, y_pred))
```

The following output will then appear in the console. One should note that, as the splitting of the data in training and testing data is random, this output may look different from execution to execution.

Code

```
[[48 4]
 [13 78]]
```

Based on the confusion matrix generated, our model shows an accuracy of $126/143 = 88.1\%$. Of course, the model could be further optimized by adding more features and by trying different values for k. Alternatively, we can use the `accuracy_score()` function from scikit learn to calculate this number.

Code

```
accuracy_score(y_test, y_pred)
```

3.2 Naïve Bayes

Inference
This is the process of using a trained machine learning model on new, incoming data.

Bayesian classifiers provide a probabilistic approach to **inference** and bases itself on "the assumption that the quantities of interest are governed by probability distributions and that optimal decisions can be made by reasoning about these probabilities together with observed data" (Mitchell, 1997, p. 154). The naïve Bayes classifier is based on Bayes' theorem, which is explained in the following section. It makes the "naïve" (yet, in practice, surprisingly performant) assumption that the distributions of the features are independent of each other.

Bayes' Theorem

We are often interested in determining the best hypothesis h from a space of different possible hypotheses H given the training data D. The best hypothesis h equates to the most likely hypothesis given the training data D and any initial information concerning the prior probabilities of the hypotheses H. $\mathrm{P(h)}$ refers to the initial probability that hypothesis h is true without having already observed the data D. It is frequently called the prior probability of h and can represent all previous knowledge we have indicating that h is true.

Analogously, $\mathrm{P(D)}$ expresses the prior probability that the data D will be observed. $\mathrm{P(D|h)}$ is the probability of observing the training data D given that hypothesis h is true. To that same effect, in machine learning problems, we are interested in the posterior probability $\mathrm{P(h|D)}$ that hypotheses h is true given the observed training data D. The Bayes theorem presents a way to compute this posterior probability $\mathrm{P(h|D)}$ based on the prior probability $\mathrm{P(h)}$ and the probabilities $\mathrm{P(D)}$ and $\mathrm{P(D|h)}$. It is defined as follows:

$$\mathrm{P(h|D)} = \frac{P(D|h)P(h)}{P(D)}$$

Based on the Bayes theorem, the naïve Bayes classifier attempts to find the most probable hypothesis h from the set of all possible hypotheses H given the training data D. The result is the maximum *aposteriori* hypothesis, i.e., the hypothesis with maximum probability (Mitchell, 1997).

Naïve Bayes Classifier

How the naïve Bayes classifier works can best be seen through an example. In this scenario, we want to predict whether a student will pass or fail an exam based on two criteria: invested learning effort and the student's performance on the last exam. Our training dataset consists of ten students.

Table 5: Exam Data

Student number	Studying effort invested	Passed prior exam (1 = yes; 0 = no)	Passed latest exam (1 = yes; 0 = no)
1	low	0	0
2	medium	1	1
3	high	1	1
4	low	0	0
5	high	1	1
6	high	1	1
7	medium	1	1
8	medium	1	0
9	high	1	1
10	medium	0	1

Source: Hofer (2021).

We derive the initial guesses for the two hypotheses from the distribution of the training data:

- h means that the student passes the upcoming exam.
- h' means that the student fails the upcoming exam.

Among the students, seven successfully passed the exam and three failed. Hence, we obtain the prior probability of passing the exam $\mathrm{P(h)}{=}0.7$ and the prior probability of failing is $\mathrm{P(h')}{=}0.3$. We can now calculate the other initial probabilities as follows:

- Studying effort is low, medium, or high, given the student passed the exam:
  $\mathrm{P(low|h)} = \#(\text{low and pass})/\#(\text{low total}) = 0/2 = 0.0$
  $\mathrm{P(medium|h)} = \#(\text{medium and pass})/\#(\text{medium total}) = 3/4 = 0.75$
  $\mathrm{P(high|h)} = \#(\text{high and pass})/\#(\text{high total}) = 4/4 = 1.0$
- Studying effort is low, medium, or high, given the student failed the exam:
  $\mathrm{P(low|h')} = \#(\text{low and not pass})/\#(\text{low total}) = 2/2 = 1.0$
  $\mathrm{P(medium|h')} = \#(\text{medium and not pass})/\#(\text{medium total}) = 1/4 = 0.25$
  $\mathrm{P(high|h')} = \#(\text{high and not pass})/\#(\text{high total}) = 0/4$
- Passed the prior exam 1/0, given the student passed the latest exam:
  $\mathrm{P(1|h)} = \#(\text{1 and pass})/\#(\text{1 total}) = 6/7 = 0.86$
  $\mathrm{P(0|h)} = \#(\text{0 and pass})/\#(\text{0 total}) = 1/3 = 0.33$
- Passed the prior exam 1/0, given the student failed the latest exam:
  $\mathrm{P(1|h')} = \#(\text{1 and not pass})/\#(\text{1 total}) = 1/7 = 0.14$
  $\mathrm{P(0|h')} = \#(\text{0 and not pass})/\#(\text{0 total}) = 2/3$

Now suppose we want to predict whether a student, who has invested little studying effort and also failed the previous exam, will pass. Using Bayes' theorem, we calculate the following posterior probabilities:

$$\mathrm{P(h|D)} = \frac{P(low|h)*P(0|h)*P(h)}{P(D)} = 0.0*0.33*0.7 = 0.0$$
$$\mathrm{P(h'|D)} = \frac{P(low|h')*P(0|h')*P(h')}{P(D)} = 1.0*0.66*0.3 = 0.2$$

Note that we can drop $\mathrm{P(D)}$, as it is constant and independent of the hypotheses. Since the posterior probability of failing the exam is larger, with $\mathrm{P(h'|D)} = 20\%$, the naïve Bayes classifier predicts that the student will indeed fail.

Application

With Gaussian naïve Bayes, we will see how perhaps the simplest naïve Bayes classifier can be used with Python. For this purpose, we will continue with our previous student example. This classifier assumes that the data from each label are drawn from a simple Gaussian distribution. We start with importing the necessary libraries.

Code
```
import pandas as pd
from sklearn.naive_bayes import GaussianNB
```

We load the training data and print it.

Code
```
exam_data = pd.read_csv('bayes_data.csv', sep=';')
print(exam_data.head(10))
```

We get the following console output after executing the print command.

Code
```
Student No. Invested effort Passed last exam Passed
0           1            low                0      0
1           2         medium                1      1
2           3           high                1      1
3           4            low                0      0
4           5           high                1      1
5           6           high                1      1
6           7         medium                1      1
7           8         medium                1      0
8           9           high                1      1
9          10         medium                0      1
```

Next, we separate the labels from the features and dummy encode the feature "Invested effort" so that it can be processed by the model.

Code

```
X = exam_data.drop(columns=['Passed'])
y = exam_data['Passed']
X = pd.get_dummies(X)
```

Now, we initialize and train an instance of the Gaussian naïve Bayes.

Code

```
model = GaussianNB()
model.fit(X, y)
```

Now we generate predictions for the following three new observations to test the prediction performance of the trained classifier.

Code

```
Student No. Invested effort Passed last exam Passed
0          11           low                 0      0
1          12        medium                 0      1
2          13          high                 1      1
```

We load and prepare the test data.

Code

```
test_data = pd.read_csv('bayes_test_data.csv', sep=';')
X_test = test_data.drop(columns=['Passed'])
y_test = test_data['Passed']
X_test = pd.get_dummies(X_test)
```

To generate and print the predictions, we execute the following commands.

Code

```
y_pred = model.predict(X_test)
print('Prediction results:')
print(y_pred)
```

We receive the following output that contains the prediction results.

Code

```
Prediction results:
[0 1 1]
```

From the output shown above, we can see that the trained model correctly predicted the passing or failing of the three students.

Difficulties of Bayesian Learning Methods

The naïve Bayes classifier and Bayesian learning methods generally have two practical difficulties of note. First, Bayesian methods require initial knowledge about the various probabilities. In practice, this can sometimes be hard to manage. If these probabilities are not known beforehand, they are often approximated based on domain, background knowledge, or assumptions made concerning the underlying distributions (i.e., just as with generalized linear models). The second practical difficulty is the considerable computational cost required to determine the optimal hypothesis (Mitchell, 1997).

SUMMARY

Classification models represent a type of supervised machine learning where the label to be predicted is a pre-determined number of categorical classes. The classification algorithm learns the characteristics of the classes provided in the training dataset and is then able to apply this knowledge to previously unseen observations. Classifiers are trained using either a lazy learner or an eager learner. Lazy learners perform only minimal training, sometimes none at all. Lazy learners usually store the training dataset until they receive the test dataset or the observation to be classified. Eager learners, which require less computation time to make a prediction, have a longer training phase than do lazy learners. The k-nearest neighbor algorithm is an example of a lazy learner. The idea behind nearest neighbor classification is to derive a prediction for one observation by using a number of similar observations from the training dataset. A Bayesian classifier, an eager learner, provides a probabilistic approach to inference and bases itself on the "naïve" assumption that the distributions of the features are independent of each other.

UNIT 4

SUPPORT VECTOR MACHINES

STUDY GOALS

On completion of this unit, you will be able to …

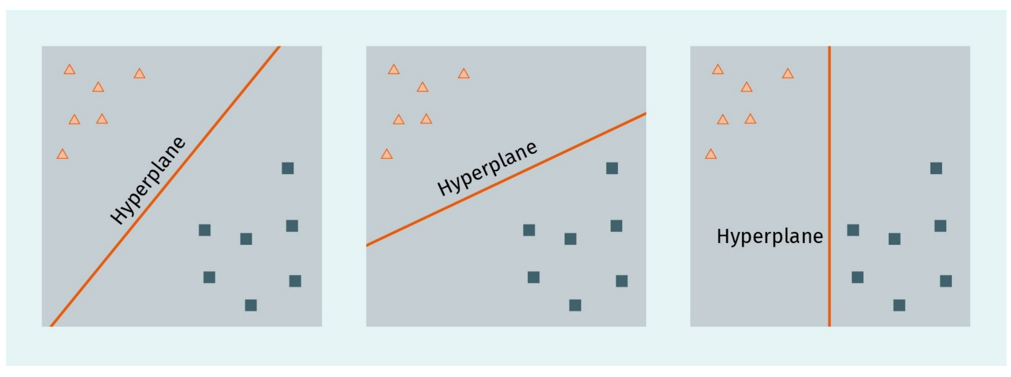
- explain the concept of large margin classification.
- conceptualize a large margin classifier with support vector machines.
- explain and make use of the kernel trick.
- apply support vector machines with the use of Python.

4. SUPPORT VECTOR MACHINES

Introduction

Support vector machines (SVM), which can perform both classification and regression tasks, are among the most popular and powerful supervised learning algorithms in existence. The idea behind SVMs is to divide the data into two classes separated by a classification boundary in an n-dimensional space called a hyperplane. A hyperplane is a subspace within a vector with one less dimension (Vapnik, 2000). New observations are assigned to one or the other class depending on which side of the hyperplane they are located. This is illustrated in the following figure.

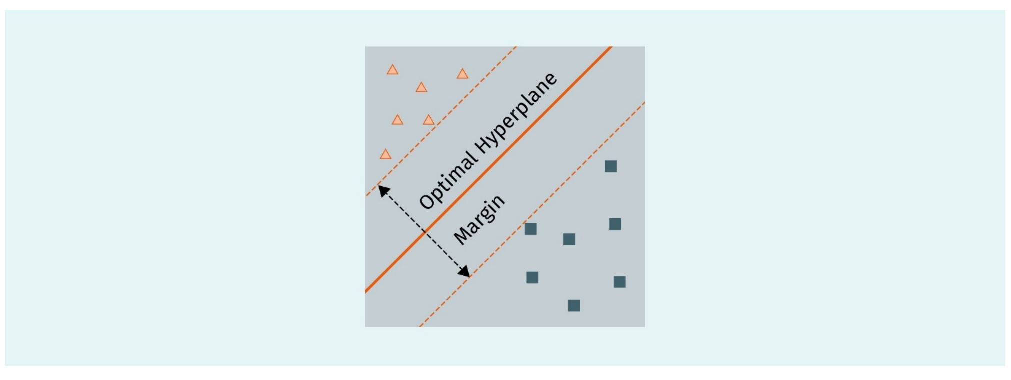
Figure 25: Different Hyperplanes to Divide the Dataset



Source: Hofer (2021).

As this figure makes visible, there is more than one way to divide the data into two classes by means of a hyperplane. SVMs understand this and, rather than opting for any hyperplane arbitrarily, they choose one where the distance between the two classes is maximized. For this reason, SVMs are also called large margin classifiers. The following figure illustrates this and provides an example of a hyperplane that separates the dataset and for which the margin between classes is at its maximum. This type of hyperplane is known as the optimal choice.

Figure 26: Different Hyperplanes to Divide the Dataset



Source: Hofer (2021).

Many linear models used for either classification or regression face some significant shortcomings, e.g., the data starting to have more overlapping features or the classes no longer being linearly separable by a linear hyperplane or decision boundary. SVMs are capable of overcoming these shortcomings and can be used for both linear and nonlinear classification. For this purpose, SVMs make use of a specific trick: the **kernel** trick. With this, data that cannot be easily separated in a lower dimensional space are mapped to higher dimensional space (where they can then be separated more easily) (Boehmke & Greenwell, 2019).

Kernel
A kernel is a class of algorithms that is used to map data into a higher dimensional space without actually performing the costly computation.

In this unit, we will analyze these SVMs, searching for answers to the following questions:

- How can we choose the best hyperplane for classification?
- How does the kernel trick, which allows SVMs to classify nonlinear data, work?
- How can we utilize SVMs with the help of Python to solve real-world problems?

4.1 Large Margin Classification

SVMs determine class boundaries and leave an object-free area as large as possible between them. With these boundaries, the data are then divided into two distinct classes. Given their characteristics, SVMs are also called large margin classifiers. Although the standard SVM algorithm is expressed as a binary classification model, it can also be used for multi-class classification. In this case, the classification problem is simply expressed in a series of binary classification models.

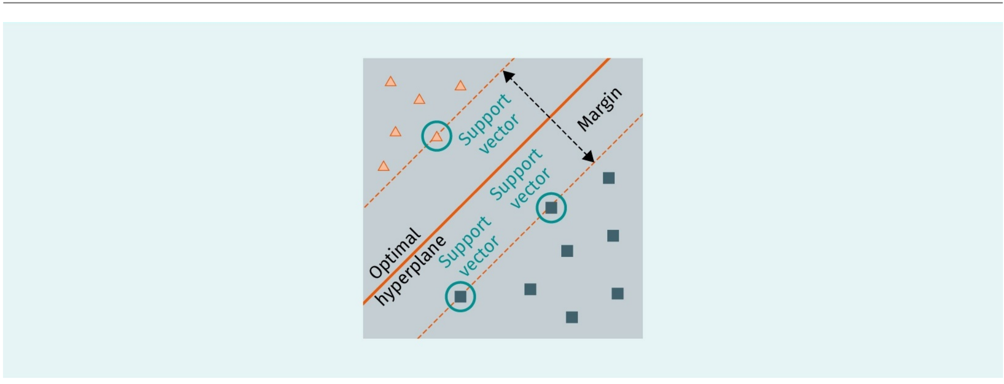
The separation of the dataset into two classes is done by a hyperplane, which is a subspace of a vector space (in this context, the feature space) with one less dimension. Thus, for a given n-dimensional vector space, a hyperplane is a subspace of the same with n-1 dimensions. The dimension of the hyperplane serving as the decision boundary (separating the data into two classes) depends on the number of features by which an observation within the data is described. This means that, if there are two features, the hyperplane is a straight line; and if there are three features, the hyperplane is a plane. This is shown in the following figure.

Figure 27: Hyperplanes in Two-Dimensional and Three-Dimensional Space

Source: Hofer (2021).

The naming of support vector machines stems from the data points that lie on the margin of a class and determine the location of the hyperplane. These are called support vectors and are shown in the figure below.

Figure 28: The Support Vectors



Source: Hofer (2021).

Depending on whether we are dealing with the classification of linearly separable classes or a nonlinear classification problem, we can distinguish between two types of support vector machines: linear and nonlinear SVMs. Linear SVMs are used for data that are linearly separable. In other words, they are used for datasets that can be classified with the decision boundary of a straight line. The SVM uses the large margin classification method to obtain the optimal decision boundary.

By comparison, nonlinear SVMs are used for data that are not linearly separable. In this scenario, the data are mapped into a higher dimensional space, where they are more easily separated. This is done by using the kernel trick and simply calculating the mathematical relationship between the points as though they were in higher dimensions. In doing this, computationally intensive transformations of the data are avoided.

Finding the Optimal Hyperplane by Maximizing the Margin

As previously described, a hyperplane in a two-dimensional space is a line and a plane in three-dimensional space. In a p-dimensional space, a hyperplane can be generally expressed as follows:

$$\beta_0 + \beta_1 X_1 + ... + \beta_p X_p$$

The prediction outcome $\mathrm{y_{en}}$ of an observation I now depends on which side of the hyperplane it is located. The goal is to find the hyperplane that maximizes the margin boundaries M, which can be formally expressed as follows:

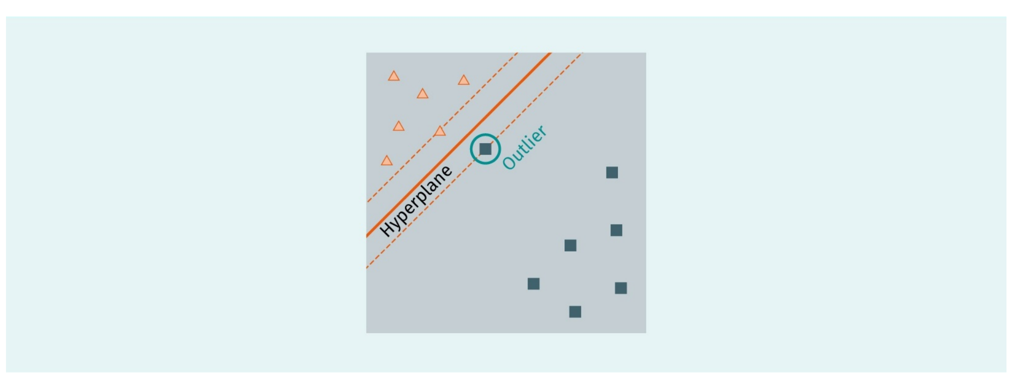
$$\begin{aligned} &\text{maximize M, given } \beta_0, \beta_1, ..., \beta_p \\ &\text{subject to} y_i\left(\beta_0 + \beta_1 x_{i1} + ... + \beta_p x_{ip}\right) \geq \mathrm{M}, \mathrm{i} = 1{,}2, ..., \mathrm{n} \end{aligned}$$

Here, $\mathrm{y_{en}}(\beta_0+\beta_1 \mathrm{x_{i1}}+\cdots+\beta_p \mathrm{x_{ip}})$ expresses the distance between the ith observation to the decision boundary. This also reveals that it must be greater than the margin (Boehmke & Greenwell, 2019).

Hard Margin and Soft Margin

By its very nature, real-world data can be quite messy, and we can almost always find a few instances where a linear classifier is unable to perfectly separate the classes. If we demand strictly that the points on the right side of the hyperplane are classified, the model will be rather sensitive to outliers, which would limit its ability to make generalizations. This hard constraint on the model is called a hard margin classification, depicted in the figure below.

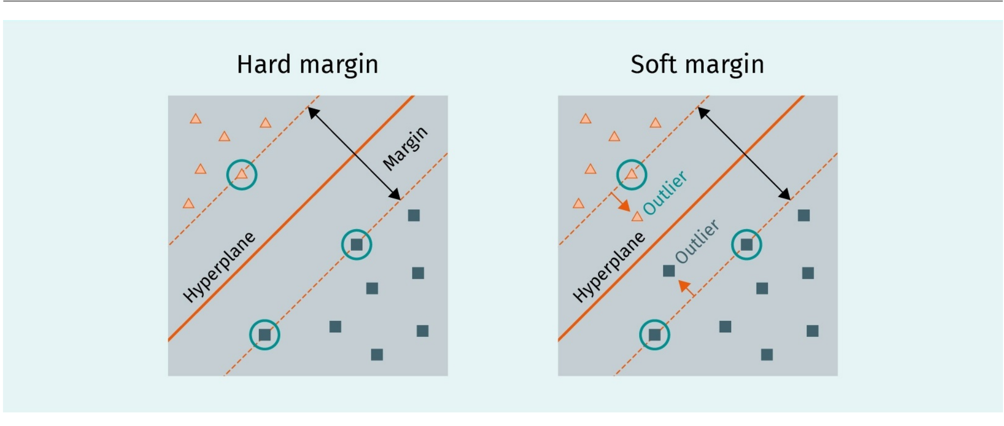
Figure 29: Sensitivity of Hard Margin Classifications to Outliers



Source: Hofer (2021).

In order to make SVMs less sensitive to outliers (thereby rendering them somewhat more flexible), we need to allow misclassifications. This adaptation is called a soft margin classification. Allowing misclassifications is a method for addressing the bias-variance-tradeoff. With the hard margin, we have a classifier that performs well on the training data and performs poorly on new, previously unseen data. This means that there is high variance. By comparison, with a soft margin, the threshold allows misclassification and thus leads to a higher bias on the training data. Nevertheless, it would also perform better on new, previously unseen data, thus resulting in lower variance. The following figure illustrates this by showing the location determination of the hyperplane by means of hard and soft margin.

Figure 30: Hard Margin and Soft Margin



Source: Hofer (2021).

Hyperparameter
A hyperparameter is a parameter with a value that helps control or govern the learning process.

To incorporate a soft margin, SVMs can be extended by adding a **hyperparameter**C that allows the algorithm to accept errors. The goal of this hyperparameter is to maximize the margin by allowing misclassifications while still accounting for outliers. The value of C is typically chosen by trial and error or cross-validation. To introduce this soft margin into the optimization function, we need to introduce a slack variable $\varepsilon_I \geq 0$ for each instance, indicating how many instances I can violate the margin. The maximization problem can then be formally expressed as follows:

$$\text{maximize M, given } \beta_0, \beta_1, \ldots, \beta_p$$
$$\text{subject to} y_i(\beta_0 + \beta_1 x_{i1} + \ldots + \beta_p x_{ip}) \geq \mathrm{M}(1 - \varepsilon_i), i = 1, 2, \ldots, n$$
$$\varepsilon_i \geq 0,$$
$$\sum_{i=1}^{n} \varepsilon_i \leq C$$

Thus, the hyperparameter C indicates the total allowed margin violation. If C is close to zero, the margin is large and many misclassifications are allowed (soft margin). If C takes a high value, the margin is narrow and misclassifications are not allowed (hard margin) (Boehmke & Greenwell, 2019).

4.2 The Kernel Trick

It is here that we must pose ourselves an important question: what happens if the data are not separable linearly? In this case, it would be impossible to find a linear hyperplane that can separate the data points. This is where the kernel trick is employed. The kernel trick sees the transformation of the non-separable data into a higher dimensional space, where they can be separated. For example, suppose we have a dataset with two given features $X_1$and X_2 that cannot be linearly separated in two-dimensional space. One solution to classify these type of data is to map it into a three-dimensional space, where the new coordinates can be specified as follows:

$$X_1 = x_1^2, X_2 = x_2^2, X_3 = \sqrt{2} x_1 x_2$$

Doing so allows the data to be separated into two classes by a hyperplane, as illustrated in the figure below.

Figure 31: The Kernel Trick

Source: Hofer (2021).

After separating the data, we can then project them and the hyperplane back onto the original two-dimensional space, as shown in the right diagram of the previous figure. However, such a transformation of the data into a higher-dimensional space is a computationally intensive task. For this reason, the kernel trick does not perform an actual transformation. Instead, it merely calculates the relation of the data points to each other as though they were in a higher-dimensional space. This can be done purely mathematically and without the need for actual transformation. In other words, the kernel trick uses algebra to save us from some spinning "loading" wheels. This mapping of the data is done with the help of a kernel function.

Kernel Functions

Kernel functions are used by SVMs to systematically find an optimal hyperplane in higher dimensions. What makes these functions so special is that they do not actually perform the transformation to higher dimensions. Instead, they directly compute the distance, i.e., the scalar products of the data points for the expanded feature representation, without ever actually computing the expansion. There are two highly common methods of mapping the data into higher dimensional space in SVMs: the polynomial kernel, which computes all possible polynomials up to a certain degree, and the radial basis function (RBF) (also called the Gaussian kernel), which corresponds to an infinite-dimensional feature space (Boehmke & Greenwell, 2019).

dth polynomial kernel

The polynomial kernel computes the decision boundary $\mathbf{K}$ via the dot product

$$X_1, X_2 \quad = \sum_{i=1}^{n} x_{1i} x_{2i}$$

of the input features $\mathbf{X_1}$ and $\mathbf{X_2}$ by raising the power of the kernel to the degree $\mathbf{d}$. Mathematically, it is defined as follows:

$$K(X_1, X_2) = (1 + \quad X_1, X_2 \quad)^d$$

The example we looked at previously is a demonstration of the polynomial kernel.

Radial basis function kernel

The radial basis function calculates the decision boundary K for the inputs X_1 and X_2 by taking the Euclidean distance

$$||X_1 - X_2||^2$$

between X_1 and $\mathrm{X}_2$and scaling it with the help of the hyperparameter determined by cross validation. Mathematically, the RBF is defined as follows:

$$K(X_1, X_2) = exp\left\{-\gamma ||X_1 - X_2||^2\right\}$$

According to Boehmke and Greenwell (2019), “the radial basis kernel is extremely flexible, and, as a rule of thumb, we generally start with this kernel when fitting SVMs in practice” (p. 279).

Application

Using the scikit-learn library, as well as the included learn package containing breast cancer data, we can see exactly how SVMs can be applied. First, we import the required libraries and modules.

Code
```
import pandas as pd
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn import svm
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
```

The dataset contains two classes of tumors: 212 malignant tumors and 357 benign tumors. There are 30 features in the dataset. We load the dataset and encode the label categorically so that it can be interpreted by the model (i.e., `malignant=0` and `benign=1`).

Code
```
dataset = load_breast_cancer()
X = pd.DataFrame(dataset.data, +
columns=dataset.feature_names)
y = pd.Categorical.from_codes(dataset.target, +
                                        dataset.target_names)
y = pd.get_dummies(y, drop_first=True)
```

We divide the dataset into the training set and the testing set, using 30 percent of the data to test the trained model. Additionally, by setting the parameter “`random_state`” to an arbitrary number (here, of course, 42), we ensure an identical split between training and test data each time, i.e., the same observations end up in the respective training and test datasets each time we run the code.

Code
```
_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, random_state=42)
```

Next, we initialize and train an instance of the SVM algorithm with a linear kernel. We do that by specifying the 'kernel' argument as 'linear'. Other valid arguments would be 'poly' for a polynomial kernel, 'rbf' for a radial basis function kernel, 'sigmoid', and 'precomputed. Using `y_train.values.ravel()`, we flatten the array to the required form $(n,)$, where n is the number of observations in the training data.

Code

```
clf = svm.SVC(kernel='linear')
clf.fit(X_train, y_train.values.ravel())
```

We can now generate predictions for the test set using the trained classifier.

Code

```
y_pred = clf.predict(X_test)
```

To evaluate how well our model performed on the test set, we should check the confusion matrix.

Code

```
print(confusion_matrix(y_test, y_pred))
```

The following confusion matrix is then output in the console.

Code

```
[[ 59   4]
 [ 2 106]]
```

According to the confusion matrix, our classifier shows 59 true positives, 106 true negatives, four false negatives, and two false positives when predicting on the test set. Therefore, the results indicate a very good prediction performance. As a final step, we can print the accuracy of our model.

Code

```
accuracy_score(y_test, y_pred)
# console output: 0.9649122807017544
```

SUMMARY

A support vector machine (SVM) is an algorithm that can perform both classification and regression tasks. They are among the most popular and powerful supervised learning algorithms in existence. The idea behind SVMs is to divide the data into two classes separated by a classification boundary in a higher dimensional space (the hyperplane).

There are two types of support vector machines: linear and nonlinear SVMs. Linear SVMs are used for datasets that can be classified with the decision boundary of a straight line. Nonlinear SVMs are used for datasets that are not linearly separable. Nonlinear SVMs map the data into a higher dimensional space, where they are more easily separated. This is done by using the kernel trick and simply calculating the mathematical

relationship between the points as though they were in higher dimensions. Therefore, SVMs use kernel functions to find the optimal hyperplane for dividing the dataset into two classes. What makes these kernel functions so special is that they do not actually perform the transformation to higher dimensions. Instead, they directly compute the distance without ever actually computing the expansion.

UNIT 5

DECISION & REGRESSION TREES

STUDY GOALS

On completion of this unit, you will be able to …

- explain the concept of decision and regression trees.
- explain the power of the ensemble methods widely used in practice.
- define bagging and boosting.
- apply two very popular ensemble models on your own with the use of Python.

5. DECISION & REGRESSION TREES

Introduction

Without a doubt, the decision tree is one of the most famous supervised learning models. Depending on the problem we need to solve, decision tress can be used for both classification (classification trees) and regression (regression trees). These trees achieve impressive predictive performance when a large number of them are bundled together, thus creating one strong estimator. Such ensemble methods, as they are commonly known, are frequently used to solve a wide variety of problems. This unit will cover decision and regression trees, as well as these powerful and widely used ensemble methods. In this unit, we will find answers to the following questions:

- How do tree-based algorithms generally work?
- How do decision trees solve classification problems?
- How do regression trees solve regression problems?
- How do ensemble methods work and how do they combine large numbers of trees to make one strong estimator?
- How can we apply random forests and gradient boosting, two very well-known ensemble methods, using Python?

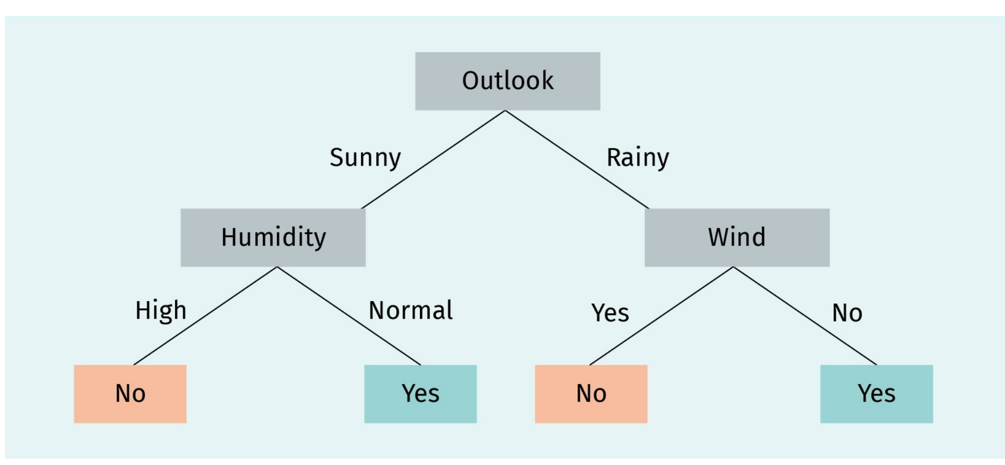
5.1 Decision & Regression Trees

The core concept behind decision and regression trees is the splitting of the dataset in a step-by-step manner based on descriptive variables. Starting from the complete dataset (i.e., the root), the data gradually branch out through the splits (i.e., the nodes) until reaching the last level of the tree (i.e., the leaves). At this point, the dataset cannot naturally be split any further, or, conversely, a stop criterion is applied. As previously mentioned, one speaks of a decision tree in the case of a classification task and a regression tree in the case of a regression task.

Decision trees make predictions about observations by sorting them along the tree, starting from the root node and ending at one of the leaf nodes. This leaf node ultimately yields the prediction. Each node on the tree is, essentially, a test case for a specific feature of the respective observation, and each branch represents a possible value of said feature. Thus, the observation receives its classification through a process comprising the following steps: First, the observation is queried at each tree level with respect to the respective feature. It then branches out to the next (lower) level, depending on the feature's value. This process is then repeated until the observation reaches one of the leaf nodes. It is at that point that the prediction is finally provided.

The following figure shows the structure of a decision tree based on a simplified example. In this example, the tree helps to make predictions concerning the likelihood that someone will go for a walk (output class means "yes") or not (output class means "no") depending on the current weather conditions.

Figure 32: Decision Tree



Source: Hofer (2021).

The prediction results of this binary classification problem are taken from the final (and lowest) level of the tree (i.e., the leaf nodes). The nodes of this tree can take the values "yes" (i.e., the person will go for a walk) or "no" (i.e., the person will not go for a walk). Based on a decision tree built in this way, each observation can be classified. For example, on a day with sunny weather and a normal level of humidity, the tree predicts that this hypothetical person will go for a walk. On a rainy, windy day, however, the tree predicts that this person will not go for a walk. It is here that the advantage of using decision trees quickly becomes apparent: their structure and functionality are easy to follow and their prediction results easy to interpret.

Regression Trees

As mentioned previously, tree-based structures can also be used on numerical features and building regression models. Numerical features become more manageable through a discretization process, i.e., by assigning threshold values. This way, these numerical features can be treated like categorical features. In this case, an observation is examined with respect to a numerical feature being less than (or equal to) or greater than (or equal to) the defined threshold value. It then moves toward the appropriate branch to reach the next level of the tree.

If the label to be predicted is a numerical variable, i.e., a regression problem must be solved, then the values that can be taken and predicted are expressed in the leaf nodes. Both cases, i.e., the handling of numerical features and the prediction of numerical values, are illustrated in the following figure.

Figure 33: Regression Tree

Source: Hofer (2021).

In comparison to the binary classification problem seen in our previous example, we now see a regression tree predicting the duration of a walk based on both weather data and the walker's age. This model predicts, for example, that a 24-year-old person will walk for 24.3 minutes on a day when it is 18° C (64° F) with 60 percent humidity.

Split Criteria

It is at this point where we must answer one essential question concerning the prediction results: How is it decided that one specific feature is placed at any given level in the decision tree? Or, more pointedly, which features are the best classifiers and, consequently, should be at the highest possible level of the tree? This decision is made by the split criterion. We will now take a look at two of the most popular and frequently used split criteria for decision trees: information gain and Gini impurity. We will then examine a split criterion designed for regression trees: the minimization of the sum of squared errors (SSE).

Information gain

Enthalpy
This concept describes the energy state of a system.

Information gain is based on the concept of entropy (not to be confused with **enthalpy**), which is a measure stemming from the field of physics describing the chaos, disorder, or diversity of states within a system. In decision trees, entropy refers to a node's impurity. Information gain measures the expected reduction in entropy (i.e., its level of impurity or disorder) and thus the effectiveness of a feature when classifying the training data (Mitchell, 1997). Expressed more formally, the information gain $\mathrm{G(S,\ A)}$ of a feature A relative to a set of observations S is given by

$$G(S,A) = \text{Entropy}\left(S\right) - \sum_{v\ \in\ \text{values}(A)} \frac{|S_v|}{|S|} \text{Entropy}\left(S_v\right)$$

where $\mathrm{values(A)}$ represent all possible values for feature A, and $\mathrm{S_v}$ is the subset of observations S, for which feature A has a value v (Mitchell, 1997). In this formal definition, the first term describes the entropy of the original dataset, and the second term describes the

entropy after the dataset has been partitioned using feature A. The latter represents the sum of entropies for each subset $\mathrm{S_v}$, weighted by the proportion of observations $|\mathrm{S_v}|/|\mathrm{S}|$ belonging to said subset (Mitchell, 1997). Using the information gain as a split criterion on each level of the tree, the feature A that minimizes the second term and thus leads to the greatest possible reduction of $\mathrm{Entropy(S)}$ is selected as the separator of the dataset on the appropriate level of the tree. Thus, starting from the root node and moving toward the leaf nodes on each level of the tree, the best classifier is selected through the information gain. Consequently, we start building the decision tree with the strongest separators. The deeper we descend through the structure of the tree, the weaker the features used on the respective levels (i.e., regarding their suitability as classifiers).

Gini impurity

The Gini impurity (GI), also referred to as the Gini index, is a measure of impurity or divergence within a dataset where a small value indicates that a node chiefly contains observations from one single class (Boehmke & Greenwell, 2019). The GI can be interpreted as the probability of a randomly chosen observation to be misclassified and is formally defined as follows (Breiman et al., 1984):

$$GI = 1 - \sum_{i\,=\,1} (p_i)^2$$

The second term here expresses the share of observations $\mathrm{p_i}$ belonging to class I for a given node. The GI for a node containing only observations of one class is, consequently, zero. Therefore, we want to minimize the GI, i.e., assign the feature that results in the lowest GI as the separator of the dataset on each level of the tree.

Sum of squared errors (SSE)

With regards to regression trees, the most common split criterion is the minimization of the sum of squared errors (SSE). If we want to split a dataset S into two subsets $\mathrm{S_1}$ and $\mathrm{S_2}$, the minimization of the SSE can be formally defined as follows:

$$SSE = \sum_{i\,\in\,S_1} (y_i - \overline{y}_1)^2 + \sum_{i\,\in\,S_2} (y_i - \overline{y}_2)^2$$

Here, $\mathrm{y_{en}}$ expresses the actual value for observation I, and $_1$ and $_2$ are the mean values of the left and right side of the possible split. Consequently, these vary with the choice of split point. The SSE (as a function of the location of the split point) is now minimized by selecting the split point that minimizes the deviations $\mathrm{y_i}$- and $\mathrm{y_i}$- of these two mean values from the actual values.

Stop Criteria

As with other estimators, trees run the risk of reducing prediction performance should too many traits of the data be included in the model (e.g., by choosing large trees with many leaves). For this reason, caution should be taken, i.e., the goal is to build an ideal-sized tree (Breiman et al., 1984). Stop criteria restrict the growth of the tree to avoid the risk of

overfitting. According to Boehmke and Greenwell (2019), there are various methods for restricting tree growth, and "two of the most common […] are to restrict the tree depth to a certain level or to restrict the minimum number of observations allowed in any terminal node" (p. 180).

Maximum depth

If we limit the maximum depth of the tree, say, to three levels, the tree will only incorporate the three features that are the most suitable separators of the data into its structure. In the following figure, we see a tree with exactly this limitation. In this example, under different circumstances, the tree would comprise six levels.

Figure 34: Limiting the Maximum Depth of a Tree to Three Levels

Source: Hofer (2021).

The more shallow the resulting tree, the less variance we have in the predictions; Boehmke and Greenwell (2019) argue that"at some point we can start to inject too much bias, as shallow trees are not able to capture complex patterns in our data" (p. 180).

Minimum number of observations in terminal nodes

An alternative (or complementary) method for preventing the tree from fully growing, and thus running into the risk of overfitting, is to limit the nodes to a minimum number of observations. This prevents a split at the point where terminal nodes would be created and results in fewer yielded observations than specified. In the following figure, the minimum number of observations in the terminal nodes is set to "2."

Figure 35: Limiting the Minimum Number of Observations in Terminal Nodes

Source: Hofer (2021).

According to Boehmke and Greenwell (2019), “a terminal node’s size of one allows a single observation to be captured in a leaf node” (p. 180). This leads to high variance and a poor level of generalizability. Conversely, large values constrain further splits, thereby reducing variance (Boehmke & Greenwell, 2019).

Tree Pruning

Another way of restricting the full growth of the tree via stop criteria is to use pruning. In other words, we allow the tree to fully develop and later remove insignificant branches. Starting at the leave nodes and moving toward the root of the tree, the branches are pruned according to the lowest level of influence on the prediction error $\mathrm{Error(T)}$ of tree T. This is done until the desired stop criterion is fulfilled, e.g., a defined maximum tree depth or a minimum number of observations per leaf node. The branching to be pruned in each pruning step, i.e., the pruning candidate C of tree T, is thus determined as follows:

$$C(T) = Error(T) + \lambda L(T)$$

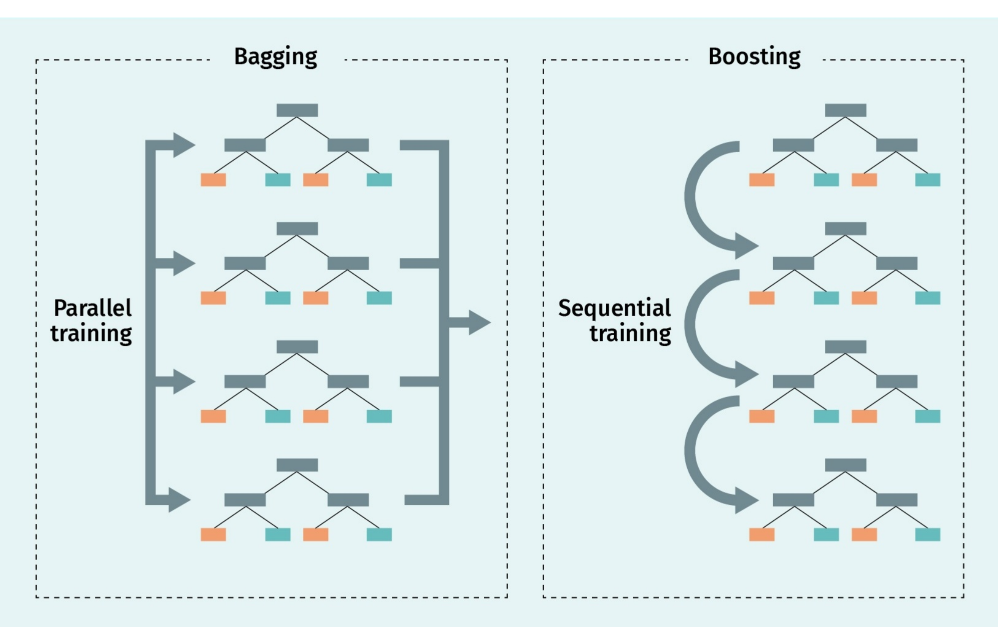
The equation operates with a pre-specified cost complexity parameter λ that penalizes the number of terminal nodes L of tree T. Smaller values for the cost complexity parameter λ tend to produce larger trees; larger values for λ result in smaller trees. We normally evaluate several models across a spectrum of λ and use cross-validation to identify the optimal value. By doing this, we arrive at the optimal subtree that is most suitable for generalizing new, unseen data (Boehmke & Greenwell, 2019).

Ensemble Methods

In practice, both decision trees and regression trees are rather commonplace. Nevertheless, they are not often used individually. Rather, it is more common for several trees (here, regarded as being weaker estimators) to be bundled together to form one strong estimator. This is where ensemble methods play a decisive role. Ensemble methods can

be divided into two categories: bagging algorithms and boosting algorithms. With bagging, the individual decision trees are independently trained in parallel. With boosting, the decision trees are trained sequentially, and one tree takes the errors of the previously constructed tree into consideration. Without a doubt, the most well-known representatives of bagging and boosting are random forest and gradient boosting, respectively. The following figure shows the two types of ensemble methods and how they generally work.

Figure 36: Bagging and Boosting

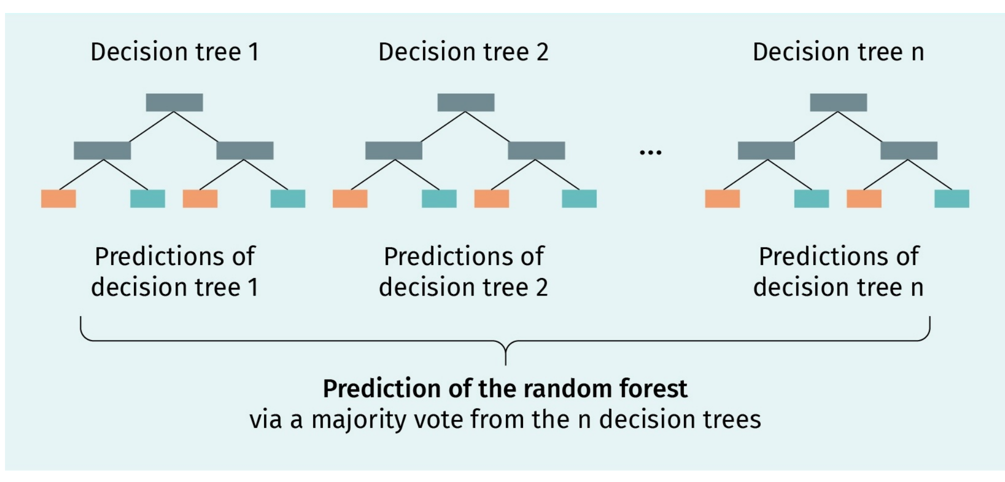


Source: Hofer (2021).

5.2 Random Forest

As a representative ensemble method, the random forest (RF) is a high performing algorithm that bundles single decision trees into one strong estimator through bagging. The final prediction of a RF is then determined by aggregating the predictions of the individual decision trees, for example, by combining them via a majority vote. This is shown in the figure below.

Figure 37: Random Forest



Source: Hofer (2021).

The individual decision trees are constructed independently of each other with the introduction of a random component. This is done by building the trees on **bootstrap** copies of the training data. Boehmke and Greenwell (2019) outline the steps of the random forest algorithm and the steps necessary to build each of the decision trees. First, select the number of trees to construct (hyperparameter “`n_trees`”). For all number of trees, complete the following set of actions:

Bootstrapping
This procedure resamples a dataset to create many simulated samples of the same size by drawing randomly with laying back.

- Generate a bootstrap sample of the original data.
- Grow a regression/classification tree based on the bootstrapped data.
- For each node/split, perform the following set of actions:
  - Select a number “`m_try`” of features at random from all p features.
  - Pick the best feature/split-point among the “`m_try`” tested features.
  - Split the node into two child nodes.
- Use common tree model stop criteria to determine when a tree is completed and unpruned.
- Output the ensemble of trees.
- Let each of the trees make a prediction.
- Use the individual predictions in a voting process in which the final prediction is determined.

Application

To illustrate how the random forest algorithm can be used as a classifier, we will use the scikit-learn library and our previous walking example. Let us assume that we have a dataset containing information concerning whether the walker has taken their walk (`Label=1`) or not (`Label=0`) for the 52 Sundays in one year. First, we import the required packages, load the dataset, and output the first five lines. Then, we can look at the information provided in the dataset.

Code
```
# Load packages
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score

# Load dataset and print 1st five rows
dataset = pd.read_csv('takingawalk_dataset.csv', sep=';')
print(dataset.head())

 Week Outlook Humidity Wind Label
    1  Rainy    High Yes     0
    2  Sunny  Normal  No     1
    3  Sunny  Normal Yes     1
    4  Sunny    High Yes     0
    5  Rainy  Normal Yes     0
```

Next, we separate the features and labels from each other. We also remove the "`Week`" feature, as it contains 52 unique values that do not add any information to the classifier. We have learned that tree-building algorithms do not need one-hot encoding—that being said, it could be useful. We encode the features and output the list of features.

Code
```
X = dataset.drop(columns=['Label', 'Week'])
y = dataset['Label']
X = pd.get_dummies(X)
print(X.columns)

Index(['Outlook_Rainy', 'Outlook_Sunny', 'Humidity_High',
      'Humidity_Normal', 'Wind_No', 'Wind_Yes'],
      dtype='object')
```

We divide the dataset into the training set and the testing set, using 30 percent of the data to test the model after training. Furthermore, we set the parameter `shuffle=True`. By doing this, we want to make sure that we are not pulling our training and testing dataset from just one season of the year. Rather, the two partitions are drawn from observations across the entire year.

Code
```
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.3, shuffle=True, random_state=42)
```

Next, we initialize and train a random forest classifier. The random forest will consist of 100 decision trees, each limited to a maximum depth of three levels. With the help of the trained classifier, we can generate predictions on the test set and output the confusion matrix for further evaluation.

Code

```
clf = RandomForestClassifier(n_estimators=100, max_depth=3, random_state=42)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print('Confusion Matrix:')
print(confusion_matrix(y_test, y_pred))

Confusion Matrix:
[[10 1]
 [0 5]]
```

The confusion matrix shows that our classifier performs very well, correctly outputting only one false negative. Additionally, we can print the accuracy of our model.

Code

```
accuracy_score(y_test, y_pred)
```

The `feature_importances_` method, a practical feature of scikit-learn's random forest implementation, shows how important each feature is for achieving the prediction quality. This allows for the output of a feature ranking contingent on the measured feature importance. The output shown below shows that for the random forest classifier, the two features `Humidity_Normal` and `Outlook_Sunny` are the most important indicators of whether the walker will, in fact, go on a walk.

Code

```
feature_scores = pd.Series(clf.feature_importances_, index=X_train.columns).sort_values(ascendi
print('Feature Scores:')
print(feature_scores)

Feature Scores:
Humidity_Normal   0.193512
Outlook_Sunny     0.192283
Humidity_High     0.168927
Wind_No           0.151238
Outlook_Rainy     0.147944
Wind_Yes          0.146097
```

5.3 Gradient Boosting

Similar to random forest, gradient boosting uses a large number of individual weak estimators (usually decision trees) that are combined into one strong estimator. In gradient boosting, however, the individual estimators are not trained in parallel and independently of each other. Rather, they are trained sequentially, and one estimator helps optimize the next. By doing so, the misclassified samples in one iteration are exaggerated in the following iteration, thereby placing emphasis on the need to classify these samples correctly in the next decision tree to be trained. Little by little, the number of misclassifications decreases.

According to Brownlee (2016), gradient boosting essentially consists of three elements:

1. A loss function we want to optimize
2. A set of weak learners generating predictions
3. An additive model for combining the predictions of the weak learners into one strong predictor (thereby minimizing the loss function)

In the figure below, we see exactly the way the gradient boosting algorithm works and how these elements interact.

Figure 38: Gradient Boosting

Source: Hofer (2021).

Boehmke and Greenwell (2019) formulate the steps of the gradient boosting algorithm as follows:

- Select the number of trees to construct (hyperparameter “`n_trees`”).
- Fit the first weak estimator to the given training data X and generate predictions $\hat{y}$, i.e., $F_1(x)=\hat{y}$
- Fit the next weak estimator to the residuals of the previous one, i.e, $h_1(x)=y-F_1(x)$
- Add this weak estimator to the model, i.e., $F_2(x)=F_1(x)+h_1(x)$
- Fit the next weak estimator to the residuals of F_2: $h_2(x)=y-F_2(x)$

- Add this weak estimator to the model, i.e., $F_3(x)=F_2(x)+h_1(x)$
- Continue this process until the number of prospective trees has been reached.

The resulting strong estimator can be mathematically expressed as the additive combination of the single i weak estimators of number n:

$$f(x) = \sum_{i=1}^{n} f^i(x)$$

Gradient boosting is considered a gradient descent algorithm, which is a very general optimization algorithm able to find the optimal solutions to a broad variety of problems. The general idea behind gradient descent is to iteratively change parameters in order to minimize a loss function. To that effect, gradient boosting is very flexible regarding the loss function used (Boehmke & Greenwell, 2019).

Application

We can use a gradient boosting classifier from the scikit-learn library almost exactly in the way as we would use a random forest classifier. In fact, there is only one notable code change from the random forest example. Specifically, the import and initialization commands of the model are altered, and we write these as follows:

Code

```
from sklearn.ensemble import GradientBoostingClassifier
clf = GradientBoostingClassifier(n_estimators=100,
max_depth=3, random_state=42
```

We evaluate the resulting prediction performance using the confusion matrix.

Code

```
[[10 1]
 [ 0 5]]
```

Thus, according to the confusion matrix, the gradient boosting classifier achieves the same performance on the test set as does the random forest classifier. However, there are some notable differences regarding the model's feature importance determinations. These can be displayed analogously by using the `feature_importances_ method`. For the gradient boosting classifier, the two features`Outlook_Sunny` and `Wind_No` are by far the most important indicators of whether the walker will, in fact, go on a walk.

Code

```
Feature Scores:
Outlook_Sunny     0.247713
Wind_No           0.237722
Wind_Yes          0.144501
```

```
Humidity_High      0.132627
Outlook_Rainy      0.127789
Humidity_Normal    0.109648
```

SUMMARY

Decision tress can be used for both classification and regression. The core concept behind decision and regression trees is the splitting of a dataset in a step-by-step manner based on descriptive features. Starting from the complete dataset (i.e., the root), the data gradually branch out through the splits (i.e., the nodes) until reaching the last level of the tree (i.e., the leaves).

A prediction about an observation is derived by sorting it along the tree, starting from the root node and ending at one of the leaf nodes. Finally, the prediction is obtained from this leaf node. In practice, decision trees and regression trees are not often used individually. It is more common to bundle many decision or regression trees together to form one strong estimator. This practice reflects the ensemble methods, which are divided into two fundamentally different types: bagging algorithms and boosting algorithms. With bagging, the individual decision trees are independently trained in parallel. With boosting, the decision trees are trained sequentially, and one tree takes the errors of the previously constructed tree into consideration.

BACKMATTER

LIST OF REFERENCES

Bishop, C. M. (2006). *Pattern recognition and machine learning*. Springer.

Boehmke, B., & Greenwell, B. (2019). *Hands-on machine learning with R*. Chapman & Hall.

Breiman, L., Friedman, J., Olshen, R. A., & Stone, J. S. (1984). *Classification and regression trees*. Chapman & Hall. https://doi.org/10.1201/9781315139470

Brownlee, J. (2016, September 9). *A gentle introduction to the gradient boosting algorithm for machine learning*. Machine Learning Mastery. https://machinelearningmastery.com/gentle-introduction-gradient-boosting-algorithm-machine-learning/

Funmilola, A., Olusayo, F., & Elizabeth, A. (2015). Classification of abnormalities in brain MRI images using PCA and SVM. *International Journal of Engineering Research and Applications*, *5*(7), 56—62.

Géron, A. (2019). *Hands-on machine learning with scikit-learn, Keras & Tensorflow* (2nd ed.). O'Reilly.

Hardin, J. W., & Hilbe, J. M. (2007). *Generalized linear models and extensions* (2nd ed.). Stata Press.

Hashan, A. M. (2021a). *MRI based brain tumor images, version 1* (Normal (1).jpg) [Photograph]. https://www.kaggle.com/mhantor/mri-based-brain-tumor-images

Hashan, A. M. (2021b). *MRI based brain tumor images, version 1* (Tumor (118).jpg) [Photograph]. https://www.kaggle.com/mhantor/mri-based-brain-tumor-images

Hingane, M. C., Matkar, S. B., Mane, A. B., & Shirsat, A. M. (2015). Classification of abnormalities in brain MRI images using SVM classifier. *International Journal of Science Technology & Engineering*, *1*(9), 24—28.

Jensen, K. (2012). *CRISP-DM process diagram* (CRISP-DM_process_diagram.png) [Illustration]. Wikimedia Commons. https://commons.wikimedia.org/wiki/File:CRISP-DM_Process_Diagram.png

Kida, Y. (2019). *Generalized linear models: Introduction to advanced statistical modeling*. Towards Data Science. https://towardsdatascience.com/generalized-linear-models-9cbf848bb8ab

McCulloch, C. E., & Searle, S. R. (2001). *Generalized, linear, and mixed models* . John Wiley & Sons.

Mitchell, T. M. (1997). *Machine learning.* McGraw-Hill.

Ray, S. (2017). *6 easy steps to learn naïve Bayes algorithm with codes in Python and R*. Analytics Vidhya. https://www.analyticsvidhya.com/blog/2017/09/naive-bayes-explained/

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, *44*(1.2), 206—226.

Singh, D., & Kaur, K. (2012). Classification of abnormalities in brain MRI images using GLCM, PCA, and SVM. *International Journal of Engineering and Advanced Technology*, *1*(6), 243—248.

Vapnik, V. (2000). *The nature of statistical learning theory*. Springer.

Webb, A. (2002). *Statistical pattern recognition*. John Wiley & Sons. https://doi.org/10.1002/0470854774

Wirth, R., & Hipp, J. (2000). CRISP-DM: Towards a standard process model for data mining. *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining* (pp. 29—39). Springer.

LIST OF TABLES AND FIGURES

Figure 1: Sample of Normal and Abnormal Brain Images . . . 14

Figure 2: Brain MRI Image Classification . . . 15

Figure 3: The Six Phases of the CRISP-DM Process Model . . . 17

Figure 4: The Steps of the Machine Learning Cycle . . . 20

Figure 5: Ten-Fold Cross Validation . . . 21

Figure 6: Gradient Descent . . . 22

Figure 7: Learning Rates . . . 23

Figure 8: Overfitting and Underfitting in Regression . . . 24

Figure 9: Overfitting and Underfitting in Classification . . . 25

Table 1: Important Regression Terms and Their Synonyms . . . 28

Figure 10: Linear Regression Model . . . 29

Figure 11: Plot of Residuals . . . 30

Figure 12: Illustration of Mean Absolute Error . . . 31

Figure 13: Illustration of Mean Squared Error . . . 33

Figure 14: Linear Regression Plot Generated by the Matplotlib Library . . . 35

Table 2: Coefficients of Features “YearsExperience” and “YearsExp/100” . . . 38

Figure 15: Number of Defective Products . . . 40

Figure 16: The Poisson Distribution with Different Values for η . . . 41

Figure 17: Poisson Regression Applied to Our Data . . . 42

Table 3: Some Probability Distributions and Their Link Functions . . . 43

Figure 18: Logistic Regression 44

Figure 19: Logistic Regression as a Binary Classifier 45

Figure 20: Plot of Estimated Probabilities with Decision Boundary 48

Figure 21: K-Nearest Neighbor Algorithm with Two Classes and k=3, k=6, and k=9 54

Figure 22: Calculation of Euclidean Distance and Manhattan Distance 56

Table 4 58

Figure 23: ROC Curve 59

Figure 24: AUC Curve 60

Table 5: Exam Data 63

Figure 25: Different Hyperplanes to Divide the Dataset 68

Figure 26: Different Hyperplanes to Divide the Dataset 69

Figure 27: Hyperplanes in Two-Dimensional and Three-Dimensional Space 70

Figure 28: The Support Vectors 70

Figure 29: Sensitivity of Hard Margin Classifications to Outliers 72

Figure 30: Hard Margin and Soft Margin 72

Figure 31: The Kernel Trick 73

Figure 32: Decision Tree 81

Figure 33: Regression Tree 82

Figure 34: Limiting the Maximum Depth of a Tree to Three Levels 84

Figure 35: Limiting the Minimum Number of Observations in Terminal Nodes 85

Figure 36: Bagging and Boosting 86

Figure 37: Random Forest 87

Figure 38: Gradient Boosting 90

IU Internationale Hochschule GmbH
IU International University of Applied Sciences
Juri-Gagarin-Ring 152
D-99084 Erfurt

Mailing Address
Albert-Proeller-Straße 15-19
D-86675 Buchdorf

media@iu.org
www.iu.org

Help & Contacts (FAQ)
On myCampus you can always find answers
to questions concerning your studies.