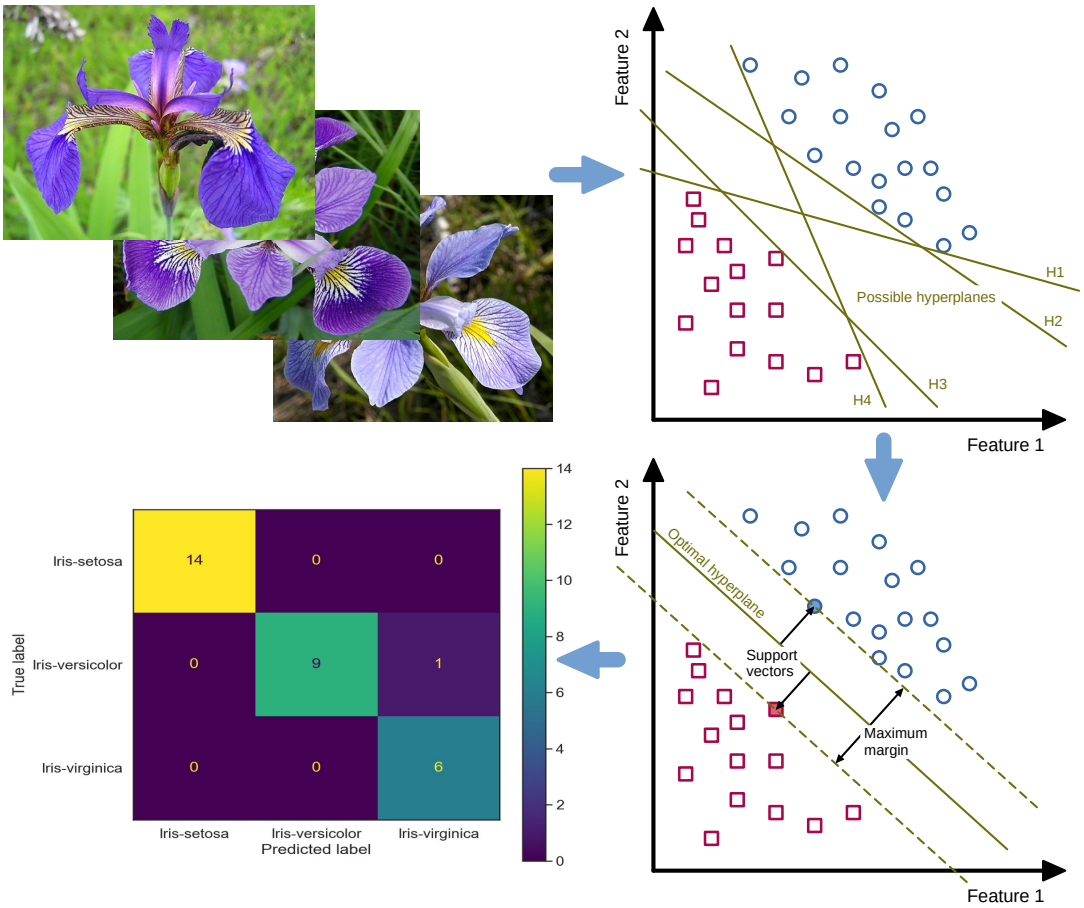
Getting started with Machine Learning (ML) and Support Vector Classifiers (SVC) - A systematic step-by-step approach

Dipl.-Ing. Björn Kasper (kasper.bjoern@bgetem.de)

Test and Certification Body for Electrical Engineering at BG ETEM

October 25, 2022; version 0.8 (pre-release)



Anyone who wants to seriously deal with the emerging topic of our time "Artificial Intelligence (AI)" cannot avoid dealing with the basic mathematical models and algorithms from the field of "Machine Learning (ML)" as a subset of AI. However, someone who opens the door for the first time to this equally very exciting as well as arbitrarily complex and, at first glance, confusing world will very quickly be overwhelmed. Here, it is a good idea to consult introductory and systematic tutorials. Therefore, this Getting Started tutorial systematically demonstrates the typical ML work process step-by-step using the very powerful and performant "Support Vector Classifier (SVC)" and the widely known and exceptionally beginner-friendly "Iris Dataset". Furthermore, the selection of the "correct" SVC kernel and its parameters are described and their effects on the classification result are shown.

Contents

1 Introduction

1.1 English introduction

In the **digitized work environment**, there is an increasing demand for **Work equipment** to be able to adapt independently and in a task-related manner to changing work situations. Depending on the strength of the degree of flexibility, this **situational adaptivity** can often only be realized by applying mathematical models and algorithms from the field of **Machine Learning (ML)** as a subset of **Artificial Intelligence (AI)**.

Examples of such AI applications in work environments can range from comparatively simple **voice assistance systems** (similar, for example, to Siri or Alexa from the private sphere) to partially or **highly automated systems**. The transition from **automation to autonomy** is currently the subject of much controversy among experts and can be viewed in terms of the transition of responsibility from humans to technical systems (Adler 2021; Adler 2019).

By definition, a system is called **autonomous** only when it can achieve a given goal **independently**

and adapted to the situation **without human control** or detailed **programming** (Dumitrescu et al. 2018; Kadner et al. 2017).

However, the distinction between the degree of automation and the autonomy of a technical system is relatively vague and difficult to define, depending on the technical context and the degree of abstraction. Crucial for the classification are the degrees of **self-determination**, **independence** as well as the **freedom of decision or action** of a technical system towards **human intervention** or preprogrammed behavior patterns (vgl. Wikipedia: Autonomie 2022).

In contrast to highly automated systems, autonomous systems are only able to act autonomously, solve problems, and learn to constantly improve in the process through the use of AI algorithms (Kadner et al. 2017).

For example, **driverless transport systems (AGVs)** can navigate **autonomously** through larger industrial facilities using self-learned self-updated maps shared with other AGVs, and avoid location-changing obstacles by independently finding and optimizing suitable routes. However, at a higher level of abstraction, new logistics tasks are given to them by human operators, which is why AGVs tend to be **highly automated systems** from a human perspective.

In addition to the many very interesting advantages, e.g. in terms of economic efficiency and workload reduction, such highly automated systems and, depending on the point of view, autonomous subsystems are characterized by a very high level of technical complexity. This concerns both their **operating functions** (e.g. autonomous navigation through complex industrial environments with shared use of the roadways by other human-controlled vehicles) and their **safety functions** (e.g. evaluation of interlinked imaging and non-imaging safety sensors for monitoring the driving space to avoid collisions).

Very high requirements are placed on such autonomous systems and the AI algorithms used for this purpose with regard to **functional safety**. However, the requirements for safety evaluability in terms of **transparency** (complete understanding of the system) and **explainability** of decisions made by AI are currently very difficult or impossible to achieve, especially when using AI algorithms from the field of **deep learning** (Kuhn and Liggesmeyer 2019).

Unlike automated systems, the functionality of AI-powered autonomous systems is not fully programmed out before operational use, but is created by applying algorithms with learning capabilities to data. This results in a model that is merely executed by the software at runtime. Due to its **inherent complexity**, the resulting model is generally **not comprehensible** to humans, which means that the **decisions** of an AI system are often **not transparent**. Although the requirements for the AI system typically cannot be fully described, it must still function reliably later at runtime in a very large application space (Heidrich, Schneider, and Jedlitschka 2021). This pushes today's established methods and techniques of systematic software design and testing of safety-related software to their limits (cf. **V model** according to DIN EN 61508-3 2011).

Furthermore, in terms of their **recognition rates** and thus the **reliability of their decisions**, today's AI algorithms very often do not meet the functional safety requirements to achieve higher safety levels, even under the most favorable conditions. For example, a software-based safety function with a performance level d (PL_d) typically required for machines in accordance with ISO 13849-1 may only fail dangerously with a probability of $10^{-7} - 10^{-6}$ per hour during continuous use (see table K.1 in DIN EN ISO 13849-1 2016).

Compared to traditional, fully programmed software, the relatively low robustness of data-driven algorithms from the field of deep learning is another challenge. This can cause **small changes** in the function-determining **training data** to cause **large and unpredictable changes** in system behavior under some circumstances. However, the **predictability** and **transparency** of the system behavior are elementary for a **safety verification** (Jürgensohn et al. 2021).

An appropriate assessment or even **testing** with regard to the required functional safety according to uniform and ideally standardized criteria has numerous consequences for the future orientation and organization of technical **occupational safety and health (OSH)** in Germany and in Europe. In addition to the currently still very difficult safety-related assessability, an important point is that the previous clear separation between **placing on the market law** (see e.g. Machinery Directive) and **occupational safety and health law** (see European Framework Directive for Occupational Safety and Health and German Ordinance on Occupational Safety and Health) can no longer be continued in this way. The reason for this is that **safety-related properties** will also change, especially of systems **continuously**

learning at runtime, due to new or **adapted behaviors** learned during operation (Jürgensohn et al. 2021). From today's point of view, systems based on **learned-out** and at runtime **invariable models** are not affected by this.

For these reasons, especially the actors of **technical occupational safety and health** who will deal with the **evaluation** of such **systems capable of learning** or system components with AI algorithms in the future should familiarize themselves in depth with the software structures used for this purpose as early as possible. This is the only way to ensure that the rapid development of systems capable of learning can be accompanied by OSH and their testing authorities in a constructive, critical and technically appropriate manner. If this is omitted, it must be assumed on the basis of the experiences of recent years that the OSH system will be ruthlessly circumvented or undermined by the economic interests of globally operating software giants. This would have the consequence that serious or fatal **occupational accidents** are more likely to occur **due to inadequately designed AI-based work systems**.

However, the safety-related evaluation of such learning-capable systems requires a more in-depth technical entry into the world of **machine learning** as a subfield of **artificial intelligence**. For this purpose, it is necessary to deal with the basic operation of typical ML algorithms, corresponding software tools, libraries and programming systems.

However, someone who opens the door for the first time to this equally very exciting as well as arbitrarily complex and, at first glance, confusing world will very quickly be overwhelmed. In addition to reading general technical literature, it is advisable to consult introductory and systematic tutorials.

This Getting Started tutorial has exactly this goal, demonstrating systematically and step-by-step the typical ML workflow using the very powerful **Support Vector Classifier (SVC)** as an example.

This tutorial will be presented in the context of a workshop at the **Conference "Artificial Intelligence"**, hosted by the German Social Accident Insurance (DGUV), probably in November 2022 in Dresden. The workshop addresses interested ML novices in the technical occupational safety and health of the social accident insurance institutions.

Besides the **deep neural networks**, which are very present in the media, there is a very rich diversity of other very powerful ML algorithms - suitable for the particular use case. For a more generally comprehensible introduction, the SVC algorithm was deliberately chosen for the target audience of the workshop. Its operating principles are easy to convey to ML novices as well as in the time frame given for the workshop - quite in contrast to the entry into the world of deep neural networks.

The following main sections will demonstrate the typical ML workflow step-by-step. In **step 0**, specific guidance is provided for selecting hardware and software suitable for machine learning. To allow an ML novice to first familiarize themselves with the ML algorithms, tools, libraries, and programming systems, the ready-made and very beginner-friendly **Iris dataset** is involved in **step 1**. Only after a comprehensive acquaintance with the application of ML tools would it make sense to examine one's own environment for ML-suitable applications and to obtain suitable datasets from them. However, this is beyond the scope of this introductory tutorial.

One of the most important steps in the entire ML process is **step 2**, in which the dataset included in step 1 is examined using typical data analysis tools. In addition to exploring the **data structure** and **internal correlations** in the dataset, errors such as gaps, duplications, or obvious misentries must also be found and corrected where possible. This is enormously important so that the classification can later provide plausible results.

After exploring the dataset, in **step 3** one has to decide on a specific ML algorithm based on certain selection criteria. Among other ML algorithms suitable for the Iris dataset (such as the decision-tree-based **random-forests classifier**), the reasoned choice here in the tutorial falls on the **support vector classifier**. A dedicated SVC model is now being implemented.

In **step 4** the dataset is preprocessed for the actual classification by SVC. Depending on the selected ML algorithm as well as the data structure, it may be necessary to prepare the data before training (e.g., by standardization or normalization). After splitting the dataset into a training and test dataset, the SVC model is trained with the training dataset in **step 5**. Subsequently, classification predictions are made with the trained SVC model based on the test data. In **step 6**, the quality of the classification result is evaluated using known **metrics** such as the **confusion matrix**.

Since the classification in step 5 was initially performed with standard parameters (so-called **hyper-parameters**), their meaning is explained in **step 7** and then their effect on the classification result is demonstrated by manually varying the individual hyper-parameters.

In the final **step 8**, two approaches to systematic hyper-parameter search are presented: **Grid Search** and **Randomized Search**. While the former exhaustively considers all parameter combinations for given values, the latter selects a number of candidates from a parameter space with a particular random distribution.

1.2 German introduction

Von den **Arbeitsmitteln** in der **digitalisierten Arbeitswelt** wird immer stärker gefordert, dass sie sich selbstständig und aufgabenbezogen an sich ändernde Arbeitssituationen anpassen können. Diese **situative Adaptivität** kann je nach Stärke des Flexibilisierungsgrades oft nur durch die Anwendung mathematischer Modelle und Algorithmen aus dem Bereich des **Maschinellen Lernens (ML)** als Teilmenge der **Künstlichen Intelligenz (KI)** realisiert werden.

Beispiele für solche KI-Anwendungen in der Arbeitswelt reichen von vergleichsweise einfachen **Sprachassistenzsystemen** (ähnlich z. B. Siri oder Alexa aus dem privaten Umfeld) bis hin zu teil- oder **hochautomatisierten Systemen**. Der Übergang von **Automatisierung zu Autonomie** wird derzeit in der Fachwelt sehr kontrovers diskutiert und kann unter dem Aspekt des Übergangs der Verantwortung vom Menschen zum technischen System betrachtet werden (Adler 2021; Adler 2019).

Definitionsgemäß wird ein System erst dann als **autonom** bezeichnet, wenn es **ohne menschliche Steuerung** oder detaillierte **Programmierung** ein vorgegebenes Ziel **selbstständig** und an die Situation angepasst erreichen kann (Dumitrescu et al. 2018; Kadner et al. 2017).

Allerdings ist die Unterscheidung des Grades der Automatisierung bis hin zur Autonomie eines technischen Systems relativ fließend und je nach fachlichem Kontext und Abstraktionsgrad nur schwer zu definieren. Maßgeblich für die Einordnung sind die Grade der **Selbstbestimmtheit**, die **Unabhängigkeit** sowie die **Entscheidungs- bzw. Handlungsfreiheit** eines technischen Systems gegenüber **menschlichem Eingriff** oder vorprogrammierter Verhaltensmuster (vgl. Wikipedia: Autonomie 2022).

Im Gegensatz zu hochautomatisierten Systemen sind autonome Systeme nur durch Einsatz von KI-Algorithmen in der Lage, eigenständig zu agieren, Probleme zu lösen und dabei zu lernen, sich ständig zu verbessern (Kadner et al. 2017).

Beispielsweise können **fahrerlose Transportsysteme (FTS)** anhand selbst erlernter, selbstständig aktualisierter und mit anderen FTS geteilter Karten **autonom** durch größere Industrieanlagen navigieren und ortsveränderlichen Hindernissen ausweichen, indem sie selbstständig geeignete Routen finden und optimieren. Jedoch werden ihnen in einer höheren Abstraktionsebene neue Logistikaufträge durch menschliche Bediener vorgegeben, weswegen es sich bei FTS aus menschlicher Perspektive eher um **hochautomatisierte Systeme** handelt.

Neben den vielen sehr interessanten Vorteilen z. B. bzgl. Wirtschaftlichkeit und Arbeitserleichterung kennzeichnet solche hochautomatisierten und je nach Betrachtung autonomen Teilsysteme eine sehr hohe technische Komplexität. Diese betrifft sowohl ihre **Betriebsfunktionen** (z. B. autonome Navigation durch komplexe industrielle Umgebungen bei gemeinsamer Nutzung der Fahrwege durch andere menschlich gesteuerte Fahrzeuge) als auch ihre **Sicherheitsfunktionen** (z. B. Auswertung miteinander verknüpfter bildgebender und nicht-bildgebender Sicherheitssensorik zur Überwachung des Fahrraums zur Kollisionsvermeidung).

An solche autonomen Systeme und die hierfür eingesetzten KI-Algorithmen werden sehr hohe Anforderungen hinsichtlich der **funktionalen Sicherheit** gestellt. Jedoch sind die Anforderungen für eine sicherheitstechnische Bewertbarkeit bezüglich der **Transparenz** (vollständiges Systemverständnis) und **Erklärbarkeit** der durch KI getroffenen Entscheidungen insbesondere bei Einsatz von KI-Algorithmen aus dem Bereich des **Deep Learnings** derzeit nur sehr schwer oder gar nicht erreichbar (Kuhn and Liggesmeyer 2019).

Im Gegensatz zu automatisierten Systemen wird die Funktionalität KI-gestützter autonomer Systeme nicht vor der betrieblichen Verwendung vollständig ausprogrammiert, sondern durch das Anwenden lernfähiger Algorithmen auf Daten erstellt. Dadurch entsteht ein Modell, das von der Software zur Laufzeit

lediglich ausgeführt wird. Das resultierende Modell ist aufgrund seiner **inhärenten Komplexität** im Allgemeinen **für den Menschen nicht verständlich**, wodurch die **Entscheidungen** eines KI-Systems oft **nicht transparent** sind. Obwohl die Anforderungen an das KI-System typischerweise nicht vollständig beschrieben werden können, muss es später zur Laufzeit in einem sehr großen Anwendungsraum trotzdem verlässlich funktionieren (Heidrich, Schneider, and Jedlitschka 2021). Dadurch kommen die heute etablierten Methoden und Techniken des systematischen Softwareentwurfes und -testens sicherheitsgerichteter Software an ihre Grenzen (vgl. **V-Modell** nach DIN EN 61508-3 2011).

Weiterhin erfüllen heutige KI-Algorithmen hinsichtlich ihrer erreichbaren **Erkennungsraten** und damit der **Zuverlässigkeiten ihrer Entscheidungen** selbst unter günstigsten Bedingungen sehr oft nicht die Anforderungen an die funktionale Sicherheit, um höhere Safety-Level zu erreichen. Beispielsweise darf eine software-gestützte Sicherheitsfunktion mit einem für Maschinen typischerweise geforderten Performance Level d (PL_d) nach ISO 13849-1 bei kontinuierlicher Nutzung nur mit einer Wahrscheinlichkeit von $10^{-7} - 10^{-6}$ pro Stunde gefährlich ausfallen (siehe Tabelle K.1 in DIN EN ISO 13849-1 2016).

Im Vergleich zu traditioneller, vollständig ausprogrammierter Software ist bei datengetriebenen Algorithmen aus dem Bereich des Deep Learnings die verhältnismäßig geringe Robustheit eine weitere Herausforderung. Diese kann dazu führen, dass **kleine Änderungen** in den funktionsbestimmenden **Trainingsdaten** unter Umständen **große und unvorhersehbare Veränderungen** des Systemverhaltens bewirken. Jedoch sind die **Vorhersehbarkeit** und **Nachvollziehbarkeit** des Systemverhaltens für einen **Sicherheitsnachweis** elementar (Jürgensohn et al. 2021).

Eine hinsichtlich der geforderten funktionalen Sicherheit angemessene Bewertung oder gar **Prüfung** nach einheitlichen und idealerweise genormten Maßstäben hat viele Konsequenzen für die zukünftige Ausrichtung und Gestaltung des **technischen Arbeitsschutzes** in Deutschland und in Europa. Neben der derzeit noch sehr schwierigen sicherheitstechnischen Bewertbarkeit von KI-Algorithmen ist ein wichtiger Punkt, dass die bisherige klare Trennung zwischen **Inverkehrbringensrecht** (siehe z. B. Maschinenrichtlinie) und **betrieblichem Arbeitsschutzrecht** (siehe Arbeitsschutz-Rahmenrichtlinie und Betriebssicherheitsverordnung) so nicht mehr aufrechterhalten werden kann. Grund hierfür ist, dass sich auch die **sicherheitsrelevanten Eigenschaften** insbesondere von zur Laufzeit **weiterlernenden Systemen** durch während des Betriebs erlernte, neue oder **angepasste Verhaltensweisen** verändern werden (Jürgensohn et al. 2021). Systeme auf Basis **ausgelernter** und zur Laufzeit **unveränderlicher Modelle** sind aus heutiger Sicht hiervon nicht betroffen.

Aus diesen Gründen sollten sich insbesondere die Akteure des **technischen Arbeitsschutzes**, die sich zukünftig mit der **Prüfung** solcher **lernfähigen Systeme** oder Systemkomponenten mit KI-Algorithmen befassen werden, möglichst frühzeitig mit den hierfür eingesetzten Software-Strukturen vertieft auseinandersetzen. Nur dadurch lässt sich erreichen, dass die stürmische Entwicklung lernfähiger Systeme durch den Arbeitsschutz und dessen Prüfinstitute konstruktiv, kritisch und fachlich angemessen begleitet werden kann. Wird dies versäumt, muss aufgrund der Erfahrungen der vergangenen Jahre davon ausgegangen werden, dass das Arbeitsschutzsystem durch die wirtschaftlichen Interessen global agierender Softwaregiganten skrupellos umgangen oder ausgehebelt werden wird. Dies hätte die Folge, dass schwere oder tödliche **Arbeitsunfälle wegen unzulänglich gestalteter KI-basierter Arbeitssysteme** wahrscheinlicher werden.

Allerdings erfordert die sicherheitstechnische Bewertung solcher lernfähigen Systeme einen tiefer gehenden fachlichen Einstieg in die Welt des **maschinellen Lernens** als Teilgebiet der **künstlichen Intelligenz**. Hierzu muss sich mit den grundlegenden Funktionsweisen typischer ML-Algorithmen, entsprechenden Software-Werkzeugen, Bibliotheken und Programmiersystemen auseinander gesetzt werden.

Wer jedoch zum ersten Mal die Tür zu dieser ebenso spannenden wie beliebig komplexen und auf den ersten Blick verwirrenden Welt öffnet, wird sehr schnell überfordert sein. Hier empfiehlt es sich neben dem Lesen allgemeiner Fachliteratur, einführende und systematische Anleitungen zu Rate zu ziehen.

Genau dieses Ziel verfolgt das vorliegende Getting-Started-Tutorial, indem systematisch und Schritt-für-Schritt der typische ML-Arbeitsablauf am Beispiel des sehr leistungsfähigen **Support Vector Classifier (SVC)** demonstriert wird.

Dieses Tutorial wird im Rahmen eines Workshops auf der **Fachtagung "Künstliche Intelligenz"**, ausgerichtet durch die Deutsche Gesetzliche Unfallversicherung (DGUV), voraussichtlich im November 2022 in Dresden vorgestellt. Der Workshop richtet sich an interessierte ML-Neulinge im technischen Arbeitsschutz der gesetzlichen Unfallversicherungsträger.

Neben den medial sehr präsenten **tiefen neuronalen Netzen** gibt es eine sehr reichhaltige Auswahl anderer sehr leistungsfähiger ML-Algorithmen - passend für den jeweiligen Anwendungsfall. Für einen allgemein verständlicheren Einstieg wurde für die Zielgruppe des Workshops der SVC-Algorithmus bewusst gewählt. Dessen Arbeitsweise ist sowohl für ML-Neulinge als auch in dem für den Workshop vorgegebenen Zeitrahmen leicht vermittelbar - ganz im Gegensatz zum Einstieg in die Welt der tiefen neuronalen Netze.

Die folgenden Hauptabschnitte demonstrieren den typischen ML-Arbeitsablauf Schritt-für-Schritt. Im **Schritt 0** werden konkrete Hinweise für die Auswahl der für das maschinelle Lernen geeigneten Hardware und Software gegeben. Damit sich ein ML-Neuling zunächst mit den ML-Algorithmen, Werkzeugen, Bibliotheken und Programmiersystemen vertraut machen kann, wird im **Schritt 1** der fertige und sehr einsteigerfreundliche **Iris-Datensatz** hinzugezogen. Erst nach einer umfassenden Einarbeitung in die Anwendung der ML-Werkzeuge wäre es sinnvoll, die eigene Umgebung auf ML-taugliche Anwendungen hin zu untersuchen und daraus geeignete Datensätze zu gewinnen. Dies geht jedoch über den Rahmen dieses einführenden Tutorials hinaus.

Mit der wichtigste Schritt im gesamten ML-Prozess ist **Schritt 2**, in dem der in Schritt 1 einbezogene Datensatz mit Hilfe typischer Datenanalyse-Werkzeuge untersucht wird. Neben der Erkundung der **Datenstruktur** sowie **innerer Zusammenhänge** im Datensatz müssen auch Fehler wie z. B. Lücken, Dopplungen oder offensichtliche Fehleingaben gefunden und nach Möglichkeit behoben werden. Dies ist enorm wichtig, damit die Klassifikation später plausible Ergebnisse liefern kann.

Nach der Erkundung des Datensatzes muss man sich im **Schritt 3** anhand bestimmter Auswahlkriterien für einen konkreten ML-Algorithmus entscheiden. Neben anderen für den Iris-Datensatz passenden ML-Algorithmen (wie z. B. der entscheidungsbaum-basierte **Random-forests-Classifier**) fällt die begründete Auswahl hier im Tutorial auf den **Support-Vector-Classifier**. Ein entsprechendes SVC-Modell wird nun implementiert.

Im **Schritt 4** wird der Datensatz für die eigentliche Klassifikation per SVC vorbereitet. Je nach gewähltem ML-Algorithmus sowie der Datenstruktur kann es erforderlich sein, dass die Daten vor dem Training aufbereitet werden müssen (z. B. durch Standardisierung oder Normalisierung). Nach der Aufteilung des Datensatzes in einen Trainings- und Testdatensatz, wird das SVC-Modell im **Schritt 5** mit dem Trainingsdatensatz trainiert. Anschließend werden mit dem trainierten SVC-Modell anhand der Testdaten Klassifikationsvorhersagen getroffen. Im **Schritt 6** wird die Güte des Klassifikationsergebnisses anhand bekannter **Metriken** wie z. B. der **Konfusionsmatrix** evaluiert.

Da die Klassifikation im Schritt 5 zunächst mit Standard-Parametern (den sogenannten **Hyper-Parametern**) durchgeführt wurde, wird ihre Bedeutung im **Schritt 7** erklärt und danach ihr Einfluss auf das Klassifikationsergebnis durch manuelle Variation der einzelnen Hyper-Parameter demonstriert.

Im abschließenden **Schritt 8** werden zwei Ansätze zur systematischen Hyper-Parameter-Suche vorgestellt: **Grid Search** und **Randomized Search**. Während bei ersterer für gegebene Werte erschöpfend alle Parameterkombinationen betrachtet werden, wird beim zweiten Ansatz eine Anzahl von Kandidaten aus einem Parameterraum mit einer bestimmten zufälligen Verteilung ausgewählt.

1.3 Steps of the systematic ML process

The following **steps of the systematic ML process** are covered in the next main sections:

- STEP 0: Select hardware and software suitable for ML
- STEP 1: Acquire the ML dataset
- STEP 2: Explore the ML dataset
- STEP 3: Choose and create the ML model
- STEP 4: Preprocess the dataset for training
- STEP 5: Carry out training, prediction and testing
- STEP 6: Evaluate model's performance
- STEP 7: Vary parameters of the ML model manually
- STEP 8: Tune the ML model systematically

2 STEP 0: Select hardware and software suitable for ML

In this step, specific guidance is provided for selecting hardware and software suitable for machine learning.

2.1 Community Support

When selecting and deciding for or against the use of certain hardware and software components, in addition to purely technical or financial characteristics, significant attention should be paid to broad **support from a well-networked community**. This community should consist of a balanced share of **manufacturers** of hardware components (e.g. GPU suppliers, manufacturers of embedded systems or sensors), **software developers** ideally from the **open source** ecosystem, and an active **user community** (e.g. for reporting hardware and software bugs or providing help in forums).

The author's many years of development experience show that the technically best hardware or software component is worthless if you are (apparently) the only user. This impression arises either because the component is actually very exotic and has only a few users or because the development takes place "behind closed doors", i.e. in the company's internal **closed source** domain.

Without the support of an active community, you are (almost) on your own when it comes to questions or problems. Progress in the development and maintenance of an AI application is therefore very difficult! The clear recommendation is therefore: Go for the (technically, price-wise, etc.) **second-best alternative** but with an even bigger **community**.

2.2 Hardware

When considering hardware requirements, two systems and their use cases must be taken into account: the **training system** and the **application system**.

2.2.1 Training system

The **training phase** requires a lot of **computational power** and **memory (RAM)**, depending on the **amount of data** to be processed and the **ML algorithm (so-called estimator)** chosen.

Depending on the estimator model, highly parallel processing on a **Graphics Processing Unit (GPU)** can provide significant **speed advantages** over processing on a **Central Processing Unit (CPU)** (e.g., when training deep neural networks in the area of **deep learning**). To take advantage of this speed benefit, the AI application must be suitable in terms of **parallelizability** of the estimator model used as well as **GPU support** through special driver layers, the so-called Operating System Abstraction Layer (OSAL) (Wikipedia: OSAL 2022).

Such GPUs are installed on powerful **3D graphics cards**. However, these must be explicitly qualified for the application for AI - not every game-suitable graphics card from any manufacturer can be used. The manufacturer **Nvidia** offers GPUs suitable for AI in its high-performance graphics cards with **CUDA architecture**. CUDA stands for "Compute Unified Device Architecture" and is a **programming interface** (API) developed by Nvidia, with which program parts can be processed by the graphics processor (Wikipedia: CUDA 2022). A GPU with its several tens of thousands of threads can process highly parallelizable tasks that require only little data communication between the memory areas significantly more performantly than conventional CPUs. This speed advantage can be considerable despite currently available CPU technologies like **Multicore** with **Hyper-Threading** with Intel CPUs!

Nvidia graphics cards with CUDA-supporting GPUs are ranked based on their **compute capability** (NVIDIA.Developer 2022).

However, it should be mentioned that currently only the manufacturer Nvidia offers 3D graphics cards with CUDA implementation, since CUDA is a **proprietary** framework. In addition, there is also the much less well-known **open source** alternative **OpenCL**, which has now been implemented by a large number of graphics card manufacturers (Wikipedia: OpenCL 2022). Since OpenCL is an **open industry standard**, Intel and AMD chips and their GPUs, ATI Radeon cards of the 5, 6, 7 and R9 series as well as various Nvidia GeForce cards are supported, for example.

Regarding the **code execution performance** of both alternatives in direct comparison, there are different statements in the technical literature. The 2011 paper A Performance Comparison of CUDA and OpenCL sees the CUDA implementation as the clear favorite (Karimi, Dickson, and Hamze 2011). More recent publications point out the strong dependence of performance on **code quality**, **algorithm type** and the **GPU hardware** used, among other things - see e.g. here: CUDA vs OpenCL: Which to Use for GPU Programming (Exterman 2021).

It is therefore recommended that the decision for **CUDA or OpenCL** should depend on the extent to which most of the applications employed and the GPU hardware used are better supported by one of the two approaches in each case.

The **state of the art** should be also taken into account when selecting the rest of the training system's hardware. Otherwise, seemingly (price-wise) inexpensive components could very quickly nullify the speed advantage of the GPU. In addition to a mainboard suitable for one (or more) high-performance graphics cards with a correspondingly powerful BUS system (e.g. PCI Express), the RAM should be as large as possible (min. 64 GB) and fast. A large RAM allows, for example, the **virtualization** of several parallel systems in the form of **virtual machines** and thus a significantly better utilization of the available computing capacity (Wikipedia: VM 2022). The permanent memory should also be as large and fast as possible - high-performance **solid-state drives (SSDs)** should be clearly preferred over classic hard disks (HDDs).

2.2.2 Application system

In the **application phase** of the trained estimator model, considerably less computing power and RAM are usually required. If the concrete application does not require **continuous learning during operation**, significantly less expensive systems (in terms of acquisition costs, power consumption, etc.) can also be used. Such application-specific **embedded systems** have only one CPU (usually in **ARM architecture**), comparatively limited RAM (e.g. 1 - 8 GB) and usually no GPU. A popular **embedded computer** that is very well supported in terms of ML software is the **Raspberry Pi** (Wikipedia: Raspi 2022). In addition to its ARM CPU, the Raspberry Pi also has a GPU installed on the same processor in the so-called **System on a Chip design (SoC)**. However, the SoC manufacturer **Broadcom** does not support the CUDA API.

There are references in the technical literature that the open source alternative **OpenCL** can be installed on the Raspberry Pi and that the AI framework **TensorFLow** (see section "Software") can be compiled with **SYCL** support, where SYCL stands for "Single Source OpenCL" (Wikipedia: SYCL 2022). However, a first rough review gives the impression that support for this approach is still very experimental at the moment. Therefore, parallelizing the AI application on the GPU of the Raspberry Pi does not seem to be an option (yet). Here are some links for further reading:

- Deep learning with Raspberry Pi and alternatives in 2022 (Politiek 2022a)
- Benchmarking Machine Learning on the New Raspberry Pi 4, Model B (Allan 2019)
- Portable Computer Vision: TensorFlow 2.0 on a Raspberry Pi (Johnson 2019)
- Install OpenCL on Raspberry Pi 3 B+ (Politiek 2022b)
- Does TensorFlow Support OpenCL? (IndianTechWarrior 2022)
- TensorFlow for OpenCL using SYCL (Iwanski 2016)

2.3 Software

2.3.1 Programming languages

The comparison of **advantages and disadvantages** of the various programming languages and the evaluation of their suitability for ML was inspired by the following articles, among others:

- What Is the Best Language for Machine Learning? (Gupta 2021)
- Is Octave Good for Machine Learning? (Adhikari 2021)

In summary, there is **no best language for machine learning**, each is good where it fits best.

However, there are definitely some programming languages that are better suited for machine learning tasks than others (Gupta 2021). On the one hand, this is due to whether the programming language is

fundamentally well suited to **implement complex mathematical and statistical tasks** in efficient algorithms.

On the other hand, when deciding for or against a programming language, it should definitely also be taken into account whether it contains sufficient **basic functionalities for data analysis and its processing**, as well as very diverse **extension libraries** (so-called **packages**) that are well supported by the community are available. By using these libraries, it is possible to concentrate on the concrete task when creating an ML application and not have to constantly solve the same trivial problems anew in every new application (e.g. the efficient **handling of datasets** or the execution of **matrices calculations**).

Following trend chart shows how the popularity of selected programming languages suitable for machine learning has evolved since 2008:

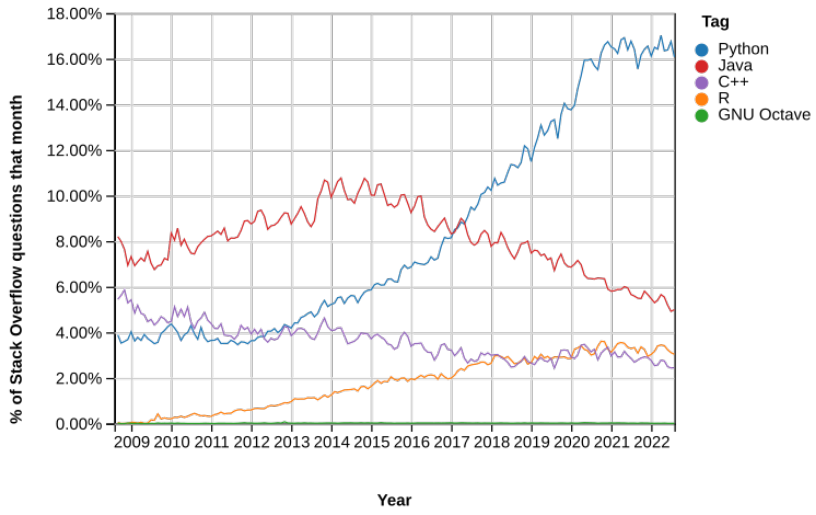


Figure 1: Trend chart shows popularity of programming languages for ML (source: Stack Overflow Trends, license: CC BY-SA 4.0)

Python It is a high-level, **general-purpose** programming language where its design philosophy emphasizes **code readability**. The **variable types** in Python are **dynamic** and **memory** is **automatically managed** to create and delete data objects (see garbage collection).

Pros:

- Python offers simple, concise, and **readable code** for allowing to write robust and reliable programs.
- It lets you focus on solving the ML problem instead of getting lost in the language's technical nuances.
- Python has **extensive libraries for ML**, e.g. `Scikit-learn`, `Pandas`, `TensorFlow` or `Keras` have become standard libraries for various ML tasks.
- The language has been around for decades and has developed a large and helpful community.
- Besides extensive online documentation, there are thousands of question-answers and community guides for various functionalities of the language (this is also very well reflected in the trend graph on the popularity of programming languages).

Cons:

- (unknown drawbacks so far …)

R It is a programming language for **statistical computing** and **graphics** supported by the **R Core Team** and the **R Foundation for Statistical Computing**. Created by statisticians Ross Ihaka and Robert Gentleman, R is used among data miners, bioinformaticians and statisticians for data analysis and developing statistical software.

Pros:

- After Python, R is the recommended ML programming language.
- R is a flexible and cross-platform compatible language.
- It has a growing, supportive community.
- R is well suited for data visualization and **statistics**, often making it the language of choice for applications with a large amount of statistical data.
- It is considered a powerful choice for **machine learning**, offering a variety of machine learning techniques (e.g., data visualization, data sampling, data analysis and supervised and unsupervised machine learning models) via post-installable libraries.

Cons:

- R is often reported to be laggier and slower as compared to Python when dealing with large-scale datasets.
- It has a **significantly lower community** support when answering questions or giving guidance **compared to Python** (see trend chart on popularity of programming languages).
- The **learning curve** for the basic entry into R and the application in more complex projects for data analysis or machine learning is significantly **steeper than with Python**.

Java It is a high-level, **class-based**, **object-oriented** programming language that is designed to have **as few implementation dependencies** as possible. It is a **general-purpose** programming language intended that compiled Java code can run on all platforms that support Java without the need to recompile.

Pros:

- Using Java for machine learning is especially popular among developers with a Java background, as it skips the need to learn another programming language such as Python or R.
- Like Python and R, Java also has a variety of third-party machine learning libraries, e.g. **JavaML** is a built-in library with a collection of algorithms implemented in Java for ML.
- Scalability is an important feature for many ML projects, which is well supported by Java.
- Java Virtual Machine (JVM) enables the development of ML applications for multiple platforms.
- Java is very well suited for speed-critical ML projects.

Cons:

- Java has a much lower community support in answering questions or giving guidance compared to Python - but a better one than R (see trend chart on popularity of programming languages).

GNU Octave It is a high-level programming language that's **designed for numerical computations** (Adhikari 2021).

Pros:

- With Octave, **linear and non-linear numerical problems** can be solved quickly.
- Octave is syntactically very similar to MATLAB and mostly **compatible with MATLAB**. If no MATLAB-specific functions are used, the program code also runs in Octave. In addition, Octave even has some language functions and a syntax diversity that MATLAB lacks.

Cons:

- However, Octave is not a good programming language for machine learning in a production environment.
- It doesn't have the same functionality as other languages used for ML, due to **missing libraries** and frameworks to speed up ML tasks.
- It's not as flexible, simple, and feature-rich as other programming languages.
- Compared to Python, R and Java, Octave has almost **no community support** when it comes to answering questions or providing guidance (compare trend chart on popularity of programming languages).

2.3.2 Python packages

The **mathematics** and the **numerical implementation** of various algorithms for data analysis and machine learning are usually **very complex** and often only comprehensible for ML experts (Gupta 2021). For a broad and praxis-oriented **usability**, better **reusability of code** and a successful **integration** into a concrete ML application, the functional relationships should be **encapsulated in libraries** (so-called "packages").

From the user's point of view, when selecting libraries for the respective task, attention should be paid not only to functionality but also to the **comprehensibility of the user interface supported by good documentation**. Furthermore, the **size of the community** behind the library, consisting of active developers as well as technical experts for supporting the users in the event of questions or problems arising, should be decisive in the selection.

Following trend chart shows how the popularity of selected python packages suitable for **data analysis**, **data visualization** and **machine learning** has evolved since 2008:

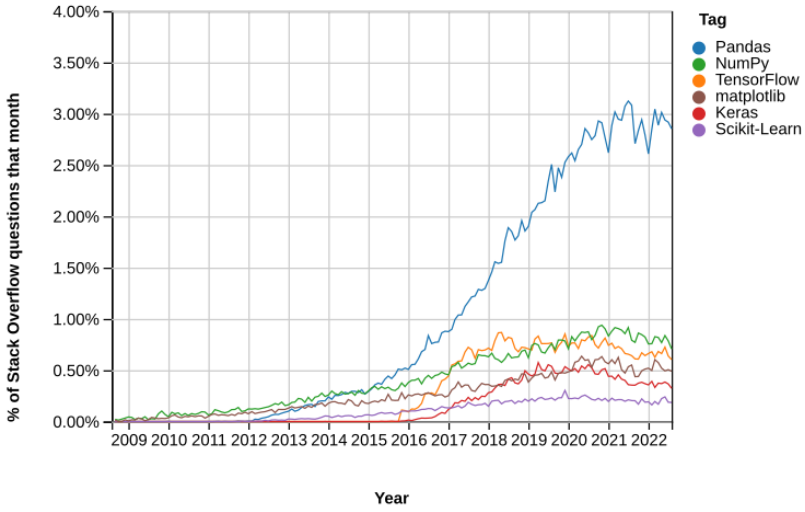


Figure 2: Trend chart shows popularity of selected python packages for data analysis, data visualization and machine learning (source: Stack Overflow Trends, license: CC BY-SA 4.0)

In the scientific research and systematic improvement of ML algorithms, a very dynamic progression can be observed in recent years. The latest scientific findings are regularly compared with each other in **"Machine Learning Competitions"** using known and **freely available datasets** (see benchmarking competitions of ML algorithms on platforms such as `https://www.kaggle.com/competitions`). At the same time, the corresponding ML libraries are revised, extended and made available to general users by the scientific community. Therefore, this **scientific transfer** ideally takes place in the context of **open source developments**.

Due to the superior advantages of **Python** (see previous section), a selection of **open source** packages available for this programming language usable for **data analysis**, **data visualization** and **machine learning** are presented in this section.

Data analysis

- NumPy is a Python library that provides a **multidimensional array object**, various derived objects (such as masked arrays and **matrices**), and an assortment of routines for **fast operations on arrays**, including mathematical, logical, shape manipulation, sorting, selecting, discrete Fourier transformations, basic linear algebra, basic statistical operations, random simulation and much more.
- Pandas is a Python package providing fast, flexible and expressive data structures designed to work with **relational** or **labeled** datasets. It provides two primary data structures: `pandas.Series` (1-dimensional time series) and `pandas.DataFrame` (2-dimensional spreadsheets). The data structure

`pandas.DataFrame` offers the same functionality as the structure `data.frame` known from the programming language R and much more.

Data visualization

- Matplotlib is a library for making **2D plots of arrays** in Python. Although it has its origins in **emulating the MATLAB graphics commands**, it is independent of MATLAB, and can be used in a Pythonic, object oriented way. Although Matplotlib is written primarily in pure Python, it makes heavy use of NumPy and other extension code to provide good performance even for large arrays (Hunter 2007).
- Seaborn is a library for making **statistical graphics** in Python. It builds **on top of matplotlib** and integrates closely with **pandas data structures**. Seaborn helps to explore and understand the data. Its plotting functions operate on **dataframes** and **arrays** containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots (Waskom 2021).

Machine learning

- Scikit-Learn is a **free software machine learning library** for **Python**. It features various **classification**, **regression** and **clustering** algorithms including **support-vector machines**, **random forests**, **gradient boosting** and **k-means**. It is designed to interoperate with the Python numerical and scientific libraries **NumPy** and **SciPy**. Scikit-Learn will be used in the next steps of this of this getting started tutorial.
- TensorFlow offers, among other things, the possibility to create and train **artificial neural networks (ANN)** based on **Google AI**. it is an open source software library for **machine learning** and **artificial intelligence**. It can be used across a range of tasks but has a particular focus on training and inference of **deep neural networks**. However, the installation and usage is very much beyond the scope of this beginner tutorial.
- Keras is an open source software library for **deep learning** that provides a Python interface for **ANNs**. Keras acts as an **general interface** for several **backends**, such as **TensorFlow**, **Microsoft Cognitive Toolkit** and **Theano**. Keras will also not be used in this beginner tutorial.

2.3.3 Import Python packages globally

The aim of this section is to import globally used Python packages for data analysis and ML, such as `Pandas`, `NumPY`, `matplotlib` and `Scikit-Learn`.

[1]:
```
import time

from IPython.display import HTML

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm, metrics
import seaborn as sns
%matplotlib inline

# Set font sizes of figure title, axes and labels
# globally via a rcParams dictionary
import matplotlib.pylab as pylab
params = {'legend.fontsize': 'large',
          'axes.labelsize':   'large',
          'axes.titlesize':   'large',
          'xtick.labelsize':  'medium',
          'ytick.labelsize':  'medium',
```

```
        'axes.edgecolor':   '#000000'}
pylab.rcParams.update(params)
```

2.3.4 Programming IDEs

Integrated development environments (IDE) are software applications that provide comprehensive features to computer programmers for **software development**. An IDE typically consists of a **source code editor**, automated **build tools** for compiling or an **interpreter** for scripting languages, a front end to the **version control system** like e.g. Git and a **debugger** (Wikipedia: IDE 2022).

Following trend chart shows how the popularity of selected IDEs suitable for ML programming languages has evolved since 2008:

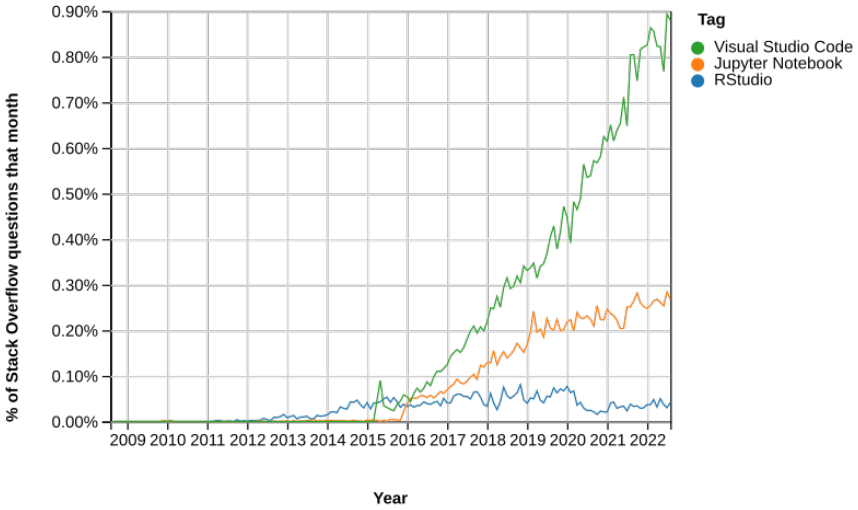


Figure 3: Trend chart shows popularity of selected IDEs for ML programming languages (source: Stack Overflow Trends, license: CC BY-SA 4.0)

Visual Studio Code (VSC) It is an IDE made by **Microsoft** for **Windows**, **Linux** and **macOS**. Features include support for **debugging**, **syntax highlighting** for many different programming languages, intelligent **code completion** and embedded **version control system** Git. Users can change the theme, keyboard shortcuts, preferences, and install **extensions** from a huge repository that add additional functionality. Despite of its platform independence, VSC is **not open source** - in fact it is released under a traditional Microsoft product license.

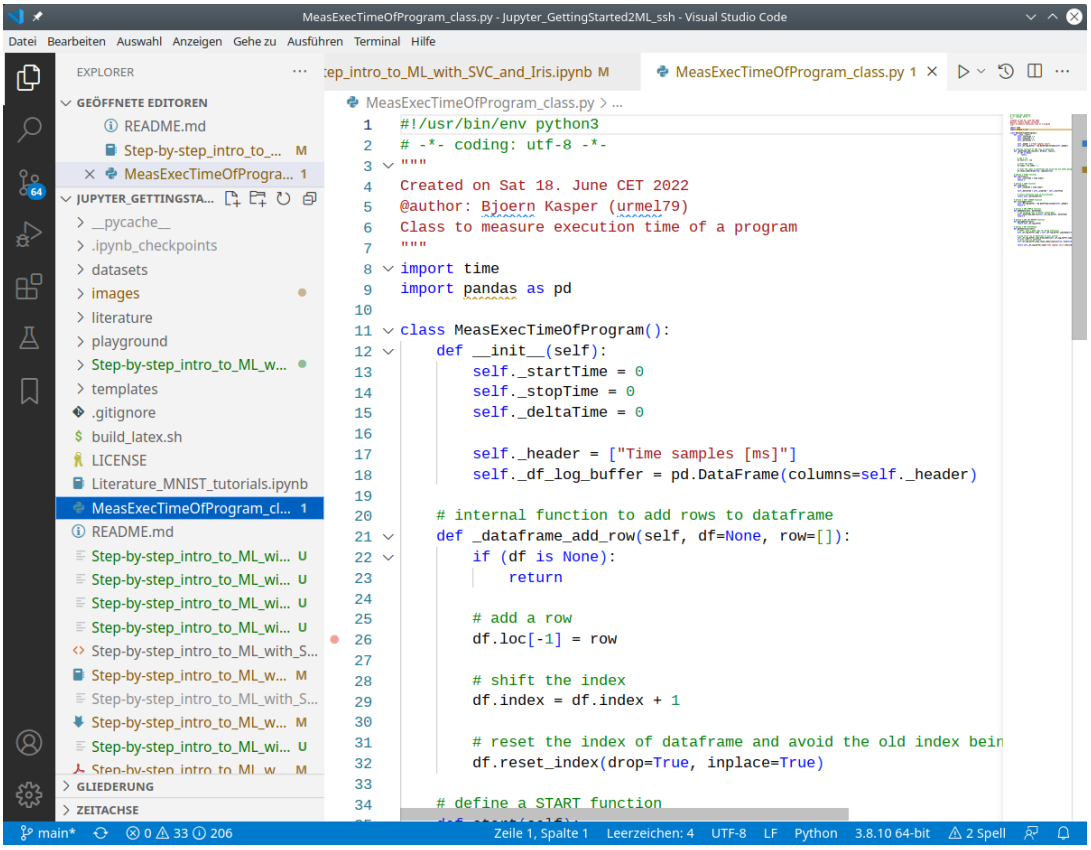


Figure 4: Screenshot of IDE *Visual Studio Code* (source: Kasper, license: CC BY-SA 4.0)

JupyterLab It is the successor product for the web-based interactive environment **Jupyter Notebook**. Within this IDE, Jupyter Notebook documents can be created, edited, and executed interactively. The notebooks consist of **input and output cells**, each of which can contain program code, formatted text in **Markdown** format and live plots generated from the code.

Jupyter is a new **open source** alternative to the proprietary numerical software Mathematica from **Wolfram Research** that is well on the way to becoming a **standard for exchanging research results** (Somers 2018; Romer 2018).

Originally Jupyter was intended as an IDE for the programming languages **Julia** and **Python**. Besides that it is also possible to install other interpreter kernels, such as the **IRkernel** for R. This can be interesting if the IDE **RStudio Desktop** is not available on the target platform used. For example, it is very difficult to install RStudio on the ARM-based embedded computer **Raspberry Pi** due to many technical dependencies. In contrast, using the R kernel in JupyterLab on the Raspberry Pi works very well and performant.

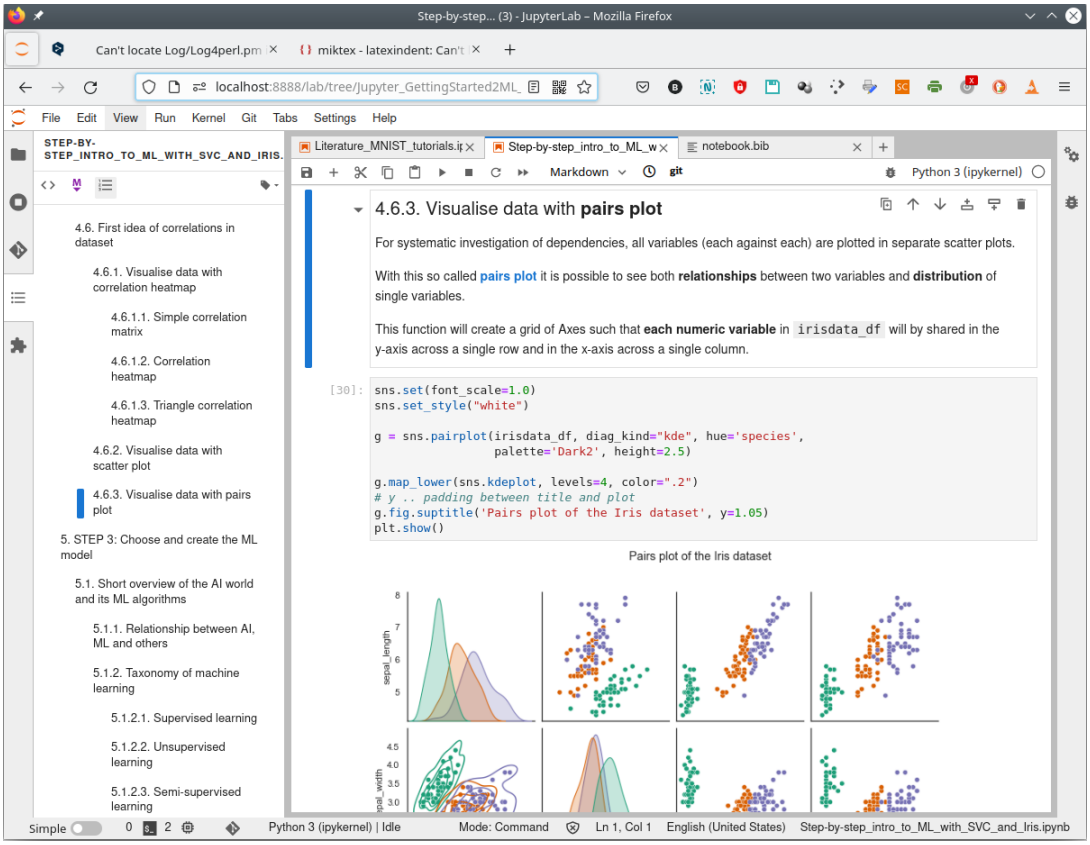


Figure 5: Screenshot of IDE *JupyterLab* (source: Kasper, license: CC BY-SA 4.0)

RStudio It is an IDE and graphical user interface for the statistical programming language **R** offered by **RStudio, Inc.** and is made available in two formats. **RStudio Desktop** is a regular desktop application while **RStudio Server** runs on a remote server and allows accessing RStudio using a web browser. Both software products are available in **open source** and **commercial** versions, each with different functionalities.

The program editor in RStudio allows **autocompletion**, **automatic indentation**, **syntax highlighting**, **code folding** as well as **integrated help** and information about functions and objects in the working environment. There is the ability to view and edit the contents of variables and datasets. To facilitate collaboration, scripts, data and other files can be combined into projects (.Rproj) and versioned with **Git**.

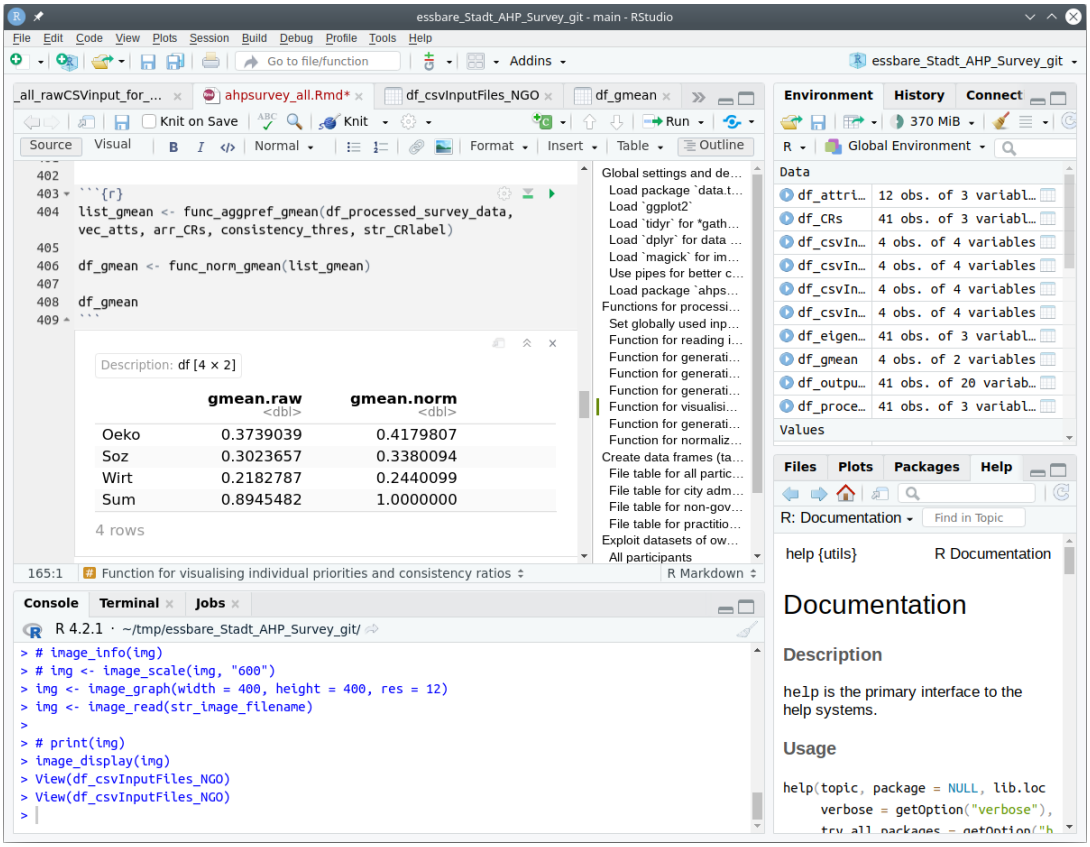


Figure 6: Screenshot of IDE *RStudio* (source: Kasper, license: CC BY-SA 4.0)

GNU Octave (GUI) It is the official graphical user interface for the **GNU Octave** programming language and is available for Windows, macOS, Linux and BSD under **Open Source** licensing.

If the command line interpreter (CLI) starts instead of the graphical user interface (GUI) when `octave` is called, this can be forced via the `octave --gui` option.

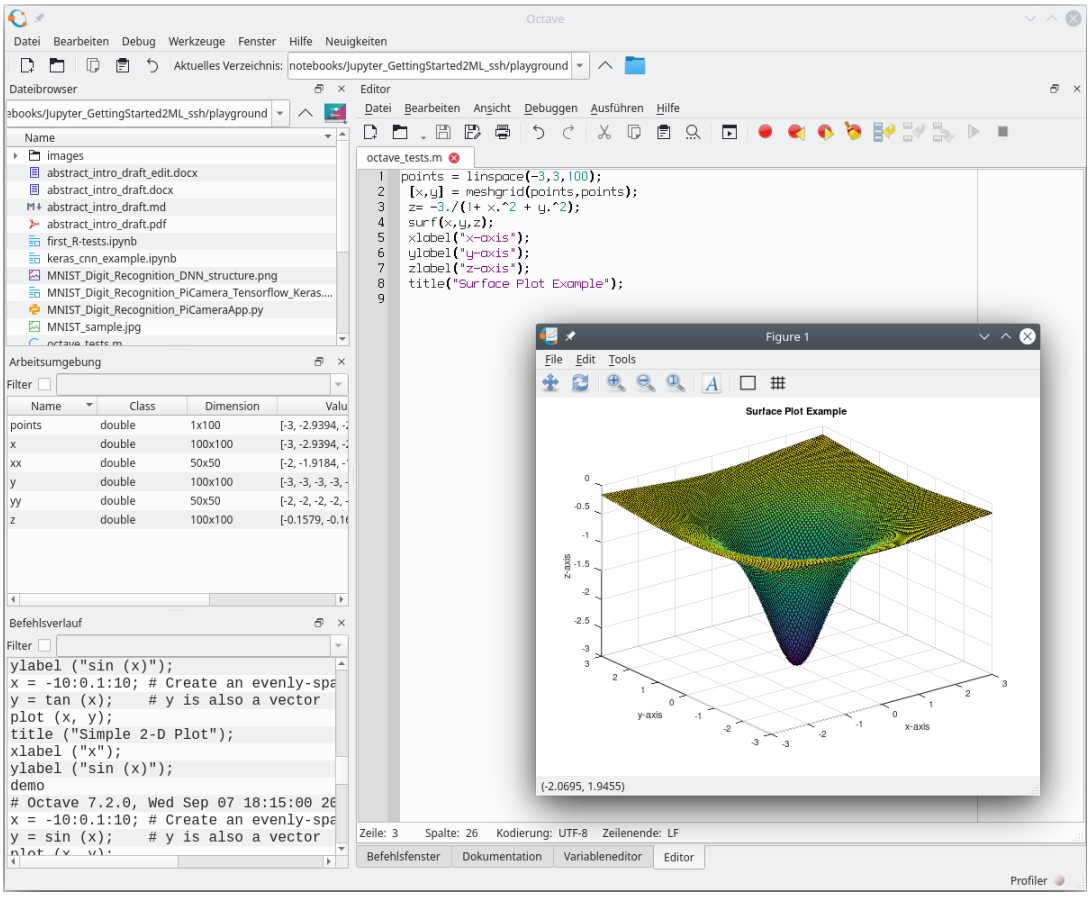


Figure 7: Screenshot of IDE *GNU Octave* (source: Kasper, license: CC BY-SA 4.0)

2.3.5 Cloud-hosted IDEs

A very interesting alternative to own, local and for the ML application adequately powerful and thus price-intensive hardware resources can be **cloud-hosted Jupyter environments**. These offer features such as cloud storage, model training and deployment capabilities, version control and much more.

Since the entire hardware and backend configurations are hosted in the cloud by the various providers, the user can concentrate on creating his ML application. The cloud provider takes care of purchasing the hardware and the sometimes time-consuming installation and configuration of the programming environment (Rapp 2021).

The cloud environments briefly presented here can be used freely after registration - on condition that own projects remain accessible to other researchers. Even in the free variant, GPUs and Tensor Processing Units (TPUs) can be selected in the project for hardware acceleration. This is particularly interesting for training deep neural networks.

In the premium versions, for example, more powerful GPUs and TPUs as well as more memory can be accessed. Additionally, there is the option to keep the projects private and thus prevent accessibility for other researchers.

However, with all the advantages, **data protection aspects** should definitely be considered. Before using a cloud environment, it should be clarified whether and to what extent, for example, **own datasets with personal data** may be uploaded to the cloud projects. If there are uncertainties here, local and self-hosted ML resources should be used in any case!

Google Colaboratory In recent years, **Google Colaboratory** (**Colab** for short) has become a popular choice for cloud-based Jupyter notebooks. Thanks to its free-to-use GPUs and cloud storage linked to Google Drive, it is used by many researchers in the ML and data science community (Rapp 2021).

Due to the similarity of the web interface to Jupyter, Python developers can write and run arbitrary Python program codes. Colab is a cloud-hosted version of Jupyter Notebook that provides free access to compute infrastructure such as memory, storage, processing capacity, GPUs and TPUs (Das 2022).

Furthermore, commonly used libraries such as **PyTorch**, **TensorFlow** and **Keras** can be used to develop deep learning applications (Misal 2021).

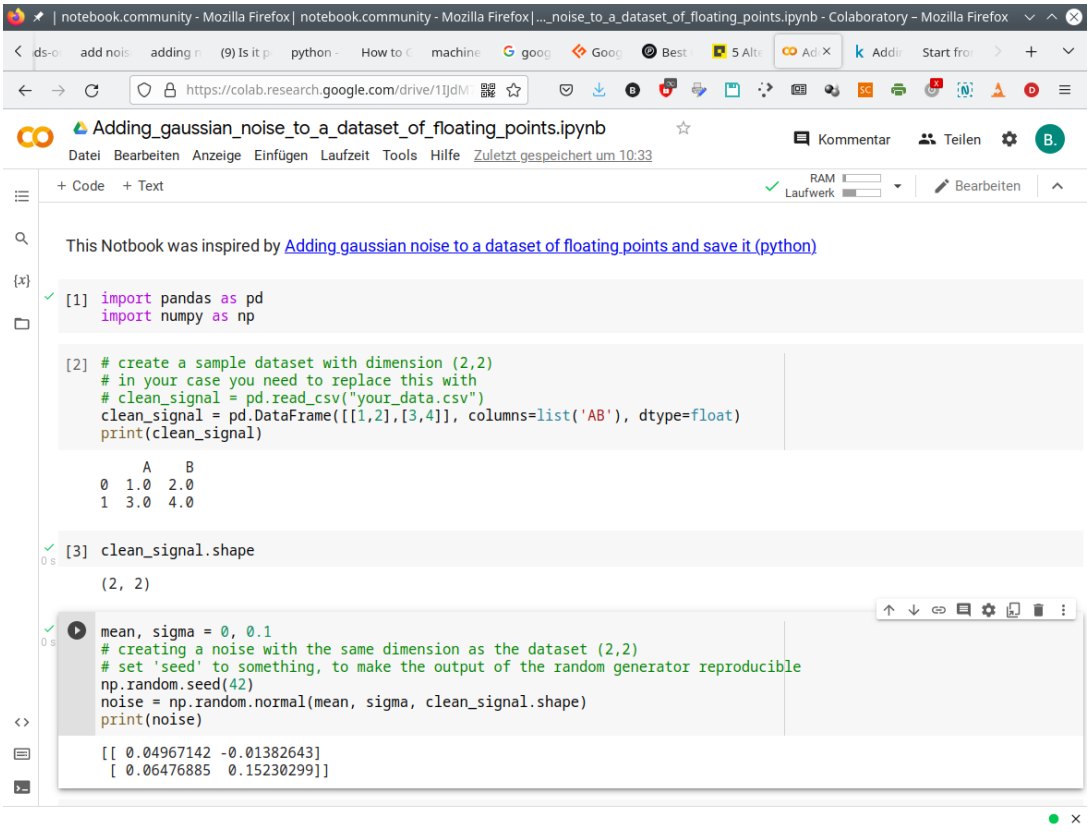


Figure 8: Screenshot of IDE *Google Colaboratory* (source: Kasper, license: CC BY-SA 4.0)

Google Kaggle This is another Google product with similar functionality to Colab. Like Colab, **Kaggle** also offers free browser-based Jupyter notebooks and the use of GPUs. Kaggle also has many **Python packages pre-installed**, which lowers the barrier to entry for many users (Rapp 2021).

Kaggle and Colab have a number of similarities - among other things, most of the keyboard shortcuts are the same as in Jupyter notebooks. Furthermore, many datasets can be imported. Kaggle has a large user community to learn and improve data science skills (Misal 2021).

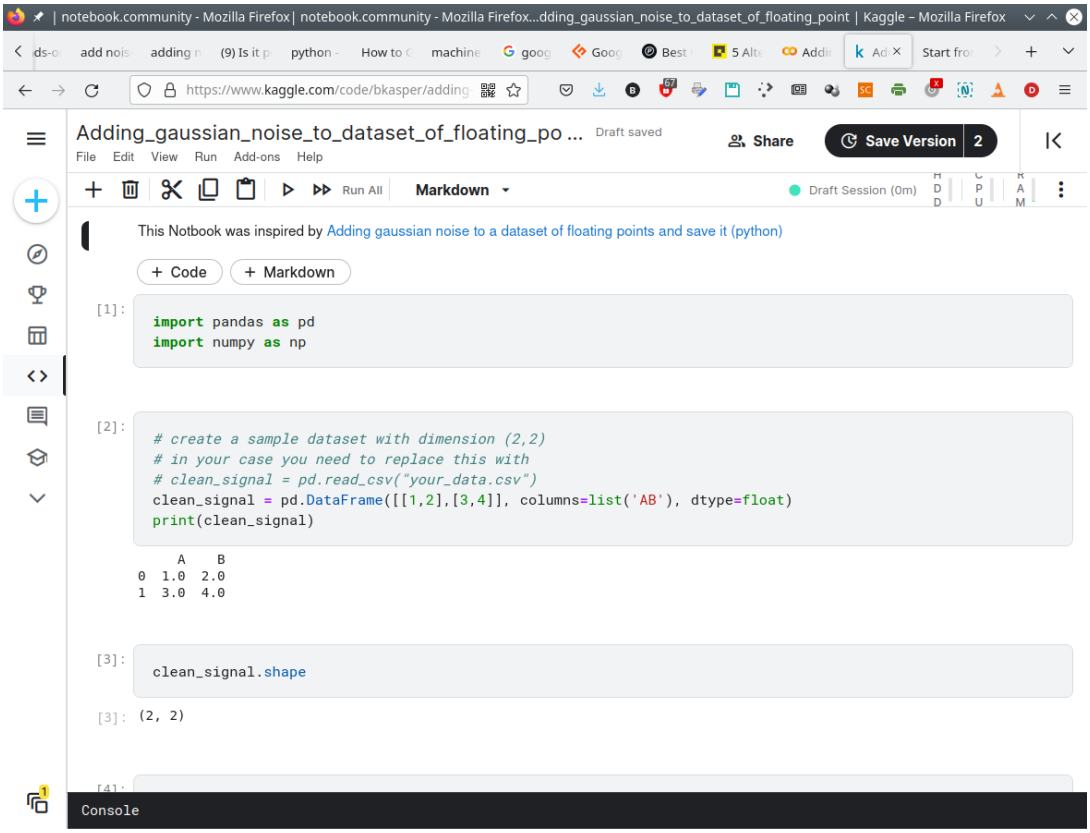


Figure 9: Screenshot of IDE *Google Kaggle* (source: Kasper, license: CC BY-SA 4.0)

Paperspace Gradient Unlike Colab, **Paperspace Gradient** can implement entire **ML workflows** from data pre-processing to training models to deploying the trained models. Furthermore, Gradient has features like a CLI tool, more control over the GPU and simpler data management services. Due to the variety of functions, one must first become familiar with the operation of the significantly more complex user interface (Siow 2020).

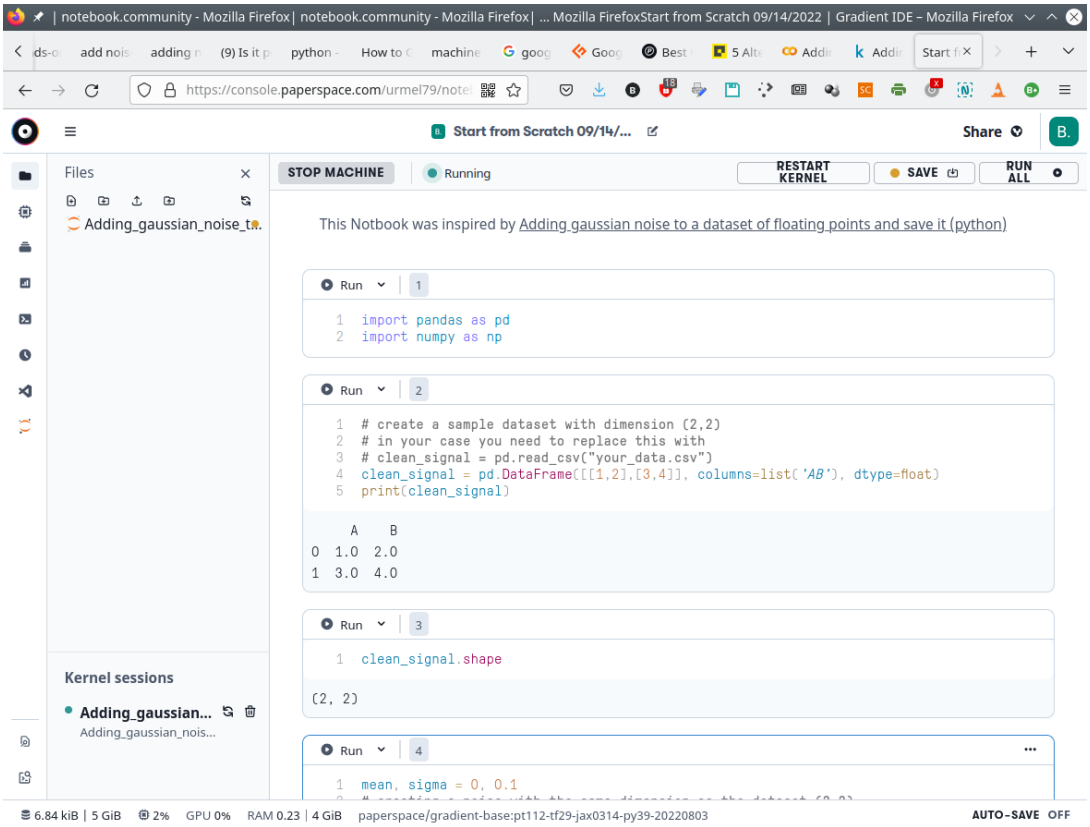


Figure 10: Screenshot of IDE *Paperspace Gradient* (source: Kasper, license: CC BY-SA 4.0)

2.3.6 Operating systems

The **programming languages**, **Python libraries** and **development environments** presented in the previous sections are available for different operating systems, such as **Linux**, **Windows** and **macOS**. Therefore, the decision for or against an operating system may **depend on the technical background** of the ML developer.

Nevertheless, the following general **requirements** can be specified for an operating system **suitable for software development**:

- **Openness**: availability of very good interface documentation and ideally open source software
- **Self-administration**: user has full installation and configuration rights
- **Communication capability**: unfiltered and bidirectional communication in the local network as well as to the internet on all necessary protocols possible
- **Extensibility**:
  - automated software installation and update management via central package management systems such as `apt`, `pip` or `conda`
  - possible integration of additional software libraries or external sensor hardware

Following trend chart shows how the popularity of selected operating systems used by **data analysts** and **ML developers** has evolved since 2008:

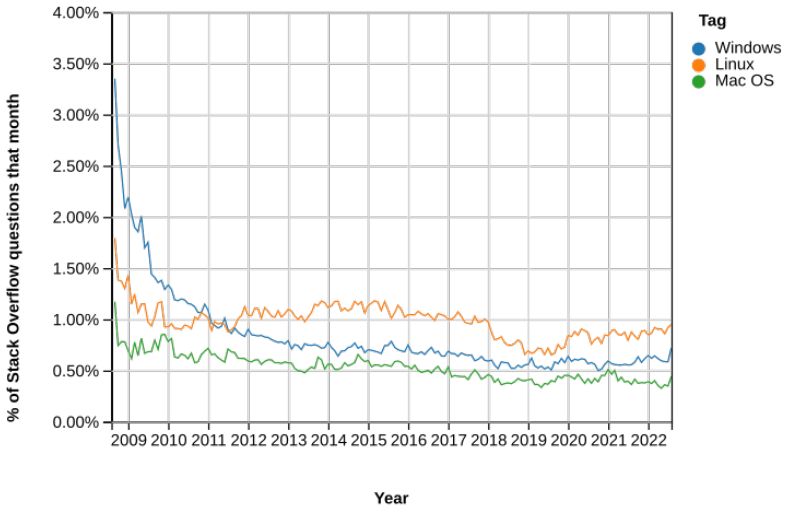


Figure 11: Trend chart shows popularity of selected operating systems used by **data analysts** and **ML developers** (source: Stack Overflow Trends, license: CC BY-SA 4.0)

For **security** reasons, the **IT departments** of many employers massively **restrict installation and configuration rights**. Furthermore, very restrictive firewall settings severely **restrict** unfiltered and bidirectional **communication** in the local network and to the internet. Automated **software installations** via package managers are often **not possible** or only possible with difficulty due to blocked protocols.

To deal with these challenges, two possible solutions are presented below.

Virtual machine To be able to install, configure and update the required software (IDEs, programming languages and ML packages) independently, the use of a Virtual Machine (VM) could be a possible alternative.

However, there are also significant disadvantages here:

- The **communication problem** is **not solved**, because the VM shares the access to the internet with the host system.
- The **access to 3D graphics cards** is usually **not possible** due to virtualization.
- This solution has only **low application performance**, as regular business computers are often only very sparsely equipped in terms of RAM and processor performance for cost reasons.

Separate lab computer All the problems mentioned in the previous section can only be solved satisfactorily by acquiring a **separate laboratory computer** with **its own internet access** (e.g. via an **LTE-capable wifi router**).

This laboratory computer can be configured according to your own requirements, depending on the available budget in terms of hardware and software.

However, it should be noted here that the **IT departments** of many employers do **not offer any support** for this solution. You are usually responsible for software installation, maintenance and troubleshooting yourself!

3 STEP 1: Acquire the ML dataset

To allow an ML novice to first familiarize themselves with the ML algorithms, tools, libraries, and programming systems, the ready-made and very beginner-friendly **Iris dataset** is involved in this step. Only after a comprehensive acquaintance with the application of ML tools would it make sense to examine one's own environment for ML-suitable applications and to obtain suitable datasets from them. However, this is beyond the scope of this introductory tutorial.

Several details, for example, on the history of the creation of the Iris flower dataset can be found e.g. on Wikipedia (see Wikipedia: Iris dataset 2022).

It can be downloaded on Kaggle: Iris Flower Dataset (Baba 2018). Furthermore, the dataset is available via Python in the machine learning package Scikit-learn, so that users can access it without having to find a special source for it.

[81]:
```
# Import Iris dataset for exploration
irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle.csv')
```

4 STEP 2: Explore the ML dataset

One of the most important steps in the entire ML process is this step, in which the dataset included in Step 1 is examined using typical data analysis tools. In addition to exploring the **data structure** and **internal correlations** in the dataset, errors such as **gaps**, **duplications**, or obvious **misentries** must also be found and corrected where possible. This is enormously important so that the classification can later provide plausible results.

4.1 Goals of exploration

The objectives of the exploration of the dataset are as follows:

1. Clarify the **origins history**:
   - Where did the data come from? → Contact persons and licensing permissions?
   - Who obtained the data and with which (measurement) methods? → Did systematic errors occur during the acquisition?
   - What were they originally intended for? → Can they be used for my application?
2. Overview of the internal **structure and organisation** of the data:
   - Which columns are there? → With which methods can they be read in (e.g. import of CSV files)?
   - What do they contain for (physical) measured variables? → Which technical or physical correlations exist?
   - Which data formats or types are there? → Do they have to be converted?
   - In which value ranges do the measurement data vary? → Are normalizations necessary?
3. Identify **anomalies** in the datasets:
   - Do the data have **gaps** or **duplicates**? → Does the dataset needs to be cleaned?
   - Are there obvious erroneous entries or measurement outliers? → Does (statistical) filtering have to be carried out?
4. Avoidance of **tendencies due to bias**:
   - Are all possible classes included in the dataset and equally distributed? → Does the dataset need to be enriched with additional data for balance?
5. Find a first rough **idea of which correlations** could be in the dataset

4.2 Clarify the origins history

> The **Iris flower datasets** is a multivariate dataset introduced by the British statistician and biologist *Ronald Fisher* in his paper "The use of multiple measurements in taxonomic problems" (Fisher 1936). It is sometimes called *Anderson's Iris dataset* because Edgar Anderson collected the data to quantify the morphologic variation of Iris flowers of three related species (Wikipedia: Iris dataset 2022).

The dataset is published in Public Domain with a CC0-License.

This dataset became a typical test case for many statistical classification techniques in machine learning such as **support vector machines**.

> [..] measurements of the flowers of fifty plants each of the two species *Iris setosa* and *I. versicolor*, found **growing together in the same colony** and measured by Dr E. Anderson (Fisher 1936)

> [..] *Iris virginica*, differs from the two other samples in **not being taken from the same natural colony** (ibidem)

4.3 Overview of the internal structure and organization of the data

The dataset consists of 50 samples from each of three species of Iris: *Iris setosa*, *Iris virginica* and *Iris versicolor*, so there are 150 samples in total (Wikipedia: Iris setosa 2022, Wikipedia: Iris virginica 2022 and Wikipedia: Iris versicolor 2022).

Four features were measured from each sample: the length and the width of the **sepals** and **petals**, in centimetres (Wikipedia: Sepal 2022 and Wikipedia: Petal 2022). Here you can see a principle illustration of a flower in which, among other things, the sepals and petals are shown:

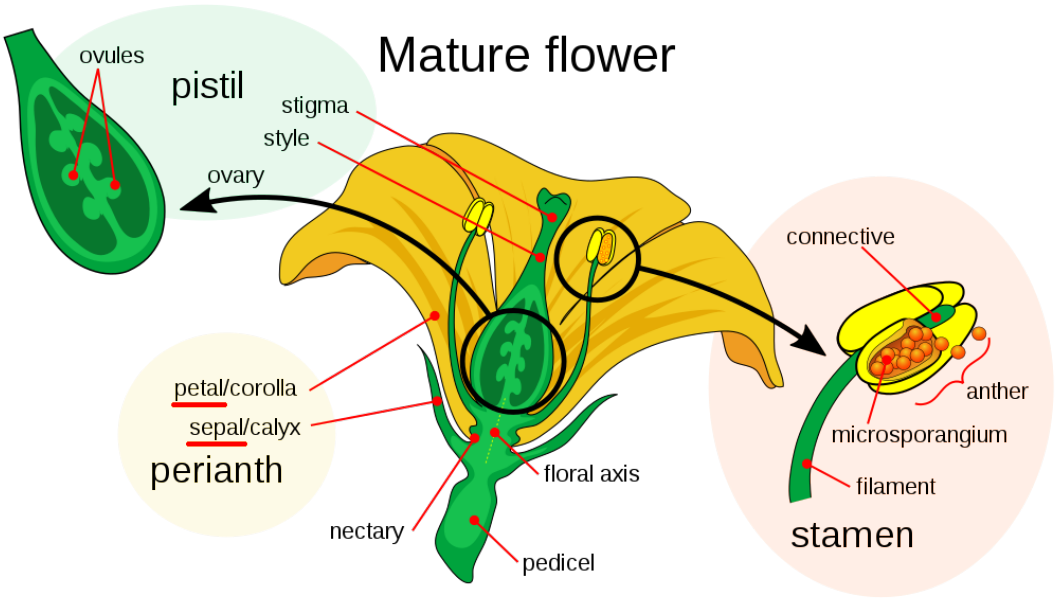


Figure 12: Principle illustration of a flower with sepal and petal (source: Mature_flower_diagram.svg, license: public domain)

Here are pictures of the three different Iris species (*Iris setosa*, *Iris virginica* and *Iris versicolor*). Given the dimensions of the flower, it will be possible to predict the class of the flower.

Figure 13: Left: *Iris setosa* (source: Irissetosa1.jpg, license: public domain); middle: *Iris versicolor* (source: Iris_versicolor_3.jpg, license: CC SA 3.0); right: *Iris virginica* (source: Iris_virginica.jpg, license: CC SA 2.0)

4.3.1 Inspect structure of dataframe

Print first or last 10 rows of dataframe:

[4]:
```
irisdata_df.head(10)
```

[4]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa
8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa

[5]:
```
irisdata_df.tail()
```

[5]:

	sepal_length	sepal_width	petal_length	petal_width	species
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

While printing a dataframe - only an abbreviated view of the dataframe is shown :(
Default setting in the `Pandas` library makes it to display only 5 lines from head and from tail.

[6]:
```
irisdata_df
```

[6]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
..	…	…	…	…	…
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

[150 rows x 5 columns]

To print all rows of a dataframe, the option `display.max_rows` has to set to `None` in `Pandas`:

[7]:
```
pd.set_option('display.max_rows', None)
irisdata_df
```

[7]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
5	5.4	3.9	1.7	0.4	Iris-setosa
6	4.6	3.4	1.4	0.3	Iris-setosa
7	5.0	3.4	1.5	0.2	Iris-setosa

8	4.4	2.9	1.4	0.2	Iris-setosa
9	4.9	3.1	1.5	0.1	Iris-setosa
10	5.4	3.7	1.5	0.2	Iris-setosa
11	4.8	3.4	1.6	0.2	Iris-setosa
12	4.8	3.0	1.4	0.1	Iris-setosa
13	4.3	3.0	1.1	0.1	Iris-setosa
14	5.8	4.0	1.2	0.2	Iris-setosa
15	5.7	4.4	1.5	0.4	Iris-setosa
16	5.4	3.9	1.3	0.4	Iris-setosa
17	5.1	3.5	1.4	0.3	Iris-setosa
18	5.7	3.8	1.7	0.3	Iris-setosa
19	5.1	3.8	1.5	0.3	Iris-setosa
20	5.4	3.4	1.7	0.2	Iris-setosa
21	5.1	3.7	1.5	0.4	Iris-setosa
22	4.6	3.6	1.0	0.2	Iris-setosa
23	5.1	3.3	1.7	0.5	Iris-setosa
24	4.8	3.4	1.9	0.2	Iris-setosa
25	5.0	3.0	1.6	0.2	Iris-setosa
26	5.0	3.4	1.6	0.4	Iris-setosa
27	5.2	3.5	1.5	0.2	Iris-setosa
28	5.2	3.4	1.4	0.2	Iris-setosa
29	4.7	3.2	1.6	0.2	Iris-setosa
30	4.8	3.1	1.6	0.2	Iris-setosa
31	5.4	3.4	1.5	0.4	Iris-setosa
32	5.2	4.1	1.5	0.1	Iris-setosa
33	5.5	4.2	1.4	0.2	Iris-setosa
34	4.9	3.1	1.5	0.1	Iris-setosa
35	5.0	3.2	1.2	0.2	Iris-setosa
36	5.5	3.5	1.3	0.2	Iris-setosa
37	4.9	3.1	1.5	0.1	Iris-setosa
38	4.4	3.0	1.3	0.2	Iris-setosa
39	5.1	3.4	1.5	0.2	Iris-setosa
40	5.0	3.5	1.3	0.3	Iris-setosa
41	4.5	2.3	1.3	0.3	Iris-setosa
42	4.4	3.2	1.3	0.2	Iris-setosa
43	5.0	3.5	1.6	0.6	Iris-setosa
44	5.1	3.8	1.9	0.4	Iris-setosa
45	4.8	3.0	1.4	0.3	Iris-setosa
46	5.1	3.8	1.6	0.2	Iris-setosa
47	4.6	3.2	1.4	0.2	Iris-setosa
48	5.3	3.7	1.5	0.2	Iris-setosa
49	5.0	3.3	1.4	0.2	Iris-setosa
50	7.0	3.2	4.7	1.4	Iris-versicolor
51	6.4	3.2	4.5	1.5	Iris-versicolor
52	6.9	3.1	4.9	1.5	Iris-versicolor
53	5.5	2.3	4.0	1.3	Iris-versicolor
54	6.5	2.8	4.6	1.5	Iris-versicolor
55	5.7	2.8	4.5	1.3	Iris-versicolor
56	6.3	3.3	4.7	1.6	Iris-versicolor
57	4.9	2.4	3.3	1.0	Iris-versicolor
58	6.6	2.9	4.6	1.3	Iris-versicolor
59	5.2	2.7	3.9	1.4	Iris-versicolor
60	5.0	2.0	3.5	1.0	Iris-versicolor
61	5.9	3.0	4.2	1.5	Iris-versicolor
62	6.0	2.2	4.0	1.0	Iris-versicolor
63	6.1	2.9	4.7	1.4	Iris-versicolor
64	5.6	2.9	3.6	1.3	Iris-versicolor

65	6.7	3.1	4.4	1.4	Iris-versicolor
66	5.6	3.0	4.5	1.5	Iris-versicolor
67	5.8	2.7	4.1	1.0	Iris-versicolor
68	6.2	2.2	4.5	1.5	Iris-versicolor
69	5.6	2.5	3.9	1.1	Iris-versicolor
70	5.9	3.2	4.8	1.8	Iris-versicolor
71	6.1	2.8	4.0	1.3	Iris-versicolor
72	6.3	2.5	4.9	1.5	Iris-versicolor
73	6.1	2.8	4.7	1.2	Iris-versicolor
74	6.4	2.9	4.3	1.3	Iris-versicolor
75	6.6	3.0	4.4	1.4	Iris-versicolor
76	6.8	2.8	4.8	1.4	Iris-versicolor
77	6.7	3.0	5.0	1.7	Iris-versicolor
78	6.0	2.9	4.5	1.5	Iris-versicolor
79	5.7	2.6	3.5	1.0	Iris-versicolor
80	5.5	2.4	3.8	1.1	Iris-versicolor
81	5.5	2.4	3.7	1.0	Iris-versicolor
82	5.8	2.7	3.9	1.2	Iris-versicolor
83	6.0	2.7	5.1	1.6	Iris-versicolor
84	5.4	3.0	4.5	1.5	Iris-versicolor
85	6.0	3.4	4.5	1.6	Iris-versicolor
86	6.7	3.1	4.7	1.5	Iris-versicolor
87	6.3	2.3	4.4	1.3	Iris-versicolor
88	5.6	3.0	4.1	1.3	Iris-versicolor
89	5.5	2.5	4.0	1.3	Iris-versicolor
90	5.5	2.6	4.4	1.2	Iris-versicolor
91	6.1	3.0	4.6	1.4	Iris-versicolor
92	5.8	2.6	4.0	1.2	Iris-versicolor
93	5.0	2.3	3.3	1.0	Iris-versicolor
94	5.6	2.7	4.2	1.3	Iris-versicolor
95	5.7	3.0	4.2	1.2	Iris-versicolor
96	5.7	2.9	4.2	1.3	Iris-versicolor
97	6.2	2.9	4.3	1.3	Iris-versicolor
98	5.1	2.5	3.0	1.1	Iris-versicolor
99	5.7	2.8	4.1	1.3	Iris-versicolor
100	6.3	3.3	6.0	2.5	Iris-virginica
101	5.8	2.7	5.1	1.9	Iris-virginica
102	7.1	3.0	5.9	2.1	Iris-virginica
103	6.3	2.9	5.6	1.8	Iris-virginica
104	6.5	3.0	5.8	2.2	Iris-virginica
105	7.6	3.0	6.6	2.1	Iris-virginica
106	4.9	2.5	4.5	1.7	Iris-virginica
107	7.3	2.9	6.3	1.8	Iris-virginica
108	6.7	2.5	5.8	1.8	Iris-virginica
109	7.2	3.6	6.1	2.5	Iris-virginica
110	6.5	3.2	5.1	2.0	Iris-virginica
111	6.4	2.7	5.3	1.9	Iris-virginica
112	6.8	3.0	5.5	2.1	Iris-virginica
113	5.7	2.5	5.0	2.0	Iris-virginica
114	5.8	2.8	5.1	2.4	Iris-virginica
115	6.4	3.2	5.3	2.3	Iris-virginica
116	6.5	3.0	5.5	1.8	Iris-virginica
117	7.7	3.8	6.7	2.2	Iris-virginica
118	7.7	2.6	6.9	2.3	Iris-virginica
119	6.0	2.2	5.0	1.5	Iris-virginica
120	6.9	3.2	5.7	2.3	Iris-virginica
121	5.6	2.8	4.9	2.0	Iris-virginica

122	7.7	2.8	6.7	2.0	Iris-virginica
123	6.3	2.7	4.9	1.8	Iris-virginica
124	6.7	3.3	5.7	2.1	Iris-virginica
125	7.2	3.2	6.0	1.8	Iris-virginica
126	6.2	2.8	4.8	1.8	Iris-virginica
127	6.1	3.0	4.9	1.8	Iris-virginica
128	6.4	2.8	5.6	2.1	Iris-virginica
129	7.2	3.0	5.8	1.6	Iris-virginica
130	7.4	2.8	6.1	1.9	Iris-virginica
131	7.9	3.8	6.4	2.0	Iris-virginica
132	6.4	2.8	5.6	2.2	Iris-virginica
133	6.3	2.8	5.1	1.5	Iris-virginica
134	6.1	2.6	5.6	1.4	Iris-virginica
135	7.7	3.0	6.1	2.3	Iris-virginica
136	6.3	3.4	5.6	2.4	Iris-virginica
137	6.4	3.1	5.5	1.8	Iris-virginica
138	6.0	3.0	4.8	1.8	Iris-virginica
139	6.9	3.1	5.4	2.1	Iris-virginica
140	6.7	3.1	5.6	2.4	Iris-virginica
141	6.9	3.1	5.1	2.3	Iris-virginica
142	5.8	2.7	5.1	1.9	Iris-virginica
143	6.8	3.2	5.9	2.3	Iris-virginica
144	6.7	3.3	5.7	2.5	Iris-virginica
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

4.3.2 Get data types

[8]:
```
irisdata_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 150 entries, 0 to 149
Data columns (total 5 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   sepal_length  150 non-null    float64
 1   sepal_width   150 non-null    float64
 2   petal_length  150 non-null    float64
 3   petal_width   150 non-null    float64
 4   species       150 non-null    object
dtypes: float64(4), object(1)
memory usage: 5.3+ KB
```

[9]:
```
irisdata_df.describe()
```

[9]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	5.843333	3.054000	3.758667	1.198667
std	0.828066	0.433594	1.764420	0.763161
min	4.300000	2.000000	1.000000	0.100000
25%	5.100000	2.800000	1.600000	0.300000
50%	5.800000	3.000000	4.350000	1.300000
75%	6.400000	3.300000	5.100000	1.800000

```
max         7.900000      4.400000      6.900000      2.500000
```

4.3.3 Get data ranges and distribution

Histograms This type of visualization is useful to explore the **frequency distribution** for each feature in univariate plots. This requires the separation of the data into classes (so-called **bins**). These bins are represented in the histogram as rectangles of equal or variable width. The height of each rectangle then represents the (relative or absolute) **frequency density**.

Each **feature** of the **Iris dataset** is displayed in its own histogram.

To illustrate the principle, the histogram subplots are first presented in a **not very elegant code** with many repetitions:

[358]:
```
# Number of bins for the histogram
n_bins = 10
fig, axs = plt.subplots(2, 2, figsize=(12, 10))
# Set margins between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.3)

axs[0,0].hist(irisdata_df['sepal_length'], bins = n_bins, rwidth=0.95)
axs[0,0].set_title('Sepal Length')
# Show grid
axs[0,0].grid(visible=True)
# Hide grid behind the bars
axs[0,0].set_axisbelow(True)
# Label x and y-axis
axs[0,0].set_xlabel('value range [cm]')
axs[0,0].set_ylabel('frequency density')

axs[0,1].hist(irisdata_df['sepal_width'], bins = n_bins, rwidth=0.95)
axs[0,1].set_title('Sepal Width')
# Show grid
axs[0,1].grid(visible=True)
# Hide grid behind the bars
axs[0,1].set_axisbelow(True)
# Label x and y-axis
axs[0,1].set_xlabel('value range [cm]')
axs[0,1].set_ylabel('frequency density')

axs[1,0].hist(irisdata_df['petal_length'], bins = n_bins, rwidth=0.95)
axs[1,0].set_title('Petal Length')
# Show grid
axs[1,0].grid(visible=True)
# Hide grid behind the bars
axs[1,0].set_axisbelow(True)
# Label x and y-axis
axs[1,0].set_xlabel('value range [cm]')
axs[1,0].set_ylabel('frequency density')

axs[1,1].hist(irisdata_df['petal_width'], bins = n_bins, rwidth=0.95)
axs[1,1].set_title('Petal Width')
# Show grid
axs[1,1].grid(visible=True)
# Hide grid behind the bars
axs[1,1].set_axisbelow(True)
# Label x and y-axis
axs[1,1].set_xlabel('value range [cm]')
```

```
axs[1,1].set_ylabel('frequency density')

plt.show()
```

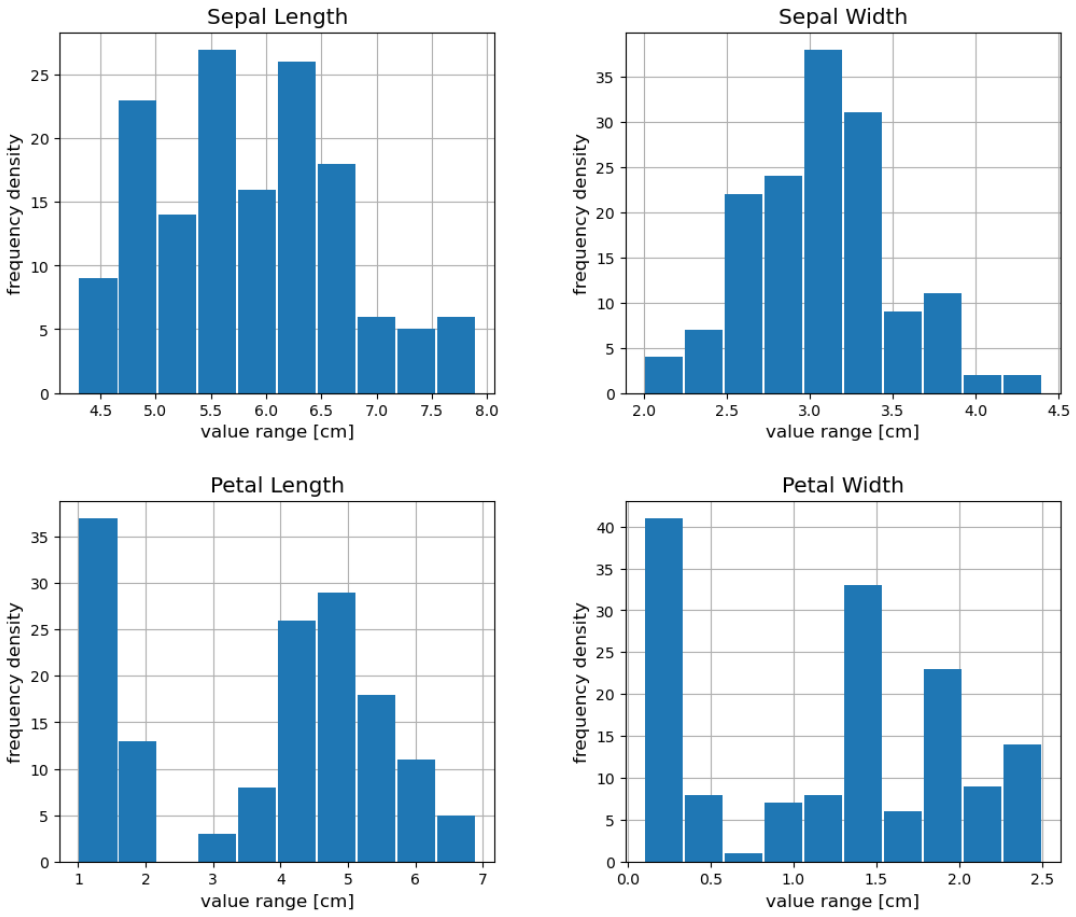


Figure 14: Histograms used to explore the frequency distribution of the 4 features in the Iris dataset

To improve the code, the function `subplots.flatten()` converts the subplot array to an iterable list. Afterwards, a loop allows to iterate through the subplots - this **saves many repetitions** in the code.

In addition, **probability density functions (PDF)** were overlaid on the histograms, whose hyper-parameters **mean** and **standard deviation** were previously identified using the features of the dataset. This makes it possible to estimate whether the **data is normally distributed**. In order to be able to reuse the code later, it was implemented as the **function** `func_plot_histograms_with_PDF()`.

[131]:
```
from scipy.stats import norm

def func_plot_histograms_with_PDF(df, features, titles):
    # Number of bins for the histogram
    # - bins=<integer>: defines the number of equal-width bins in the range
    # - bins=<string>: one of the binning strategies is used:
    #   'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'
    n_bins = 'auto'
    fig, subplots = plt.subplots(2, 2, figsize=(12, 10))
    # Set margins between subplots
    plt.subplots_adjust(wspace=0.3, hspace=0.3)

    # Make subplots iterable via 'subplots.flatten()'
```

```
    for feature, title, subplot in zip(features, titles, subplots.flatten()):
        subplot.hist(df[feature], bins = n_bins, rwidth=0.95,
                     density=True, alpha=0.8)

        # Fit a normal distribution to the data
        # with mean and standard deviation
        mu, std = norm.fit(df[feature])

        # Plot the probability density function (PDF)
        xmin, xmax = subplot.get_xlim()
        x = np.linspace(xmin, xmax, 100)
        p = norm.pdf(x, mu, std)
        subplot.plot(x, p, 'k', linewidth=2)

        title_concat = "{} (Mean: {:.2f}, " \
                       "Std. deviation: {:.2f})".format(title, mu, std)
        subplot.set_title(title_concat)
        # Show grid
        subplot.grid(visible=True)
        # Hide grid behind the bars
        subplot.set_axisbelow(True)
        # Label x and y-axis
        subplot.set_xlabel('value range [cm]')
        subplot.set_ylabel('frequency density')

    plt.show()
```

Call the new function to plot the **histograms** with overlaid **probability density functions**:

[132]:
```
features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
titles =   ['Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width']

func_plot_histograms_with_PDF(irisdata_df, features, titles)
```

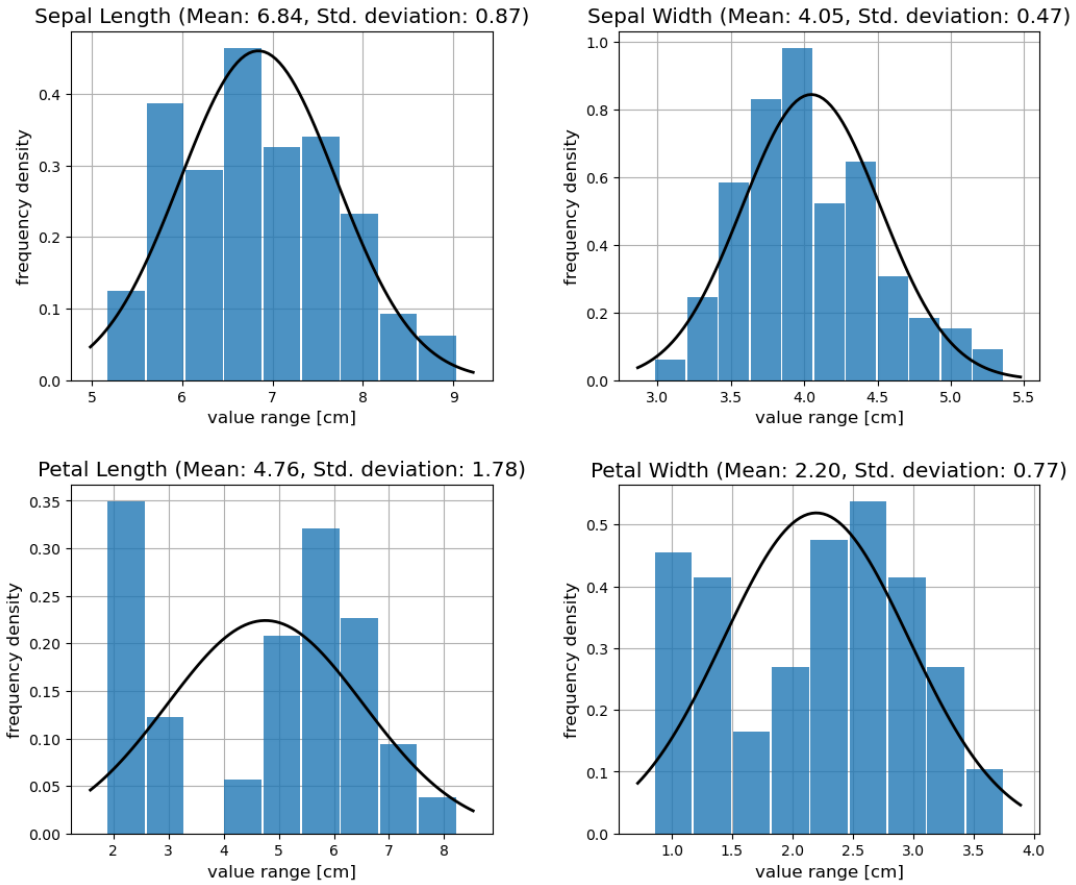


Figure 15: Histograms used to explore the frequency distribution of the 4 features in the Iris dataset (with improved code and overlaid probability density functions (PDF))

Boxplots This type of visualization can be used to explore the **data ranges** in the dataset. **Boxplots** also provide information about **outliers**.

In the following code example, the 4 variables of the Iris dataset are displayed side-by-side in individual boxplots. As in the previous histogram example, a loop is used to iterate through the subplots, which saves a lot of repetition in the code.

[137]:
```
fig, subplots = plt.subplots(2, 2, figsize=(12, 10))
# Set margins between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.3)

class_names = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
features =    ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

# Make subplots iterable via 'subplots.flatten()'
for feature, subplot in zip(features, subplots.flatten()):
    # x, y: names of features in dataset
    # data: dataset for plotting
    # order: order to plot the class names in
    # ax: assignment of the plot to the subplot
    sns.boxplot(x = 'species', y = feature,
                data = irisdata_df, order = class_names, ax = subplot)
    # Show grid
    subplot.grid(axis='y')
    # Hide grid behind the bars
    subplot.set_axisbelow(True)
```

```
    # Set title of subplot
    subplot.set_title('Feature: {}'.format(feature))
    # Label y-axis
    subplot.set_ylabel('value range [cm]')

plt.show()
```

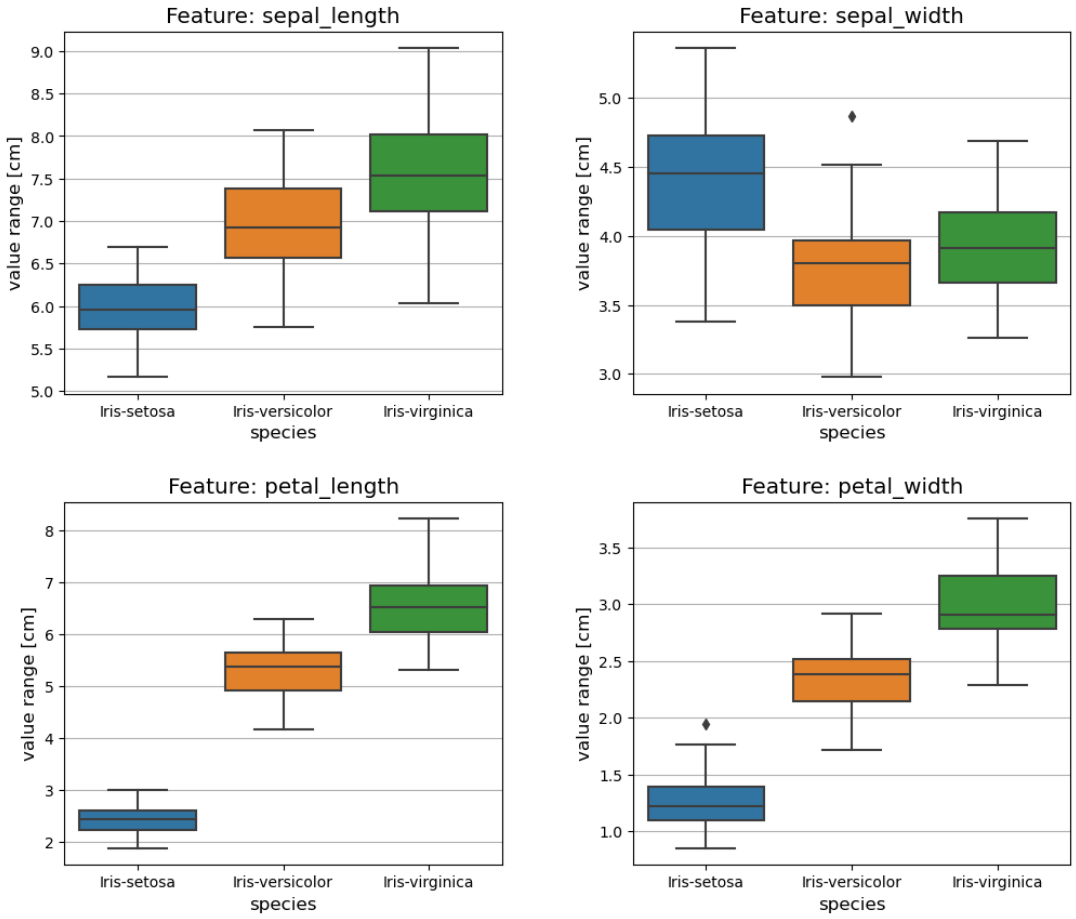


Figure 16: Boxplots used to explore the data ranges in the Iris dataset

Violin plots Another type of visualization is the **violin plot**, which **combines** the advantages of both the **histogram** and the **box plot**. As in the two previous examples, a loop is used to iterate through the subplots, which saves a lot of repetition in the code.

[138]:
```
fig, subplots = plt.subplots(2, 2, figsize=(12, 10))
# Set margins between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.3)

class_names = ['Iris-setosa', 'Iris-versicolor', 'Iris-virginica']
features =    ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']

# Make subplots iterable via 'subplots.flatten()'
for feature, subplot in zip(features, subplots.flatten()):
    # x, y: names of features in dataset
    # data: dataset for plotting
    # order: order to plot the class names in
    # ax: assignment of the plot to the subplot
```

```
    sns.violinplot(x = 'species', y = feature,
                   data = irisdata_df, order = class_names, ax = subplot)
    # Show grid
    subplot.grid(axis='y')
    # Hide grid behind the bars
    subplot.set_axisbelow(True)
    # Set title of subplot
    subplot.set_title('Feature: {}'.format(feature))
    # Label y-axis
    subplot.set_ylabel('value range [cm]')

plt.show()
```

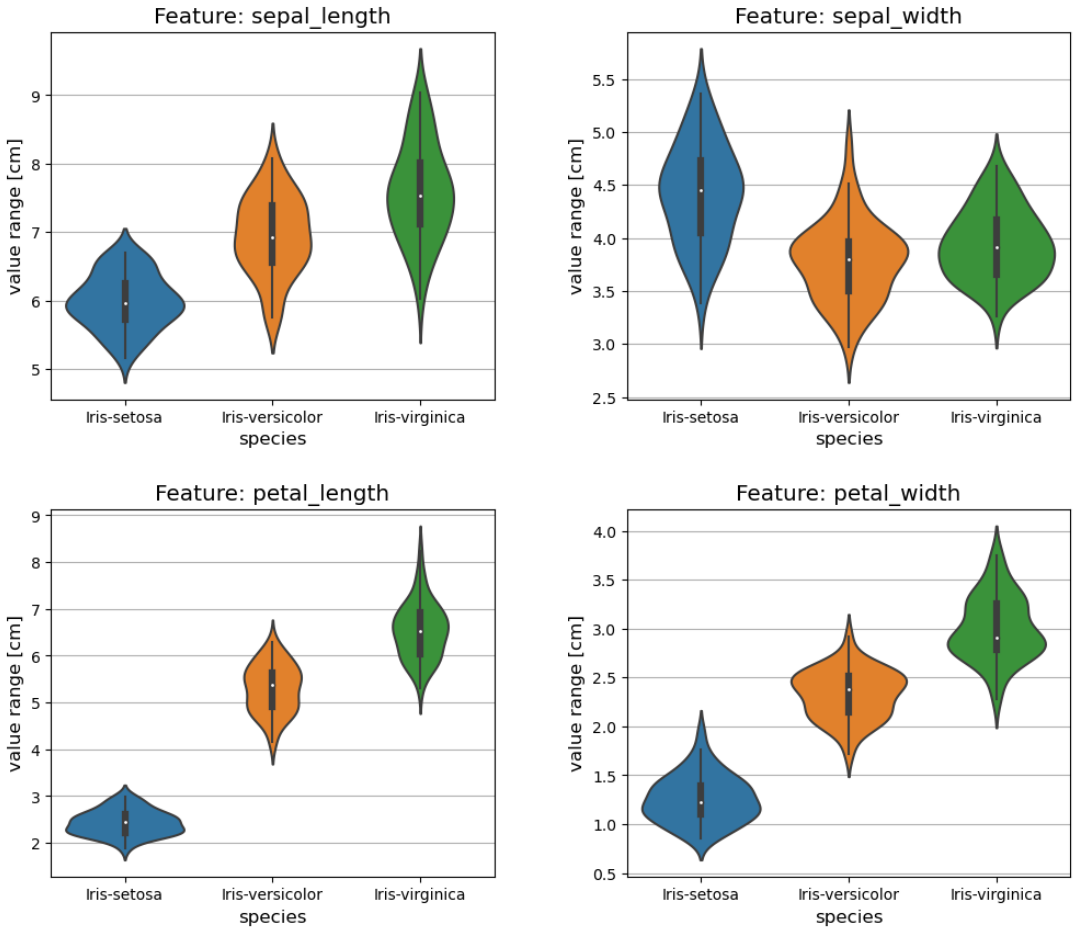


Figure 17: Violin plots combine histograms and box plots

4.4 Identify anomalies in the datasets

4.4.1 Find and repair gaps in dataset

This section was inspired by Working with Missing Data in Pandas.

Check for missing values using `isnull()` In order to check for missing values in a `pandas.DataFrame`, the function `isnull()` is used here. This function returns a dataframe of boolean values which are `True` for **NaN values**.

[118]:
```
pd.set_option('display.max_rows', 10)
pd.set_option('display.min_rows', 10)
```

[82]:
```
irisdata_df.isnull()
```

[82]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	False	False	False	False	False
1	False	False	False	False	False
2	False	False	False	False	False
3	False	False	False	False	False
4	False	False	False	False	False
..	…	…	…	…	…
145	False	False	False	False	False
146	False	False	False	False	False
147	False	False	False	False	False
148	False	False	False	False	False
149	False	False	False	False	False

[150 rows x 5 columns]

Show only the gaps:

[83]:
```
irisdata_df_gaps = irisdata_df[irisdata_df.isnull().any(axis=1)]
irisdata_df_gaps
```

[83]:
```
Empty DataFrame
Columns: [sepal_length, sepal_width, petal_length, petal_width, species]
Index: []
```

Fine - the **Iris dataset** seems to be **complete** :)

So let's look for **another dataset to exercise**. For this purpose, the original employees dataset, which will be used in the next subsections, has been **slightly modified**.

[84]:
```
# Import data to dataframe from CSV file
employees_df = pd.read_csv("./datasets/employees_edit.csv")
```

For the **before-and-after comparison**, the **edited** and the **original data frames** are connected with each other (so-called **merging**). This merging **requires unique identifiers** for the individual data records (rows of the data frame). Using the **index** of the data frame for this is far too **unreliable**, since it can **change constantly** due to reordering or the deletion and addition of rows.

Therefore, directly after importing the dataset from the CSV file, the dataframe **index** is **transferred to a new column** as a permanent and stable **records identifier**.

[85]:
```
# Retrieve indices of all rows into a temporary list
li_idx = employees_df.index

# Insert indices as a new index column at the first position with 'loc=0'
employees_df.insert(loc=0, column='idx', value=li_idx)

employees_df
```

[85]:

	idx	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	\
0	0	Douglas	Male	8/6/1993	12:42 PM	97308.0	6945.00	
1	1	Thomas	Male	3/31/1996	6:53 AM	61933.0	4.17	
2	2	Maria	Female	4/23/1993	11:17 AM	130590.0	11858.00	
3	3	Jerry	Male	3/4/2005	1:00 PM	138705.0	9.34	
4	4	Larry	Male	1/24/1998	4:47 PM	101004.0	1389.00	

```
…     …        …      …          …                …        …         …
999    999      Henry     NaN  11/23/2014          6:09 AM  132483.0  16655.00
1000  1000    Phillip    Male   1/31/1984          6:30 AM   42392.0  19675.00
1001  1001    Russell    Male   5/20/2013         12:39 PM   96914.0   1421.00
1002  1002      Larry    Male   4/20/2013          4:45 PM   60500.0  11985.00
1003  1003     Albert    Male   5/15/2012          6:24 PM  129949.0  10169.00

     Senior Management                 Team
0                 True            Marketing
1                 True                  NaN
2                False              Finance
3                 True              Finance
4                 True       Client Services
…                 …                  …
999               False          Distribution
1000              False              Finance
1001              False              Product
1002              False  Business Development
1003               True                Sales

[1004 rows x 9 columns]
```

Now a **deep copy** is created to preserve the **original data frame** for later **before-and-after comparison** - including the new index column to uniquely identify the records.

[86]:
```
employees_df_orig = employees_df.copy(deep=True)
```

[87]:
```
# Highlight cells with NaN values
# HINT: Set to 'False' when compiling to PDF!
highlight = False

if highlight:
    output = employees_df.style.highlight_null('yellow')
else:
    output = employees_df

output
```

[87]:
```
        idx First Name  Gender  Start Date Last Login Time    Salary   Bonus %  \
0         0    Douglas    Male    8/6/1993          12:42 PM   97308.0   6945.00
1         1     Thomas    Male   3/31/1996           6:53 AM   61933.0      4.17
2         2      Maria  Female   4/23/1993          11:17 AM  130590.0  11858.00
3         3      Jerry    Male    3/4/2005           1:00 PM  138705.0      9.34
4         4      Larry    Male   1/24/1998           4:47 PM  101004.0   1389.00
…     …        …      …          …                …        …         …
999    999      Henry     NaN  11/23/2014          6:09 AM  132483.0  16655.00
1000  1000    Phillip    Male   1/31/1984          6:30 AM   42392.0  19675.00
1001  1001    Russell    Male   5/20/2013         12:39 PM   96914.0   1421.00
1002  1002      Larry    Male   4/20/2013          4:45 PM   60500.0  11985.00
1003  1003     Albert    Male   5/15/2012          6:24 PM  129949.0  10169.00

     Senior Management                 Team
0                 True            Marketing
1                 True                  NaN
2                False              Finance
3                 True              Finance
4                 True       Client Services
```

```
…                …                    …
999              False          Distribution
1000             False               Finance
1001             False               Product
1002             False  Business Development
1003              True                 Sales

[1004 rows x 9 columns]
```

Show only the **gaps** (NaN values) from this incomplete dataset again:

[88]:
```
employees_df_gaps = employees_df[employees_df.isnull().any(axis=1)]

# Highlight cells with NaN values
# HINT: Set to 'False' when compiling to PDF!
highlight = False

if highlight:
    output = employees_df_gaps.style.highlight_null('yellow')
else:
    output = employees_df_gaps

output
```

[88]:
```
      idx First Name  Gender  Start Date Last Login Time    Salary   Bonus %  \
1       1     Thomas    Male   3/31/1996          6:53 AM   61933.0      4.17
7       7        NaN  Female   7/20/2015         10:43 AM   45906.0  11598.00
10     10     Louise  Female   8/12/1980          9:01 AM   63241.0  15132.00
17     17      Shawn    Male   12/7/1986          7:45 PM       NaN   6414.00
20     20       Lois     NaN   4/22/1995          7:18 PM   64714.0   4934.00
..    …        …     …         …                …        …       …
965   965    Antonio     NaN   6/18/1989          9:37 PM  103050.0      3.05
976   976     Victor     NaN   7/28/2006          2:49 PM   76381.0  11159.00
989   989    Stephen     NaN   7/10/1983          8:10 PM   85668.0   1909.00
993   993     Justin     NaN   2/10/1991          4:58 PM   38344.0   3794.00
999   999      Henry     NaN  11/23/2014          6:09 AM  132483.0  16655.00

    Senior Management          Team
1                True           NaN
7                 NaN       Finance
10               True           NaN
17              False       Product
20               True         Legal
..                …           …
965             False         Legal
976              True         Sales
989             False         Legal
993             False         Legal
999             False  Distribution

[240 rows x 9 columns]
```

Fill in missing *string* values with `fillna()` Now all null values (NaN) in the column "Gender" of the data type String are filled with *"No gender"*.

Warning: The following example replaces the strings directly in the original dataframe with `inplace = True` - no deep copy is made!

[89]:
```
# Fill all null values in column 'Gender' using fillna()
employees_df['Gender'].fillna('No Gender', inplace = True)

# Switch to apply highlight style to dataframe
# HINT: Set to 'False' when compiling to PDF!
highlight = False

# Show only rows with subtituted 'Gender' column
employees_df_filled_gender = employees_df[employees_df['Gender'] == 'No Gender']

if highlight:
    # Highlight cells by condition
    output = employees_df_filled_gender.style.apply(lambda x:
                                                    ['background: yellow'
                                                     if v == 'No Gender'
                                                     else "" for v in x],
                                                    axis = 1)
else:
    output = employees_df_filled_gender

output
```

[89]:

	idx	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	Senior Management	Team
20	20	Lois	No Gender	4/22/1995	7:18 PM	64714.0	4934.00	True	Legal
22	22	Joshua	No Gender	3/8/2012	1:58 AM	90816.0	18816.00	True	Client Services
27	27	Scott	No Gender	7/11/1991	6:58 PM	122367.0	5218.00	False	Legal
31	31	Joyce	No Gender	2/20/2005	2:40 PM	88657.0	12752.00	False	Product
41	41	Christine	No Gender	6/28/2015	1:08 AM	66582.0	11308.00	True	Business Development
..	…	…	…	…	…	…	…	…	…
965	965	Antonio	No Gender	6/18/1989	9:37 PM	103050.0	3.05	False	Legal
976	976	Victor	No Gender	7/28/2006	2:49 PM	76381.0	11159.00	True	Sales
989	989	Stephen	No Gender	7/10/1983	8:10 PM	85668.0	1909.00	False	Legal
993	993	Justin	No Gender	2/10/1991	4:58 PM	38344.0	3794.00	False	Legal
999	999	Henry	No Gender	11/23/2014	6:09 AM	132483.0	16655.00	False	Distribution

[146 rows x 9 columns]

Fill in missing *numerical* values with median values Missing integer or float values can be filled with the **mean** or **median values** of the corresponding feature column.

Often, **removing individual rows** in the dataset or even entire feature columns is **impractical** because too much valuable data would be lost. In this case, various **interpolation procedures** can be used to estimate the missing values based on the other values present in the data set. The most common interpolation methods are **mean** and **median imputation**, where missing values are replaced by the

mean or **median** of the entire feature column (see Raschka and Mirjalili 2018).

Further information on the subject of **imputation** can be found here, among other places:

- Pandas: How to Fill NaN Values with Median (3 Examples)
- pandas DataFrame: replace nan values with average of columns
- scikit-learn: Imputation of missing values

Show rows with missing salary only:

[90]:
```
employees_df_gaps = employees_df[employees_df['Salary'].isnull()]
employees_df_gaps
```

[90]:
```
    idx First Name  Gender Start Date Last Login Time  Salary  Bonus %  \
17   17      Shawn    Male  12/7/1986         7:45 PM     NaN  6414.00
63   63    Matthew    Male   1/2/2013        10:33 PM     NaN    18.04
76   76   Margaret  Female  9/10/1988        12:42 PM     NaN  7353.00

   Senior Management            Team
17              False         Product
63              False  Human Resources
76               True     Distribution
```

As will be shown later in the section Display Histogram, the **salary structure depends** very much **on gender**. In order **not to distort the dataset** too much, the **median salaries** are determined **gender-specifically**.

First, the **male** and **female employees** with missing salary records are filtered:

[91]:
```
# Filter MALE employees with missing salary records
employees_df_gaps = employees_df.loc[(employees_df['Gender'] == 'Male') \
                                     & employees_df['Salary'].isnull()]

# Get indices of incomplete rows
index_list_male_salary_nan = employees_df_gaps.index.to_list()
index_list_male_salary_nan
```

[91]:
```
[17, 63]
```

[92]:
```
# Filter FEMALE employees with missing salary records
employees_df_gaps = employees_df.loc[(employees_df['Gender'] == 'Female') \
                                     & employees_df['Salary'].isnull()]

# Get indices of incomplete rows
index_list_female_salary_nan = employees_df_gaps.index.to_list()
index_list_female_salary_nan
```

[92]:
```
[76]
```

Get **median salaries** gender-specifically:

[93]:
```
# Get median salary for MALE employees
employees_df_male = employees_df.loc[employees_df['Gender'] == 'Male']

salary_median_male = employees_df_male['Salary'].median()
salary_median_male
```

[93]:
```
90370.0
```

[94]:
```
# Get median salary for FEMALE employees
employees_df_female = employees_df.loc[employees_df['Gender'] == 'Female']

salary_median_female = employees_df_female['Salary'].median()
salary_median_female
```

[94]:
```
90032.5
```

Fill missing salary records with the **gender-specifically median** of the salary column and replace NaN values in the original dataframe:

[95]:
```
# Fill missing salary records by MALE indices
employees_df.loc[index_list_male_salary_nan, 'Salary'] = salary_median_male

employees_df.loc[index_list_male_salary_nan]
```

[95]:
```
     idx First Name Gender Start Date Last Login Time   Salary  Bonus %  \
17    17      Shawn   Male  12/7/1986         7:45 PM  90370.0  6414.00
63    63    Matthew   Male   1/2/2013        10:33 PM  90370.0    18.04

   Senior Management            Team
17              False         Product
63              False  Human Resources
```

[96]:
```
# Fill missing salary records by FEMALE indices
employees_df.loc[index_list_female_salary_nan, 'Salary'] = salary_median_female

employees_df.loc[index_list_female_salary_nan]
```

[96]:
```
     idx First Name  Gender Start Date Last Login Time   Salary  Bonus %  \
76    76   Margaret  Female  9/10/1988        12:42 PM  90032.5   7353.0

   Senior Management          Team
76              True  Distribution
```

Merge male and female index lists and **extend** the indices by its direct **neighbors** for later **before-and-after comparison**:

[97]:
```
# Function for retrieving the indices of previous and next rows of a dataframe
def get_previous_and_next_rows_from_df(df, search_idx):
    # Get list of indices of dataframe rows
    index_list = df.index.to_list()

    li_elem_df_prev = 0
    li_elem_df_next = 0
    b_element_found = False

    # Cycle over list of indices of given dataframe
    for li_idx_df, li_elem_df in enumerate(index_list):
        # Check index bounds
        if (li_idx_df+1 < len(index_list) and li_idx_df-1 >= 0):

            # Get previous and successing list elements
            if li_elem_df == search_idx:
                b_element_found = True
                li_elem_df_prev = index_list[li_idx_df-1]
                li_elem_df_next = index_list[li_idx_df+1]
```

```
    if b_element_found:
        #print(li_elem_df_prev, search_idx, li_elem_df_next)
        return [li_elem_df_prev, search_idx, li_elem_df_next]
    else:
        print('Element was not found :(')
        return []
```

Generate a **combined index list** of males and females with the missing salary records as well as their respective direct **gender neighbors** for comparison:

[98]:
```
# Merge MALE and FEMALE index lists
index_list_merged = []

# Filter MALE employees
employees_df_male = employees_df.loc[(employees_df['Gender'] == 'Male')]

# Cycle over list of MALE employees with missing salary records
for li_idx_salary_nan in index_list_male_salary_nan:
    li_neighbors_male = get_previous_and_next_rows_from_df(employees_df_male,
                                                           li_idx_salary_nan)
    index_list_merged.extend(li_neighbors_male)

# Filter FEMALE employees
employees_df_female = employees_df.loc[(employees_df['Gender'] == 'Female')]

# Cycle over list of FEMALE employees with missing salary records
for li_idx_salary_nan in index_list_female_salary_nan:
    li_neighbors_female = get_previous_and_next_rows_from_df(employees_df_female,
                                                           li_idx_salary_nan)
    index_list_merged.extend(li_neighbors_female)

index_list_merged
```

[98]:
```
[16, 17, 21, 57, 63, 65, 75, 76, 78]
```

Show rows with replaced `salary` by extended index list for both genders (including the direct neighbors) for **before-and-after comparison**.

[99]:
```
# Switch to apply highlight style to dataframe
# HINT: Set to 'False' when compiling to PDF!
highlight = False

# Show rows with replaced 'salary' by index and its direct neighbors
employees_df_filled_salary = employees_df.iloc[index_list_merged]

if highlight:
    # Highlight cells by condition
    output = employees_df_filled_salary.style.apply(lambda x:
                                        ['background: yellow'
                                         if (v == salary_median_male
                                             or v == salary_median_female)
                                         else "" for v in x],
                                        axis = 1)
else:
    output = employees_df_filled_salary

output
```

[99]:
```
     idx First Name  Gender  Start Date Last Login Time    Salary   Bonus %  \
16    16     Jeremy    Male   9/21/2010         5:56 AM   90370.0   7369.00
17    17      Shawn    Male   12/7/1986         7:45 PM   90370.0   6414.00
21    21    Matthew    Male    9/5/1995         2:12 AM  100612.0  13645.00
57    57      Henry    Male   6/26/1996         1:44 AM   64715.0  15107.00
63    63    Matthew    Male    1/2/2013        10:33 PM   90370.0     18.04
65    65      Steve    Male  11/11/2009        11:44 PM   61310.0  12428.00
75    75     Bonnie  Female    7/2/1991         1:27 AM  104897.0   5118.00
76    76   Margaret  Female   9/10/1988        12:42 PM   90032.5   7353.00
78    78      Robin  Female    6/4/1983         3:15 PM  114797.0   5965.00

   Senior Management             Team
16             False  Human Resources
17             False          Product
21             False        Marketing
57              True  Human Resources
63             False  Human Resources
65              True     Distribution
75              True  Human Resources
76              True     Distribution
78              True            Sales
```

By pure coincidence, the **predecessor** of `Shawn` with the missing salary record has the **median male salary** of the entire dataset. Therefore, this value is also highlighted.

Drop missing values using `dropna()` In order to drop null values from a dataframe, we use `dropna()` function. This function drops rows or columns of datasets with NaN values in different ways.

Default is to drop rows with at least 1 null value (NaN). Giving the parameter `how = 'all'` the function drops rows with all data missing or contain null values (NaN).

Warning: The following example drops rows with missing values (NaN) directly in the original dataframe with `inplace = True` - no deep copy is made!

[100]:
```
# Drop rows with missing values directly in the original dataframe
employees_df.dropna(axis = 0, how ='any', inplace = True)
#employees_df
```

Finally we compare the sizes of dataframes so that we learn how many rows had at least 1 Null value.

[101]:
```
print("Original dataframe length:", len(employees_df_orig))
print("New dataframe length:", len(employees_df))
print("Number of rows with at least 1 NaN value: ",
      (len(employees_df_orig)-len(employees_df)))
```

```
Original dataframe length: 1004
New dataframe length: 903
Number of rows with at least 1 NaN value:  101
```

4.4.2 Find and remove duplicates in dataset

This section was inspired by:

- How to Find Duplicates in Pandas DataFrame (With Examples)
- How to Drop Duplicate Rows in a Pandas DataFrame

Check for duplicate values using `duplicated()` In order to check for duplicate values in `pandas.DataFrame`, we use a function `duplicated()`. This function can be used in two ways:

- find duplicate rows across **all columns** with `df.duplicated()`
- find duplicate rows across **specific columns** with parameter `subset=['col1', 'col2']`
- mark last duplicates for removing and **keep the first occurrences** with parameter `keep='first'`
- mark first duplicates for removing and **keep the last occurrences** with parameter `keep='last'`
- **mark all duplicates** for removing with parameter `keep=False`

Find duplicate rows across **all columns**:

[119]:
```
# Find duplicate rows across all columns
# The column 'idx' has to be ignored
column_subset = employees_df.columns.difference(['idx'])
duplicateRows = employees_df[employees_df.duplicated(subset=column_subset)]
duplicateRows
```

[119]:
```
Empty DataFrame
Columns: [idx, First Name, Gender, Start Date, Last Login Time, Salary, Bonus %,
Senior Management, Team]
Index: []
```

Find **all completely identical duplicates** (first and last occurrences). The resulting dataframe is **sorted by column** `'First Name'` to get the **duplicates grouped**:

[103]:
```
# Parameter 'keep=False' displays all duplicate rows
duplicateRows = employees_df[employees_df.duplicated(keep=False,␣
 ↪subset=column_subset)]

# Sort rows by column 'First Name' to get the duplicates grouped
duplicateRows.sort_values('First Name')
```

[103]:

	idx	First Name	Gender	Start Date	Last Login Time	Salary	Bonus %	\
153	153	Brandon	No Gender	11/3/1997	8:17 PM	121333.0	15295.0	
296	296	Brandon	No Gender	11/3/1997	8:17 PM	121333.0	15295.0	
55	55	Karen	Female	11/30/1999	7:46 AM	102488.0	17653.0	
112	112	Karen	Female	11/30/1999	7:46 AM	102488.0	17653.0	
92	92	Linda	Female	5/25/2000	5:45 PM	119009.0	12506.0	
127	127	Linda	Female	5/25/2000	5:45 PM	119009.0	12506.0	
442	442	Nicholas	Male	3/1/2013	9:26 PM	101036.0	2826.0	
580	580	Nicholas	Male	3/1/2013	9:26 PM	101036.0	2826.0	

	Senior Management	Team
153	False	Business Development
296	False	Business Development
55	True	Product
112	True	Product
92	True	Business Development
127	True	Business Development
442	True	Human Resources
580	True	Human Resources

Find **all duplicates** (first and last occurrences) across **specific columns**. The resulting dataframe is **sorted by multiple columns** `'First Name'` and `'Last Login Time'` to get the **duplicates grouped**:

[104]:
```
# Parameter 'keep=False' displays all duplicate rows
duplicateRows = employees_df[employees_df.duplicated(
                    subset=['First Name', 'Last Login Time'], keep=False)]

# Sort rows by column 'First Name' to get the duplicates grouped
duplicateRows.sort_values(['First Name', 'Last Login Time'])
```

[104]:
```
      idx First Name     Gender  Start Date Last Login Time    Salary  Bonus %  \
153   153    Brandon  No Gender   11/3/1997         8:17 PM  121333.0  15295.0
296   296    Brandon  No Gender   11/3/1997         8:17 PM  121333.0  15295.0
55     55      Karen     Female  11/30/1999         7:46 AM  102488.0  17653.0
112   112      Karen     Female  11/30/1999         7:46 AM  102488.0  17653.0
92     92      Linda     Female   5/25/2000         5:45 PM  119009.0  12506.0
..    …        …         …          …                …       …         …
973   973      Linda     Female    2/4/2010         8:49 PM   44486.0  17308.0
66     66      Nancy     Female  12/15/2012        11:57 PM  125250.0   2672.0
934   934      Nancy     Female   9/10/2001        11:57 PM   85213.0   2386.0
442   442   Nicholas       Male    3/1/2013         9:26 PM  101036.0   2826.0
580   580   Nicholas       Male    3/1/2013         9:26 PM  101036.0   2826.0

     Senior Management                 Team
153              False  Business Development
296              False  Business Development
55                True              Product
112               True              Product
92                True  Business Development
..                 …                …
973               True          Engineering
66                True  Business Development
934               True            Marketing
442               True      Human Resources
580               True      Human Resources

[12 rows x 9 columns]
```

Drop duplicate values using `drop_duplicates()` In order to drop duplicate values from a dataframe, we use `drop_duplicates()` function.

This function can be used in two ways:

- remove duplicate rows across **all columns** with `df.drop_duplicates()`
- remove duplicate rows across **specific columns** with parameter `subset=['col1', 'col2']`

Warning: The following example replaces the strings directly in the original dataframe with `inplace = True` - no deep copy is made!

Remove duplicate rows across **all columns**:

[105]:
```
# Remove duplicate rows across all columns
# The column 'idx' has to be ignored
column_subset = employees_df.columns.difference(['idx'])
employees_df.drop_duplicates(inplace=True, subset=column_subset)
employees_df
```

[105]:
```
       idx First Name     Gender  Start Date Last Login Time    Salary  \
0        0    Douglas       Male    8/6/1993        12:42 PM   97308.0
2        2      Maria     Female   4/23/1993        11:17 AM  130590.0
3        3      Jerry       Male    3/4/2005         1:00 PM  138705.0
4        4      Larry       Male   1/24/1998         4:47 PM  101004.0
5        5     Dennis       Male   4/18/1987         1:35 AM  115163.0
…    …        …         …          …                …       …
999    999      Henry  No Gender  11/23/2014         6:09 AM  132483.0
1000  1000    Phillip       Male   1/31/1984         6:30 AM   42392.0
1001  1001    Russell       Male   5/20/2013        12:39 PM   96914.0
1002  1002      Larry       Male   4/20/2013         4:45 PM   60500.0
1003  1003     Albert       Male   5/15/2012         6:24 PM  129949.0
```

```
       Bonus % Senior Management                 Team
0      6945.00              True            Marketing
2     11858.00             False              Finance
3         9.34              True              Finance
4      1389.00              True       Client Services
5     10125.00             False                Legal
…         …                 …                  …
999   16655.00             False         Distribution
1000  19675.00             False              Finance
1001   1421.00             False              Product
1002  11985.00             False  Business Development
1003  10169.00              True                Sales

[899 rows x 9 columns]
```

Remove duplicate rows across **specific columns**:

[106]:
```
# Remove duplicate rows across 'First Name' and 'Last Login Time' columns
employees_df.drop_duplicates(
    subset=['First Name', 'Last Login Time'], keep='last', inplace=True)
employees_df
```

[106]:
```
       idx First Name     Gender  Start Date Last Login Time    Salary  \
0        0    Douglas       Male    8/6/1993        12:42 PM   97308.0
2        2      Maria     Female   4/23/1993        11:17 AM  130590.0
3        3      Jerry       Male    3/4/2005         1:00 PM  138705.0
4        4      Larry       Male   1/24/1998         4:47 PM  101004.0
5        5     Dennis       Male   4/18/1987         1:35 AM  115163.0
…    …         …         …          …                …       …
999    999      Henry  No Gender  11/23/2014         6:09 AM  132483.0
1000  1000    Phillip       Male   1/31/1984         6:30 AM   42392.0
1001  1001    Russell       Male   5/20/2013        12:39 PM   96914.0
1002  1002      Larry       Male   4/20/2013         4:45 PM   60500.0
1003  1003     Albert       Male   5/15/2012         6:24 PM  129949.0

       Bonus % Senior Management                 Team
0      6945.00              True            Marketing
2     11858.00             False              Finance
3         9.34              True              Finance
4      1389.00              True       Client Services
5     10125.00             False                Legal
…         …                 …                  …
999   16655.00             False         Distribution
1000  19675.00             False              Finance
1001   1421.00             False              Product
1002  11985.00             False  Business Development
1003  10169.00              True                Sales

[897 rows x 9 columns]
```

4.4.3 Compare the edited dataset with the original dataset side-by-side

In the previous sections, the dataframe holding the **employees dataset** was **heavily edited** by **adding missing values** (where it was appropriate) or **deleting gapped rows** completely. Therefore, the **modifications made** to the dataset should be finally **checked**.

The `Pandas` package provides the `compare()` function, which can be used to **compare dataframes** and

display differences (see here: pandas.DataFrame.compare). Unfortunately, the documentation points out that this function can **only** be used to **compare dataframes** with the **same shape** (number of columns and rows) and **identical row and column labels**.

Among other things, a lot of **gapped rows have been removed** due to the heavy editing. Therefore, the original and the edited dataframe are **anything but identical** in terms of their **number of rows**. As expected, the following short test ends with a **long error message**.

[120]:
```
# Pandas.compare() does not work with dropped rows in one dataframe!
#employees_df_orig.compare(employees_df, keep_shape=False, keep_equal=True)
```

Therefore, in this subsection, a **customized method** of **comparing two dataframes** that are **not identical** in terms of **shape** has been developed.

The following approach shows one way to **compare** the **modified dataframe** with the **original dataframe** - despite the different number of rows. This is done by **merging both dataframes** into one, displaying the **same columns** of both datasets **side-by-side** and highlighting the **changes**.

The following sources were the inspiration for this subsection:

- Compare two DataFrames and output their differences side-by-side
- pandas compare two data frames and highlight the differences
- Highlighting a particular cell of a DataFrame in Pandas
- How to Compare Two Pandas DataFrames and Get Differences
- How to highlight differences between the two data frames in pandas
- pandas.DataFrame.compare
- Pandas Table Visualization
- 10 Examples to Master Pandas Styler
- Style Pandas Dataframe Like a Master
- A Quick and Easy Guide to Conditional Formatting in Pandas

[108]:
```
# Set the number of rows to output to default values
pd.set_option('display.min_rows', 10)
pd.set_option('display.max_rows', 10)
```

First, the **original** and the **modified dataframes** are **merged** into a new dataframe, using the **index column** `idx` to **synchronize** the two dataframes.

[109]:
```
employees_df_merged = pd.merge(employees_df_orig,
                               employees_df,
                               how='left', on=['idx'])
employees_df_merged
```

[109]:

	idx	First Name_x	Gender_x	Start Date_x	Last Login Time_x	Salary_x	\
0	0	Douglas	Male	8/6/1993	12:42 PM	97308.0	
1	1	Thomas	Male	3/31/1996	6:53 AM	61933.0	
2	2	Maria	Female	4/23/1993	11:17 AM	130590.0	
3	3	Jerry	Male	3/4/2005	1:00 PM	138705.0	
4	4	Larry	Male	1/24/1998	4:47 PM	101004.0	
…	…	…	…	…	…	…	
999	999	Henry	NaN	11/23/2014	6:09 AM	132483.0	
1000	1000	Phillip	Male	1/31/1984	6:30 AM	42392.0	
1001	1001	Russell	Male	5/20/2013	12:39 PM	96914.0	
1002	1002	Larry	Male	4/20/2013	4:45 PM	60500.0	
1003	1003	Albert	Male	5/15/2012	6:24 PM	129949.0	

	Bonus %_x	Senior Management_x	Team_x	First Name_y	\
0	6945.00	True	Marketing	Douglas	
1	4.17	True	NaN	NaN	
2	11858.00	False	Finance	Maria	

```
3          9.34                True                Finance        Jerry
4       1389.00                True        Client Services        Larry
…          …                    …                      …          …
999    16655.00               False           Distribution        Henry
1000   19675.00               False                Finance      Phillip
1001    1421.00               False                Product      Russell
1002   11985.00               False   Business Development        Larry
1003   10169.00                True                  Sales       Albert

       Gender_y Start Date_y Last Login Time_y  Salary_y  Bonus %_y  \
0          Male     8/6/1993          12:42 PM   97308.0    6945.00
1           NaN          NaN               NaN       NaN        NaN
2        Female    4/23/1993          11:17 AM  130590.0   11858.00
3          Male     3/4/2005           1:00 PM  138705.0       9.34
4          Male    1/24/1998           4:47 PM  101004.0    1389.00
…          …          …                  …       …          …
999   No Gender   11/23/2014           6:09 AM  132483.0   16655.00
1000       Male    1/31/1984           6:30 AM   42392.0   19675.00
1001       Male    5/20/2013          12:39 PM   96914.0    1421.00
1002       Male    4/20/2013           4:45 PM   60500.0   11985.00
1003       Male    5/15/2012           6:24 PM  129949.0   10169.00

     Senior Management_y               Team_y
0                   True            Marketing
1                    NaN                  NaN
2                  False              Finance
3                   True              Finance
4                   True       Client Services
…                   …                  …
999                False          Distribution
1000               False               Finance
1001               False               Product
1002               False  Business Development
1003                True                 Sales

[1004 rows x 17 columns]
```

The **column suffixes** automatically added during the merge operation are **renamed** to `_o` (original) and `_e` (edited dataframe) using **lambda inline functions**.

[110]:
```
employees_df_merged.rename(columns=lambda x: x.replace('_x', '_o'), inplace=True)
employees_df_merged.rename(columns=lambda x: x.replace('_y', '_e'), inplace=True)
employees_df_merged
```

[110]:
```
       idx First Name_o Gender_o Start Date_o Last Login Time_o  Salary_o  \
0        0      Douglas     Male     8/6/1993          12:42 PM   97308.0
1        1       Thomas     Male    3/31/1996           6:53 AM   61933.0
2        2        Maria   Female    4/23/1993          11:17 AM  130590.0
3        3        Jerry     Male     3/4/2005           1:00 PM  138705.0
4        4        Larry     Male    1/24/1998           4:47 PM  101004.0
…      …          …       …          …                  …       …
999    999        Henry      NaN   11/23/2014           6:09 AM  132483.0
1000  1000      Phillip     Male    1/31/1984           6:30 AM   42392.0
1001  1001      Russell     Male    5/20/2013          12:39 PM   96914.0
1002  1002        Larry     Male    4/20/2013           4:45 PM   60500.0
1003  1003       Albert     Male    5/15/2012           6:24 PM  129949.0

      Bonus %_o Senior Management_o               Team_o First Name_e  \
```

```
0       6945.00                True              Marketing      Douglas
1          4.17                True                    NaN          NaN
2      11858.00               False                Finance        Maria
3          9.34                True                Finance        Jerry
4       1389.00                True         Client Services        Larry
…          …                    …                      …           …
999    16655.00               False           Distribution        Henry
1000   19675.00               False                Finance      Phillip
1001    1421.00               False                Product      Russell
1002   11985.00               False  Business Development        Larry
1003   10169.00                True                  Sales       Albert

       Gender_e Start Date_e Last Login Time_e  Salary_e  Bonus %_e  \
0          Male     8/6/1993          12:42 PM   97308.0    6945.00
1           NaN          NaN               NaN       NaN        NaN
2        Female    4/23/1993          11:17 AM  130590.0   11858.00
3          Male     3/4/2005           1:00 PM  138705.0       9.34
4          Male    1/24/1998           4:47 PM  101004.0    1389.00
…          …          …                …       …        …
999   No Gender   11/23/2014           6:09 AM  132483.0   16655.00
1000       Male    1/31/1984           6:30 AM   42392.0   19675.00
1001       Male    5/20/2013          12:39 PM   96914.0    1421.00
1002       Male    4/20/2013           4:45 PM   60500.0   11985.00
1003       Male    5/15/2012           6:24 PM  129949.0   10169.00

     Senior Management_e               Team_e
0                   True            Marketing
1                    NaN                  NaN
2                  False              Finance
3                   True              Finance
4                   True       Client Services
…                   …                  …
999                False          Distribution
1000               False              Finance
1001               False              Product
1002               False  Business Development
1003                True                Sales

[1004 rows x 17 columns]
```

In order to have the **columns** of the original and the edited dataframe directly **next to each other** for a **better comparison**, a **column list** is created with the **new order**.

[111]:
```
li_reordered_cols = ['idx']

# Iterate over columns
for column in employees_df_orig.columns:
    if column != 'idx':
        li_reordered_cols.append(column + '_o')
        li_reordered_cols.append(column + '_e')

li_reordered_cols
```

[111]:
```
['idx',
 'First Name_o',
 'First Name_e',
 'Gender_o',
 'Gender_e',
```

```
 'Start Date_o',
 'Start Date_e',
 'Last Login Time_o',
 'Last Login Time_e',
 'Salary_o',
 'Salary_e',
 'Bonus %_o',
 'Bonus %_e',
 'Senior Management_o',
 'Senior Management_e',
 'Team_o',
 'Team_e']
```

In order to have **columns with the same meaning next to each other**, the columns of the merged data frame are **re-sorted** based on the new **column list**.

[112]:
```
employees_df_merged = employees_df_merged.reindex(columns=li_reordered_cols)
employees_df_merged
```

[112]:

	idx	First Name_o	First Name_e	Gender_o	Gender_e	Start Date_o	Start Date_e	Last Login Time_o	Last Login Time_e	Salary_o	Salary_e	Bonus %_o	Bonus %_e	Senior Management_o	Senior Management_e	Team_o	Team_e
0	0	Douglas	Douglas	Male	Male	8/6/1993	8/6/1993	12:42 PM	12:42 PM	97308.0	97308.0	6945.00	6945.00	True	True	Marketing	Marketing
1	1	Thomas	NaN	Male	NaN	3/31/1996	NaN	6:53 AM	NaN	61933.0	NaN	4.17	NaN	True	NaN	Finance	NaN
2	2	Maria	Maria	Female	Female	4/23/1993	4/23/1993	11:17 AM	11:17 AM	130590.0	130590.0	11858.00	11858.00	False	False	Finance	Finance
3	3	Jerry	Jerry	Male	Male	3/4/2005	3/4/2005	1:00 PM	1:00 PM	138705.0	138705.0	9.34	9.34	True	True	Finance	Finance
4	4	Larry	Larry	Male	Male	1/24/1998	1/24/1998	4:47 PM	4:47 PM	101004.0	101004.0	1389.00	1389.00	True	True	IT	IT
…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…	…
999	999	Henry	Henry	NaN	No Gender	11/23/2014	11/23/2014	6:09 AM	6:09 AM	132483.0	132483.0	16655.00	16655.00	False	False	Marketing	Marketing
1000	1000	Phillip	Phillip	Male	Male	1/31/1984	1/31/1984	6:30 AM	6:30 AM	42392.0	42392.0	19675.00	19675.00	False	False	Business Development	Business Development
1001	1001	Russell	Russell	Male	Male	5/20/2013	5/20/2013	12:39 PM	12:39 PM	96914.0	96914.0	1421.00	1421.00	False	False	Product	Product
1002	1002	Larry	Larry	Male	Male	4/20/2013	4/20/2013	4:45 PM	4:45 PM	60500.0	60500.0	11985.00	11985.00	False	False	Business Development	Business Development
1003	1003	Albert	Albert	Male	Male	5/15/2012	5/15/2012	6:24 PM	6:24 PM	129949.0	129949.0	10169.00	10169.00	True	True	Sales	Sales

```
                  Team_o               Team_e
0              Marketing            Marketing
1                    NaN                  NaN
2                Finance              Finance
3                Finance              Finance
4         Client Services      Client Services
…                    …                    …
999          Distribution         Distribution
1000             Finance              Finance
1001             Product              Product
1002  Business Development  Business Development
1003               Sales                Sales

[1004 rows x 17 columns]
```

To avoid problems with the subsequent comparison, **missing values** in the cells are **replaced** with the string `NaN` using the function `fillna()`.

[113]:
```
employees_df_merged.fillna('NaN', inplace = True)
```

For a better overview, only the **rows** that show **differences in the individual cells** will be displayed during the comparison. For this purpose, the new, empty dataframe `employees_df_diff` is created with the same column labels as the original dataframe `employees_df_orig`.

[114]:
```
employees_df_diff = pd.DataFrame(columns=employees_df_orig.columns)
```

The function `dataframe_add_row()` is used to easily **add rows** to a **dataframe**.

[115]:
```
def dataframe_add_row(df=None, row=[]):
    if (df is None):
        return

    # Add a row to dataframe
    df.loc[-1] = row

    # Shift the index of the dataframe
    df.index = df.index + 1

    # Reset the index of dataframe and
    # avoid the old index being added as a column
    df.reset_index(drop=True, inplace=True)
```

Now the merged dataframe `employees_df_merged` is **iterated** through **row by row** and **column by column** in a **nested loop**. In the inner loop, the **original cell value** (column with suffix `_o`) is **compared** with the adjacent **edited cell value** (column with suffix `_e`).

If a **difference** is detected, the **original** and the **edited cell value** is written into a **common new cell** - marked with the **difference symbol** `<=>`. Then, only the **rows containing differences** in the cells are **added** to the `employees_df_diff` **dataframe**.

[116]:
```
b_diffs_found = False

# Iterate over rows
for rowIndex, row in employees_df_merged.iterrows():
    # Iterate over columns
    for column, value in row.items():
        # Omit index column for comparison
        if column == 'idx':
            row_li = [value]
```

```
            continue
        if column.endswith('_o'):
            value_orig = value
        elif column.endswith('_e'):
            value_edit = value
            if value_orig != value_edit:
                # Combine original and edited value and mark with diff symbol '<=>'
                row_li.append(str(value_orig) + ' <=> ' + str(value_edit))
                b_diffs_found = True
            elif value_orig == value_edit:
                row_li.append(str(value_orig))
    # Add new row to dataframe when differences found only
    if b_diffs_found:
        dataframe_add_row(employees_df_diff, row_li)
        b_diffs_found = False

#employees_df_diff
```

Finally, the **differences** can be visualized even more prominently by **highlighting** the **cell backgrounds in color**. For this purpose the function `style.apply()` provided in `Pandas` is used. An **inline lambda function** searches the cells for the difference symbol `<=>` and **highlights** the **cell** with yellow color.

This allows to achieve a **similar functionality** as known from **spreadsheet programs** (e.g. Open Office Calc) as so-called **"conditional formatting"**.

[123]:
```
# Temporarily increase the number of rows to output
pd.set_option('display.min_rows', 20)
pd.set_option('display.max_rows', 40)

# Switch to apply highlight style to dataframe
# HINT: Set to 'False' when compiling to PDF!
highlight = False

if highlight:
    # Highlight cells by condition
    output = employees_df_diff.style.apply(lambda x:
                                           ['background: yellow'
                                            if str(v).find('<=>') != -1
                                            else "" for v in x],
                                           axis = 1)
else:
    output = employees_df_diff

output
```

[123]:

	idx	First Name	Gender	Start Date	\
0	1	Thomas <=> NaN	Male <=> NaN	3/31/1996 <=> NaN	
1	7	NaN	Female <=> NaN	7/20/2015 <=> NaN	
2	10	Louise <=> NaN	Female <=> NaN	8/12/1980 <=> NaN	
3	17	Shawn	Male	12/7/1986	
4	20	Lois	NaN <=> No Gender	4/22/1995	
5	22	Joshua	NaN <=> No Gender	3/8/2012	
6	23	NaN	Male <=> NaN	6/14/2012 <=> NaN	
7	25	NaN	Male <=> NaN	10/8/2012 <=> NaN	
8	27	Scott	NaN <=> No Gender	7/11/1991	
9	31	Joyce	NaN <=> No Gender	2/20/2005	
..	…	…	…	…	

```
235  943           Ralph  NaN <=> No Gender          7/28/1995
236  949          Gerald  NaN <=> No Gender          4/15/1989
237  950             NaN     Female <=> NaN  9/15/1985 <=> NaN
238  951             NaN       Male <=> NaN  7/30/2012 <=> NaN
239  955             NaN     Female <=> NaN  9/14/2010 <=> NaN
240  965         Antonio  NaN <=> No Gender          6/18/1989
241  976          Victor  NaN <=> No Gender          7/28/2006
242  989         Stephen  NaN <=> No Gender          7/10/1983
243  993          Justin  NaN <=> No Gender          2/10/1991
244  999           Henry  NaN <=> No Gender         11/23/2014

      Last Login Time            Salary          Bonus % Senior Management  \
0     6:53 AM <=> NaN   61933.0 <=> NaN     4.17 <=> NaN      True <=> NaN
1    10:43 AM <=> NaN   45906.0 <=> NaN  11598.0 <=> NaN               NaN
2     9:01 AM <=> NaN   63241.0 <=> NaN  15132.0 <=> NaN      True <=> NaN
3             7:45 PM   NaN <=> 90370.0           6414.0             False
4             7:18 PM           64714.0           4934.0              True
5             1:58 AM           90816.0          18816.0              True
6     4:19 PM <=> NaN  125792.0 <=> NaN   5042.0 <=> NaN               NaN
7     1:12 AM <=> NaN   37076.0 <=> NaN  18576.0 <=> NaN               NaN
8             6:58 PM          122367.0           5218.0             False
9             2:40 PM           88657.0          12752.0             False
..                …               …               …               …
235           6:53 PM           70635.0           2147.0             False
236          12:44 PM           93712.0          17426.0              True
237   1:50 AM <=> NaN  133472.0 <=> NaN  16941.0 <=> NaN               NaN
238   3:07 PM <=> NaN  107351.0 <=> NaN   5329.0 <=> NaN               NaN
239   5:19 AM <=> NaN  143638.0 <=> NaN   9662.0 <=> NaN               NaN
240           9:37 PM          103050.0              3.05             False
241           2:49 PM           76381.0          11159.0              True
242           8:10 PM           85668.0           1909.0             False
243           4:58 PM           38344.0           3794.0             False
244           6:09 AM          132483.0          16655.0             False

                       Team
0                       NaN
1            Finance <=> NaN
2                       NaN
3                   Product
4                     Legal
5            Client Services
6                       NaN
7    Client Services <=> NaN
8                     Legal
9                   Product
..                        …
235          Client Services
236             Distribution
237     Distribution <=> NaN
238        Marketing <=> NaN
239                     NaN
240                   Legal
241                   Sales
242                   Legal
243                   Legal
244             Distribution
```

```
[245 rows x 9 columns]
```

4.4.4 Save edited dataset to new CSV file

After the anomalies in the dataset have been found - and repaired where appropriate - it can be saved as a new CSV file for later use.

[74]:
```
csv_filepath = r'./datasets/employees_edit_repaired.csv'

employees_df.to_csv(csv_filepath, sep =',', index = False, header=True)
```

4.5 Avoidance of tendencies due to bias

The description of the Iris dataset says, that it consists of **50 samples** from **each of three species** of Iris (*Iris setosa*, *Iris virginica* and *Iris versicolor*), so there are **150 samples in total**.

But how can this be verified? The following subsections provide some **ideas** on how to **examine the dataset** for **tendencies** as a **cause of bias**.

4.5.1 Count occurrences of unique values

To prove whether all possible classes are included in the dataset and equally distributed, the function `df.value_counts()` can be used.

Following parameters are for fine tuning:

- `ascending=False`: sort resulting classes descending
- `dropna=False` causes that NaN values are included
- `normalize=True`: relative frequencies of the unique values are returned

[34]:
```
# Count unique values without missing values in a column,
# ordered descending and normalized
irisdata_df['species'].value_counts(ascending=False, dropna=False, normalize=False)
```

[34]:
```
Iris-versicolor    50
Iris-virginica     49
Iris-setosa        48
Name: species, dtype: int64
```

[121]:
```
# Import (again) employees dataset to dataframe from csv file
employees_df = pd.read_csv("./datasets/employees_edit.csv")
```

[124]:
```
# Count unique values and missing values in a column,
# ordered descending and absolute values
employees_df['Team'].value_counts(ascending=False, dropna=False, normalize=False)
```

[124]:
```
Client Services         106
Business Development    103
Finance                 102
Marketing                98
Product                  96
Sales                    94
Engineering              92
Human Resources          92
Distribution             90
Legal                    88
NaN                      43
Name: Team, dtype: int64
```

4.5.2 Display Histogram

This section was inspired by: Pandas Histogram – DataFrame.hist().

Histograms represent **frequency distributions** graphically. This requires the separation of the data into classes (so-called **bins**).

These bins are represented in the histogram as rectangles of equal or variable width. The height of each rectangle then represents the (relative or absolute) **frequency density**.

[461]:
```
employees_df.hist(column=['Salary'], bins = 'auto', density=True, rwidth=0.95,
                  zorder=2, alpha=0.8)
plt.title('Salary distribution over all gender')
plt.xlabel('salary')
plt.ylabel('frequency density')
plt.grid(True, zorder=-1.0)
plt.show()
```

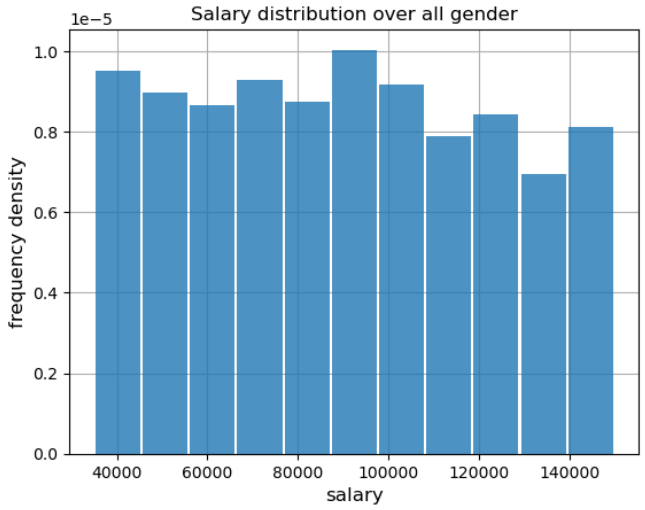


Figure 18: Histogram for frequency distribution of the salary

Apart from the not very appealing **standard formatting of the histogram** above, there is also **no breakdown of salaries** by **gender** here.

The following function allows a **gender-specific presentation** of salaries with significantly **more information content** in the individual subplots.

In addition, **probability density functions (PDF)** were overlaid on the histograms, whose hyper-parameters **mean** and **standard deviation** were previously identified using the features of the dataset. This makes it possible to estimate whether the **data is normally distributed**. In order to be able to reuse the code later, it was implemented as the **function** `func_plot_histograms_from_list_with_PDF()`.

[482]:
```
from scipy.stats import norm

def func_plot_histograms_from_list_with_PDF(df_list, column, titles, y_max):
    # Number of bins for the histogram
    # - bins=<integer>: defines the number of equal-width bins in the range
    # - bins=<string>: one of the binning strategies is used:
    #   'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'
    n_bins = 'auto'
    subplot_columns = len(df_list)
    fig, subplots = plt.subplots(1, subplot_columns, figsize=(14, 4))
```

```
    # Set margins between subplots
    plt.subplots_adjust(wspace=0.3, hspace=0.3)

    # Make subplots iterable via 'subplots.flatten()'
    for df, title, subplot in zip(df_list, titles, subplots.flatten()):

        subplot.hist(df[column], bins = n_bins, density=True,
                     rwidth=0.95, alpha=0.8)

        # Fit a normal distribution to the data
        # with mean and standard deviation
        mu, std = norm.fit(df[column])

        # Plot the probability density function (PDF)
        xmin, xmax = subplot.get_xlim()

        x = np.linspace(xmin, xmax, 100)
        p = norm.pdf(x, mu, std)

        subplot.plot(x, p, 'k', linewidth=2)

        title_concat = "Salary of {} (Mean: {:.2f}, \n" \
                               "Std. deviation: {:.2f})".format(title, mu, std)
        subplot.set_title(title_concat)
        # Show grid
        subplot.grid(visible=True)
        # Hide grid behind the bars
        subplot.set_axisbelow(True)
        # Label x and y-axis
        subplot.set_xlabel('salary')
        subplot.set_ylabel('frequency density')
        # Rotate x-ticks by -45°
        subplot.tick_params('x', labelrotation=-45)

    # Set all y-axes to the same range for comparison
    plt.setp(subplots, ylim=subplots[2].get_ylim())
    plt.show()
```

[483]:
```
genders = ['Male', 'Female', 'No Gender']

# Create list for storing the dataframes
li_employees_df = list()

# Filter employees by gender
for gender in genders:
    li_employees_df.append(employees_df.loc[(employees_df['Gender'] == gender)])

func_plot_histograms_from_list_with_PDF(li_employees_df, 'Salary', genders, 1.4)
```

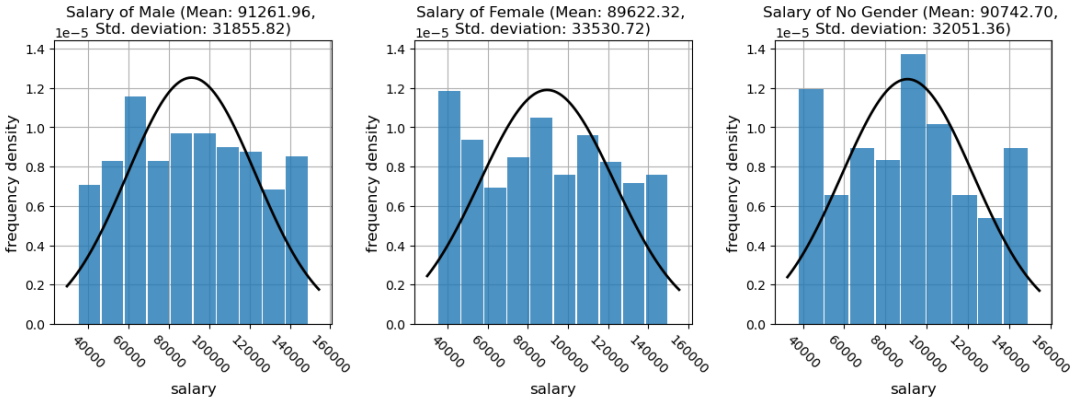


Figure 19: Histograms used to explore the frequency distribution of the salary in comparison between the genders (with overlaid probability density functions (PDF))

4.6 First idea of correlations in dataset

To get a rough idea of the **dependencies** and **correlations** in the dataset, it can be helpful to visualize the whole dataset in a **correlation heatmap**. They show in a glance which variables are correlated, to what degree and in which direction.

Later, 2 particularly well correlated variables are selected from the dataset and plotted in a **scatterplot**.

4.6.1 Visualise data with correlation heatmap

This section was inspired by How to Create a Seaborn Correlation Heatmap in Python?.

> **Correlation matrices** are an **essential tool of exploratory data analysis**. Correlation heatmaps contain the same information in a visually appealing way. What more: they show in a glance which variables are correlated, to what degree, in which direction, and alerts us to potential multicollinearity problems (source: ibidem).

Simple correlation matrix Because **string values can never be correlated**, the class names (species) have to be converted first:

[24]:
```
# encoding the class column
irisdata_df_enc = irisdata_df.replace({"species":  {"Iris-setosa":0,
                                                    "Iris-versicolor":1,
                                                    "Iris-virginica":2}})
#irisdata_df_enc
```

[25]:
```
irisdata_df_enc.corr()
```

[25]:

	sepal_length	sepal_width	petal_length	petal_width	species
sepal_length	1.000000	-0.109369	0.871754	0.817954	0.782561
sepal_width	-0.109369	1.000000	-0.420516	-0.356544	-0.419446
petal_length	0.871754	-0.420516	1.000000	0.962757	0.949043
petal_width	0.817954	-0.356544	0.962757	1.000000	0.956464
species	0.782561	-0.419446	0.949043	0.956464	1.000000

Correlation heatmap Choose the color sets from color map.

[26]:
```
# increase the size of the heatmap
plt.figure(figsize=(16, 6))
```

```
# store heatmap object in a variable to easily access it
# when you want to include more features (such as title)
# set the range of values to be displayed on the colormap from -1 to 1,
# and set 'annotation=True' to display the correlation values on the heatmap
heatmap = sns.heatmap(irisdata_df_enc.corr(), vmin=-1, vmax=1,
                      annot=True, cmap='PRGn_r')

# give a title to the heatmap
# 'pad=12' defines the distance of the title from the top of the heatmap
heatmap.set_title('Correlation Heatmap', fontdict={'fontsize':18}, pad=16)
plt.show()
```

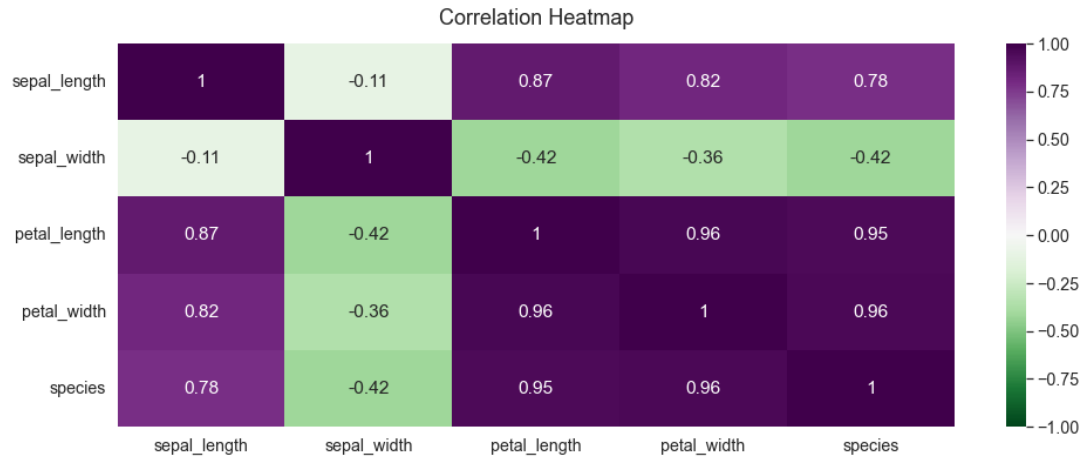


Figure 20: Correlation heatmap to explore coherences between single variables in the iris dataset

Triangle correlation heatmap When looking at the correlation heatmaps above, you would not lose any information by **cutting** away half of it **along the diagonal** line marked by 1-s.

The **numpy** function `np.triu()` can be used to isolate the upper triangle of a matrix while turning all the values in the lower triangle into 0.

[27]:
```
np.triu(np.ones_like(irisdata_df_enc.corr()))
```

[27]:
```
array([[1., 1., 1., 1., 1.],
       [0., 1., 1., 1., 1.],
       [0., 0., 1., 1., 1.],
       [0., 0., 0., 1., 1.],
       [0., 0., 0., 0., 1.]])
```

Use this mask to cut the heatmap along the diagonal:

[28]:
```
plt.figure(figsize=(16, 6))

# define the mask to set the values in the upper triangle to 'True'
mask = np.triu(np.ones_like(irisdata_df_enc.corr(), dtype=bool))

heatmap = sns.heatmap(irisdata_df_enc.corr(), mask=mask,
                      vmin=-1, vmax=1, annot=True, cmap='PRGn_r')

heatmap.set_title('Triangle Correlation Heatmap', fontdict={'fontsize':18}, pad=16)
plt.show()
```

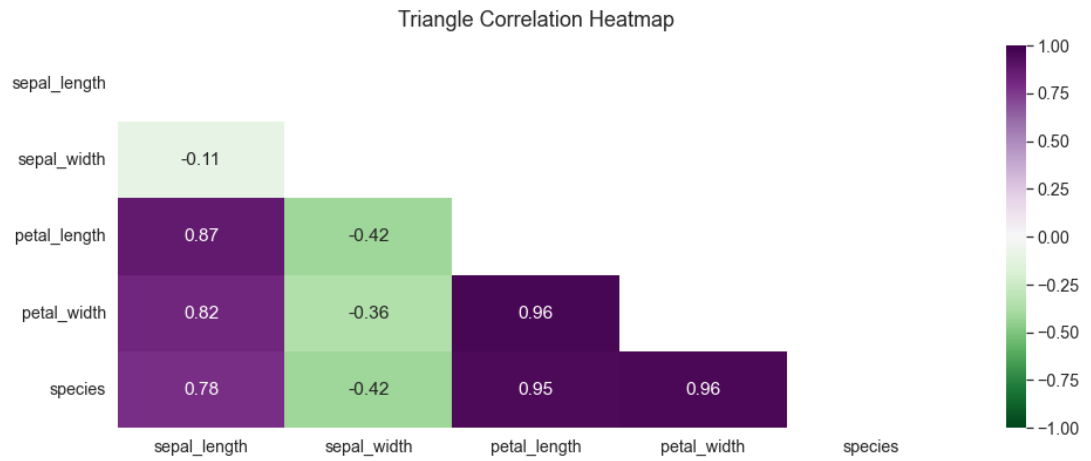


Figure 21: Correlation heatmap, which was cut at its main diagonal without losing any information

As a result from the **heatmaps** we can see, that the shape of the **petals** are the **most correlationed columns** (0.96) with the **type of flowers** (species classes).

Somewhat lower correlates **sepal length** with **petal length** (0.87).

4.6.2 Visualise data with scatter plot

In the following, Seaborn is applied which is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with `Pandas` data structures.

To investigate whether there are dependencies (e.g. correlations) in `irisdata_df` between individual variables in the dataset, it is advisable to plot them in a **scatter plot**.

[141]:
```
# There are five preset seaborn themes: darkgrid, whitegrid, dark, white, and ticks.
sns.set_style("whitegrid")
# set scale of fonts
sns.set_context("notebook", font_scale=1.3, rc={"lines.linewidth": 2.5})

# 'sepal_length', 'petal_length' are iris feature data
# 'height' used to define height of graph
# 'hue' stores the class/label of iris dataset
sns.FacetGrid(irisdata_df, hue ="species",
              height = 7).map(plt.scatter,
                              'petal_width',
                              'petal_length').add_legend()


plt.title('Scatterplot of petal length and width')
plt.xlabel('petal width [cm]')
plt.ylabel('petal length [cm]')
plt.show()
```

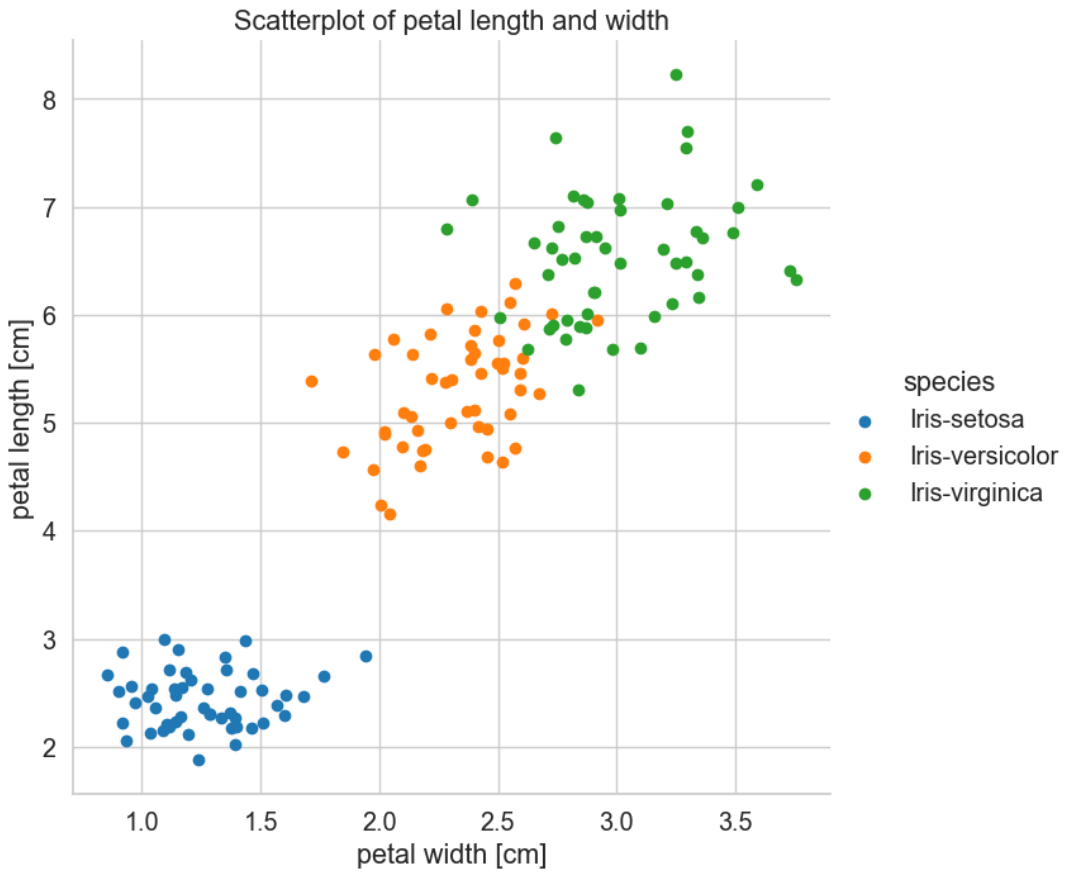


Figure 22: Plotting two individual variables of the iris dataset in the scatterplot to explore the relationships between these two

4.6.3 Visualise data with pairs plot

For systematic investigation of dependencies, all variables (each against each) are plotted in separate scatter plots.

With this so called **pairs plot** it is possible to see both **relationships** between two variables and **distribution** of single variables.

This function will create a grid of Axes such that **each numeric variable** in `irisdata_df` will by shared in the y-axis across a single row and in the x-axis across a single column.

[142]:
```
sns.set(font_scale=1.0)
sns.set_style("white")

g = sns.pairplot(irisdata_df, diag_kind="kde", hue='species',
                 palette='Dark2', height=2.5)

g.map_lower(sns.kdeplot, levels=4, color=".2")
# y .. padding between title and plot
g.fig.suptitle('Pairs plot of the Iris dataset', y=1.05)
plt.show()
```

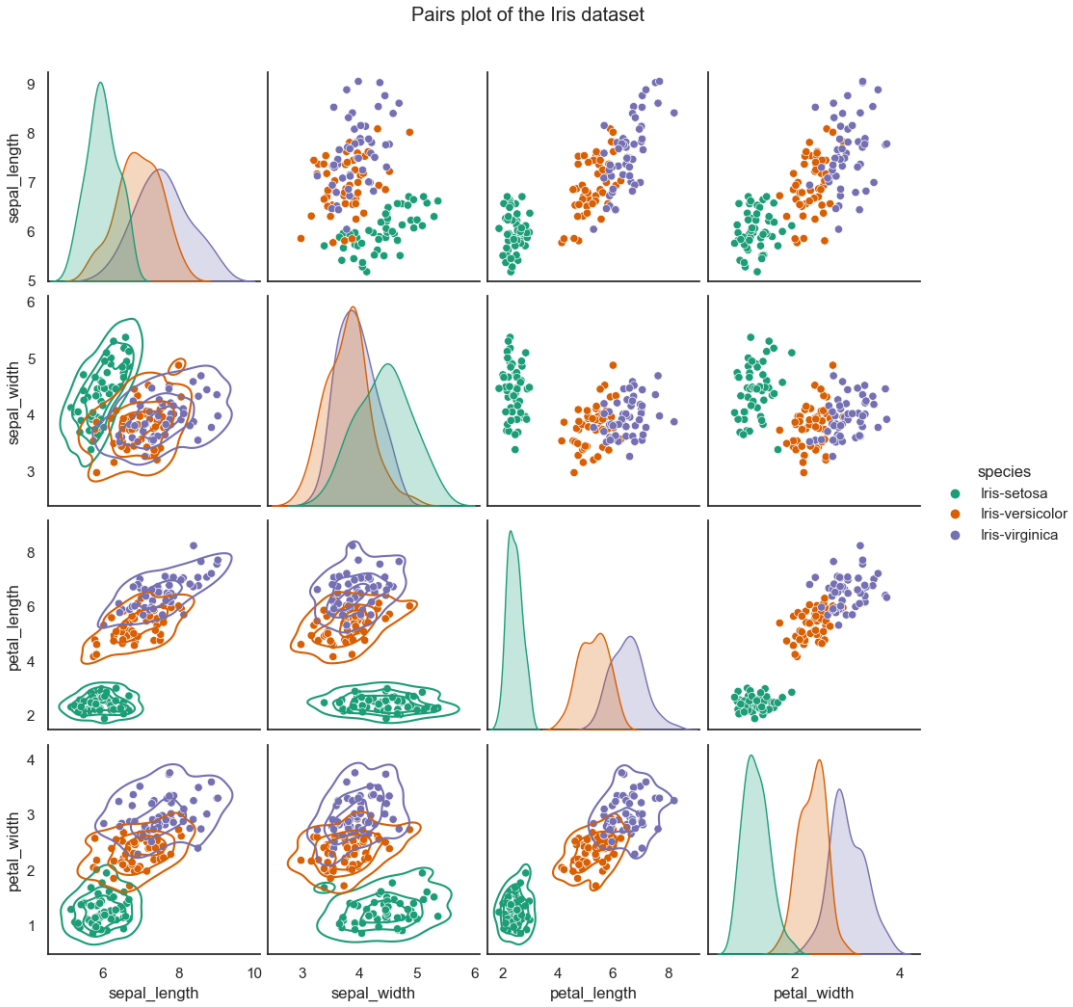


Figure 23: Plot all individual variables of the Iris dataset in pairs plot to see both the relationships between two variables and the distribution of the individual variables

5 STEP 3: Choose and create the ML model

After exploring the dataset, in this step one has to decide on a specific ML algorithm based on certain selection criteria.

However, since the AI or ML world is so huge and impossible for a ML novice to overlook, a brief description of the **relationship between AI and ML** is given in the following sections. Furthermore, a **taxonomy** of the different **learning types** is presented by also providing some example algorithms.

5.1 Short overview of the AI world

The history of **Artificial Intelligence (AI)** with the Dartmouth Conference in the summer of 1956 as its birth is characterized by several successive hype and low phases (so-called AI Winter). During the hype phases, many **new insights** were gathered by AI researchers and interesting **application areas** were explored. However, there were always times when the verifiable successes fell far short of the previously awakened (inflated) expectations. The consequence was then a decreasing interest in AI research and accompanying drastic cuts in research budgets. The **high media attention** in combination with often **vague and not very clear definitions** of AI can be seen as a potential reason for the often circulating inflated expectations of AI technologies.

Due to the increase in knowledge from AI research as well as the exploration of new application areas, the technical terms and especially the AI definitions have been subject to constant change over the past

decades. In the currently published standard ISO/IEC 22989:2022-07, **AI systems** have been defined by the **Subcommittee 42 - 'Artificial Intelligence' (SC 42)** of the **ISO/IEC Joint Technical Committee (JTC 1)** as follows (see definition 3.1.4 in ISO/IEC 22989 2022).

The main part of the **definition** describes what an AI system (should) do:

> [An **artificial intelligence system** is an] engineered system that generates outputs such as content, forecasts, recommendations or decisions for a given set of human-defined objectives.

A **Note** to the definition describes the techniques necessary to achieve this:

> [..] The engineered system can use various techniques and approaches related to artificial intelligence to develop a **model** to represent data, **knowledge**, processes, etc. which can be used to conduct **tasks**.

The **knowledge** acquires itself from abstracted information about objects, events, concepts or rules as well as their properties and relations to each other. It is organized for purposeful systematic use. The **model** is represented by a physical, mathematical, or otherwise logical representation of a system. Whereas the **task** consists of a set of actions required to achieve a specific goal.

Machine Learning (ML) as a subset of AI, on the other hand, addresses the mathematical models and algorithms that enable a computer system to recognize (new) correlations in huge amounts of sample data from various sources by inferring them independently.

The umbrella term AI covers a very large research area. It includes a number of techniques, like:

- Supervised and Unsupervised Learning,
- Reinforcement Learning and
- Genetic Algorithms

that enable computers to learn independently and solve complex problems in the fields of, e.g.:

- Computer-Vision (CV),
- Computational Linguistics (CL) or
- Robotics.

The following **Venn diagram** shows the relationship between AI, machine learning and other integrated technologies. The quantities that do not belong to the main category represent techniques that can function as stand-alone techniques and do not necessarily fall into the artificial intelligence group in all cases (Gonzalez Viejo et al. 2019).

For example, simple **robotic behaviors** can be realized via fixed pre-programmed **if-then-else decisions**. In images, objects can be identified by edge detection by applying, for example, Sobel or Laplace filters. In both examples, no learnable algorithms are needed, therefore the Venn diagram was adapted accordingly.

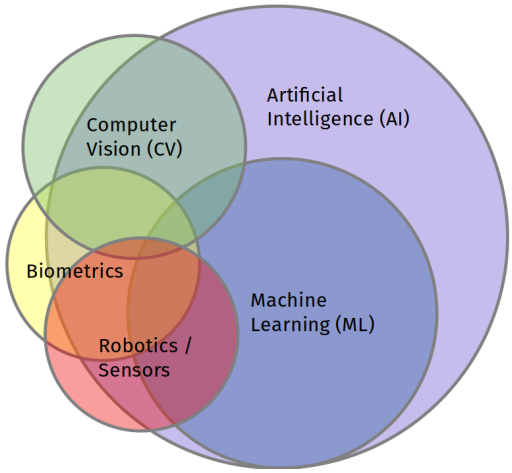


Figure 24: Venn diagram showing the relationship between AI, machine learning and other integrated technologies (source: Kasper, adapted from Gonzalez Viejo et al. 2019, license: CC BY-SA 4.0)

5.2 Taxonomy of machine learning algorithms

The field of machine learning can be divided into the following **types of learning**:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning
- Reinforcement learning

Here are some further sources:

- Taxonomy of machine learning algorithms
- Comprehensive Survey of Machine Learning Approaches in Cognitive Radio-Based Vehicular Ad Hoc Networks
- A Taxonomy of Machine Learning Techniques
- ML Algorithms: One SD
- Machine Learning Map

5.2.1 Supervised learning

The goal of supervised learning is to learn a **function** that maps a **input to an output**, based on example input-output pairs. This involves inferring a relationship describable by a mathematical function from **labeled training data** consisting of a set of training examples (Wikipedia: Supervised learning 2022).

A few well-known algorithms from the field of **supervised learning** are mentioned here:

- Naive Bayes
- Linear Regression
- Logistic Regression
- Artificial Neural Networks (ANN)
- Support Vector Classifier (SVC)
- Decision Trees
- Random Forests

5.2.2 Unsupervised learning

The algorithms of unsupervised learning look for internal structures in the data of a dataset, such as **grouping** or **clustering of data points**. These algorithms can thus learn relationships from test data that have not been labeled, classified, or categorized. Rather than responding to feedback (as in

supervised learning), unsupervised learning algorithms detect **commonalities in the data** and respond based on the presence or absence of such commonalities in each new dataset (Wikipedia: Unsupervised learning 2022).

Here are some algorithms from the field of **unsupervised learning**:

- K-means Clustering
- Spectral Clustering
- Hierarchical Clustering
- Principal Component Analysis (PCA)

5.2.3 Semi-supervised learning

The semi-supervised learning falls between **unsupervised** learning (without any labeled training data) and **supervised** learning (with completely labeled training data). Some of the training examples are missing training labels, yet many machine-learning researchers have found that unlabeled data, when used in conjunction with a small amount of labeled data, can produce a considerable improvement in learning accuracy (Wikipedia: Semi-supervised learning 2022).

5.2.4 Reinforcement learning

This is an area of machine learning concerned with how **intelligent agents** ought to **take actions in an environment** in order to maximize the notion of cumulative **reward**. Due to its generality, the field is studied in many other disciplines, such as **game theory** and **control theory**.

Reinforcement learning differs from supervised learning in **not needing labeled input/output pairs** be presented and in not needing sub-optimal actions to be explicitly corrected. Instead the focus is on **finding a balance** between **exploration** of uncharted territory and **exploitation** of current knowledge (Wikipedia: Reinforcement learning 2022).

Here are some algorithms from the field of **reinforcement learning**:

- Iterative Policy
- Q-Learning
- SARSA
- Learning Classifiers
- Stochastic Gradient
- Genetic Algorithm

5.3 Decision graph for selecting an suitable algorithm

Now that the Iris dataset has been analyzed in terms of its data structure and internal correlations, the most difficult task on the way to solving a problem using machine learning arises: finding the “right” ML algorithm (also called **estimator**).

The diverse estimators available are more or less well qualified for the respective problems with their partly very different data types. The good news is that the ML software package `Scikit-Learn` provides the following **flowchart** as a rough **guide** in choosing the right estimator for the particular task (see: Choosing the right estimator).

However, it must also be emphasized that a considerable **level of experience** through systematic trial and error is crucial to be successful in finding an “optimal” estimator.

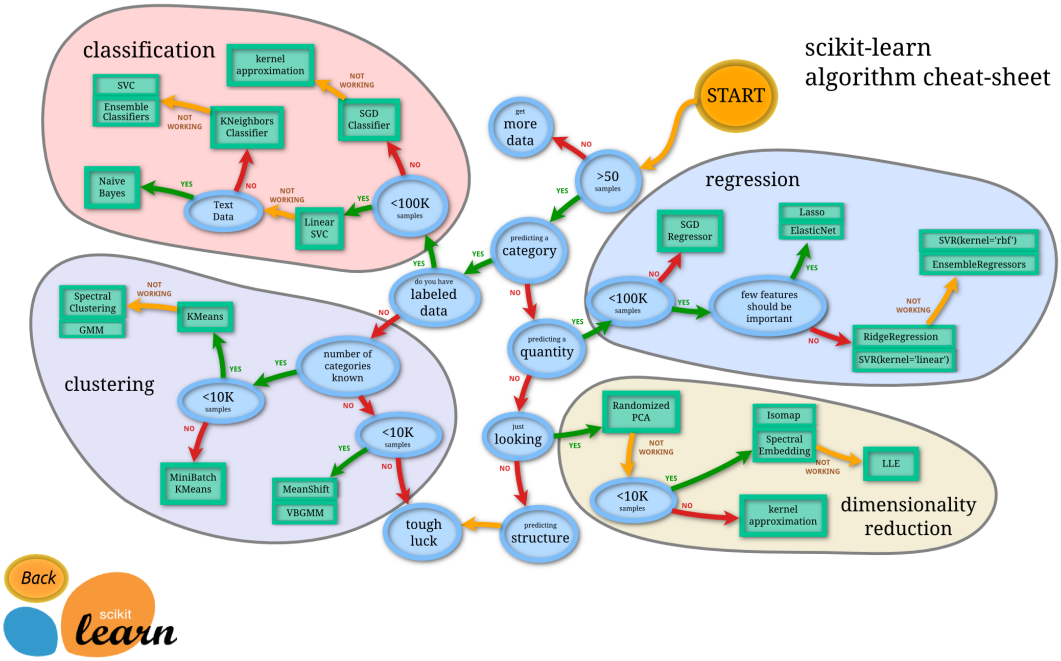


Figure 25: Decision graph for choosing an appropriate ML algorithm (source: Choosing the right estimator, license: unknown)

5.4 Reasons for choosing Support Vector Classifier (SVC)

Among other ML algorithms suitable for the Iris dataset (such as the decision-tree-based **random-forests classifier**), the reasoned choice here in this tutorial falls on the **support vector classifier (SVC)**.

The following **reasons** led to the decision for the **Support Vector Classifier (SVC)**:

- The aim is to predict the species using unlabeled test data, so the task is to **classify**.
- The iris dataset is **fully labeled** (by designating the iris species).
- The dataset contains significantly **less than 100k samples**.

But the most important reason is that it is **easy to understand** how it works - so it is exactly suitable for a beginner tutorial.

5.5 Operating principal of SVC

Support-vector machines (SVMs) are **supervised learning** models with associated learning algorithms that analyze data for **classification** and **regression** analysis (Wikipedia: Support-vector machine 2022).

Since a **classifier** is needed for the current task to classify the Iris dataset, the following description of the operating principle focuses on the **Support Vector Classifier (SVC)**.

5.5.1 Support Vectors and hyperplane

The SVC algorithm plots the training data in an **n-dimensional space**. The number of dimensions results from the **number of variables** or features. For the Iris dataset with its 4 features, this would result in a 4-dimensional space. For better understanding, the following explanation is limited to the **2-dimensional space**.

The SVC algorithm now tries to draw a **boundary with the largest possible distance** to the next sample from the training data. This boundary is actually a **hyperplane** whose **dimension is 1 smaller** than that of the training data (Lell 2019). For example, in 3-dimensional space, the hyperplane would

be a 2-dimensional non-curved plane. In 2-dimensional space, a hyperplane would simply be a straight line.

The following figure shows the principle of operation of the SVC algorithm in 2-dimensional space with 2 classes to separate: the hyperplanes *H1* to *H4* (left figure) separate the classes. A good separation of the classes is achieved by the hyperplane that has the **largest distance to the nearest training data point** of a class (so-called **functional margin**). The larger the margin, the better the classifier can later separate test data which are unknown to it. This is called minimization of the **generalization error**.

The right graph shows the **optimal hyperplane** characterized by **maximizing the margin** between classes (Fraj 2018). The perpendicular distance of the data points closest to the hyperplane determines their position and orientation. These perpendicular distances are the **support vectors** of the hyperplane - hence the algorithm got it's name.

Interestingly, the vectors that are more distant from the boundary are not important for the calculation. Therefore, they do not need to be loaded into the main memory, which makes the SVC very memory efficient.

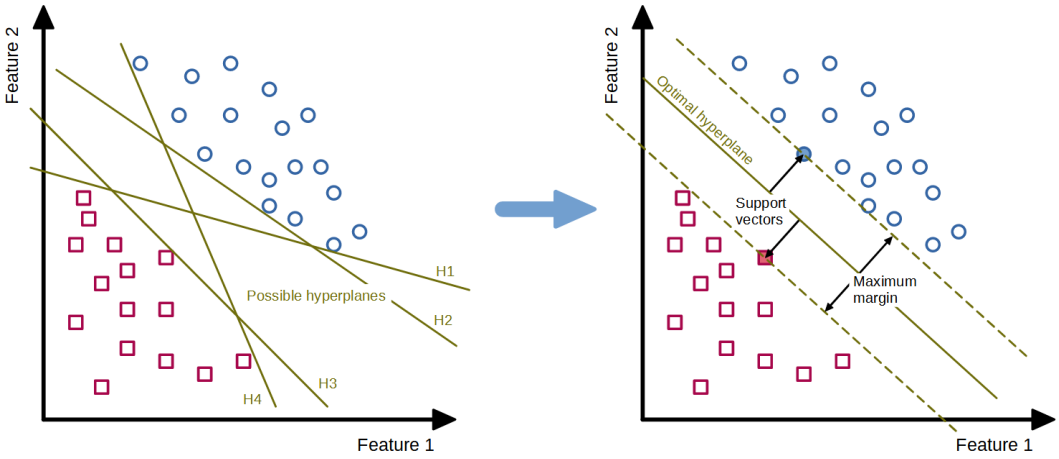


Figure 26: Support Vector Classifiers (SVC) separate the data points in classes by finding the best hyperplane by maximizing the margin to its support vectors (source: Kasper, license: CC BY-SA 4.0)

5.5.2 Non-linear transformations

The previous example assumes that the **data is linearly separable**. For most real cases in practice this is unfortunately not true and the SVC can only work with hyper**planes**.

If the data are not linearly separable in the original space, **transformations** can be applied to the data. For this purpose, the data are transferred into a **higher-dimensional feature space** where the objects are linearly separable by a hyperplane. However, during **backtransformation** this hyperplane becomes **nonlinear** and often also non-continuous (Lell 2019).

In the following example, the figure on the left shows the original data points in 1-dimensional space. In the first dimension, these data are not linearly separable. After applying the transformation $\Phi(X) = X^2$ and adding this second dimension to our feature space, the classes in the right figure become linearly separable (Wilimitis 2018).

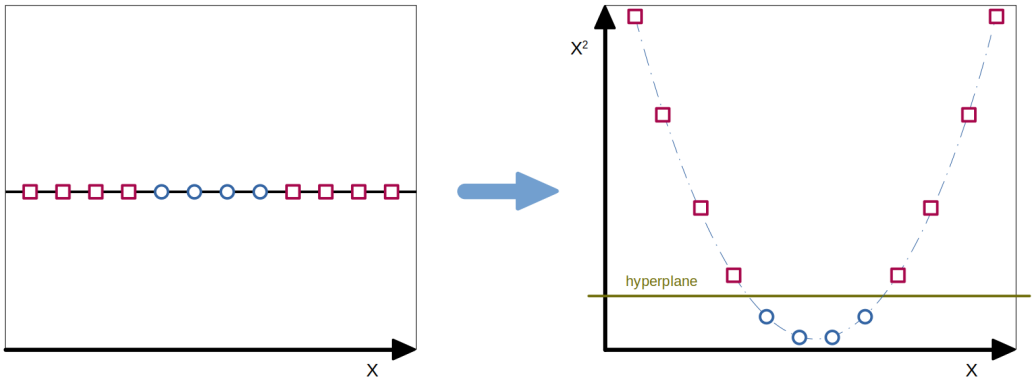


Figure 27: Transformation of 1-dimensional data into 2-dimensional space in order to separate the data by a linear hyperplane (source: Kasper, license: CC BY-SA 4.0)

5.5.3 Kernel trick

In the previous example, it was shown how a transformation to a higher dimensional feature space allows the data to be separated. In order to train an SVC and make classification predictions, mathematical operations would need to be performed on the higher dimensional vectors in the transformed feature space. However, in real-world applications, there may be many features in the data. The application of transformations involve many polynomial combinations of these features. This leads to **extremely high computational costs** (Wilimitis 2018).

The **kernel trick** provides a solution to this problem. The "trick" is that **kernel methods** represent the data only by a series of **pairwise similarity comparisons** between the original data observations X (with the original coordinates in low-dimensional space). This allows to work in the original feature space instead of explicitly applying the transformations $\Phi(X)$ and representing the data by these transformed coordinates in the higher dimensional feature space (Wilimitis 2018; Zhang 2018).

The following **kernel types** are provided by the Python package `scikit-learn`:

- linear
- radial basis function (RBF)
- polynomial
- sigmoid

The most well-known are the polynomial kernel and the radial basis function (RBF) kernel, which is also called the Gaussian kernel.

5.6 Create the SVC model

In this step we create the SVC model choosing a **linear kernel** with default parameters.

[58]:
```
from sklearn.svm import SVC
classifier = SVC(kernel = 'linear', random_state = 0)
```

6 STEP 4: Preprocess the dataset for training

In this step the dataset is prepared for the actual classification by SVC. Depending on the selected ML algorithm as well as the data structure, it may be necessary to prepare the data before training (e.g., by **standardization**, **normalization**, or **discretization** to cluster the data based on thresholds). Furthermore, errors in the dataset (e.g. **data gaps**, **duplicates** or obvious **misentries**) should be corrected now at the latest.

6.1 Heal the dataset

Through the intensive exploration of the data (see STEP 2: Explore the ML dataset), we know that special **preparation** of the data is **not necessary**. The values of the Iris dataset are **complete and without gaps**.

[25]:
```
# Import ORIGINAL Iris dataset for classification
irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle.csv')

# Import NOISED Iris dataset for classification
#irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle_noised.csv')
```

Find **all completely identical duplicates** (first and last occurrences). The resulting dataframe is **sorted by column** `'species'` to get the **duplicates grouped**:

[29]:
```
# Parameter 'keep=False' displays all duplicate rows
irisdata_duplicateRows = irisdata_df[irisdata_df.duplicated(keep=False)]

# Sort rows by column 'First Name' to get the duplicates grouped
irisdata_duplicateRows.sort_values('species')
```

[29]:
```
Empty DataFrame
Columns: [sepal_length, sepal_width, petal_length, petal_width, species]
Index: []
```

Interestingly, there are indeed **duplicates** in the original **Iris dataset**.

The duplicates occur **imbalanced** across the classes:

- class **Iris-setosa** has **3** identical duplicates
- class **Iris-virginica** has **2** identical duplicates
- class **Iris-versicolor** has **none** duplicates

This **imbalance** could cause **tendencies** and have a (negative) effect on the **classification result**. Therefore, the duplicates are removed and the **cleaned Iris dataset** is **saved** as a new CSV file.

[28]:
```
# Remove duplicate rows across all columns
irisdata_df.drop_duplicates(inplace=True)
irisdata_df
```

[28]:

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	Iris-setosa
1	4.9	3.0	1.4	0.2	Iris-setosa
2	4.7	3.2	1.3	0.2	Iris-setosa
3	4.6	3.1	1.5	0.2	Iris-setosa
4	5.0	3.6	1.4	0.2	Iris-setosa
..	…	…	…	…	…
145	6.7	3.0	5.2	2.3	Iris-virginica
146	6.3	2.5	5.0	1.9	Iris-virginica
147	6.5	3.0	5.2	2.0	Iris-virginica
148	6.2	3.4	5.4	2.3	Iris-virginica
149	5.9	3.0	5.1	1.8	Iris-virginica

[147 rows x 5 columns]

[52]:
```
# Count unique values without missing values in a column,
# ordered ascending and not normalized
irisdata_df['species'].value_counts(ascending=True, dropna=False, normalize=False)
```

[52]:
```
Iris-setosa        48
Iris-virginica     49
Iris-versicolor    50
Name: species, dtype: int64
```

[37]:
```
csv_filepath = r'./datasets/IRIS_flower_dataset_kaggle_cleaned.csv'

irisdata_df.to_csv(csv_filepath, sep =',', index = False, header=True)
```

6.2 Transform the dataset by feature scaling

Some machine learning algorithms are very sensitive to feature scaling, while others are virtually unaffected.

Distance-based algorithms such as the **Support Vector Classifier (SVC)** explained in STEP 3, or others such as K-Nearest Neighbors (KNN) and K-means clustering are most affected by the **bandwidth of features**. This is because **distances between data points** are used in the algorithm to determine their **similarity** (see Bhandari 2020).

If **features** in the dataset have very **different ranges (scales)**, the features with the larger scale could be **weighted** more prominently by the ML algorithm. This **tendency** would lead to **bias** in training, which negatively affects **generizability** in classifying test data. Therefore, the **features** in the dataset should be **scaled** before distance-based ML algorithms are used. Only by **adjusting the data ranges** it is possible to ensure that **all features contribute equally** to the classification result.

The following two subsections explain the two main methods for scaling: **Normalization** and **Standardization**. The question often arises as to when one or the other should be used, so here are a few hints:

- **Normalization** is useful when the distribution of the data **does not** follow a **Gaussian normal distribution**. This can be helpful for algorithms that do not assume normally distributed data, such as K-Nearest Neighbors and neural networks. However, **outliers** in the data have **major influence** on the **mean** (not to be mistaken with the median) used for calculation. Therefore, normalization is significantly **more vulnerable to outliers** than standardization.
- The **standardization**, on the other hand, can be helpful in cases where the data follow a **Gaussian normal distribution**. However, this does not necessarily have to be true. Also, unlike normalization, standardization does not have a limited range of values. However, standardization is significantly **less prone to outliers** in the data than normalization.

Finally, the **decision for normalization or standardization** depends on the concrete **task**, the **data** and the **ML algorithm** used. It is recommended to train the ML model first with the raw data and then with the normalized or standardized data. A subsequent comparison of the classification results (e.g., using the cross-validation or the root-mean-square error (RMSE)) provides guidance on which scaling should be used (see Bhandari 2020).

At this point, an **important note** that is repeatedly mentioned in the literature when talking about scaling (see Bhandari 2020; Géron 2018):

> It is a good practice to **fit the scaler** on the **training data** and then use it to **transform the testing data**. This would avoid any data leakage during the model testing process. Also, the scaling of target values is generally not required.

For further details about **Standardization** and **Normalization** read here:

scikit-learn:

- Preprocessing data
- Compare the effect of different scalers on data with outliers

Others:

- What are standarization and normalization? Test with iris data set in Scikit-learn

- Feature Scaling for Machine Learning: Understanding the Difference Between Normalization vs. Standardization
- Feature scaling
- Normalization (statistics)
- Standard score

6.2.1 Normalization

When scaling by **normalization**, the values are shifted and rescaled so that they range **between 0 and 1**. It is also known in the literature as **min-max scaling** (see Bhandari 2020).

The **normalization** is calculated following this **formula**:

$$X' = \frac{X - X_{min}}{X_{max} - X_{min}}$$

Thereby X_{max} and X_{min} are the maximum and the minimum values of the feature, respectively.

- If the value of X is the **minimum value** of the feature, the numerator in the fraction becomes 0. Thus, $X' = 0$.
- If the value of X is the **maximum value** of the feature, the numerator becomes equal to the denominator of the fraction. Then the value of $X' = 1$.
- If the value of X is between the minimum and maximum values, the value of X' is between 0 and 1.

For further details read here: scikit-learn: Normalization.

[402]:
```
from sklearn.preprocessing import MinMaxScaler

# Fit the MinMax scaler on raw input dataframe
# by selecting columns 1-4 with all feature rows
#norm_scaler = MinMaxScaler().fit(irisdata_df.iloc[:, 0:4])
# by ommitting the last column with class names
norm_scaler = MinMaxScaler().fit(irisdata_df.drop('species', axis=1))

# Transform the raw dateframe with fitted scaler
# and convert it to numpy array
#irisdata_np_norm = norm_scaler.transform(irisdata_df.iloc[:, 0:4])
irisdata_np_norm = norm_scaler.transform(irisdata_df.drop('species', axis=1))

#irisdata_np_norm
```

[403]:
```
# Make a deep copy of original dataframe
irisdata_df_norm = irisdata_df.copy(deep=True)

# Replace values of dataframe with normalized values from array
# by writing in the first 4 columns (ommit last column with class names)
irisdata_df_norm.iloc[:, 0:4] = irisdata_np_norm

#irisdata_df_norm
```

[404]:
```
irisdata_df_norm.describe()
```

[404]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	0.428704	0.439167	0.467571	0.457778
std	0.230018	0.180664	0.299054	0.317984
min	0.000000	0.000000	0.000000	0.000000
25%	0.222222	0.333333	0.101695	0.083333

```
50%         0.416667      0.416667      0.567797      0.500000
75%         0.583333      0.541667      0.694915      0.708333
max         1.000000      1.000000      1.000000      1.000000
```

To display the **original data** with the **scaled data** side-by-side as **boxplots** with all **features in one scale**, the function `func_boxplots_comp_scaling()` is implemented.

[411]:
```
def func_boxplots_comp_scaling(dataframes, titles):
    fig, subplots = plt.subplots(1, 2, figsize=(12, 5))
    # Set margins between subplots
    plt.subplots_adjust(wspace=0.3, hspace=0.3)

    # Make subplots iterable via 'subplots.flatten()'
    for title, df, subplot in zip(titles, dataframes, subplots.flatten()):

        # To create multiple boxplots in seaborn,
        # we must first melt the pandas.DataFrame into a long format:
        df_melted = pd.melt(df.drop('species', axis=1))
        # Rename column to 'features'
        df_melted.rename(columns={'variable': 'features'}, inplace=True)

        sns.boxplot(x='features', y='value', data=df_melted, ax = subplot)
        # Show grid
        subplot.grid(axis='y')
        # Hide grid behind the bars
        subplot.set_axisbelow(True)
        subplot.set_title('{} data'.format(title))
        subplot.set_ylabel('value range [cm]')

    plt.show()
```

Call the new function `func_boxplots_comp_scaling()` to create the **boxplots** comparing **original data** with the **scaled data** side-by-side:

[412]:
```
titles     = ['Original', 'Normalized']
dataframes = [irisdata_df, irisdata_df_norm]

func_boxplots_comp_scaling(dataframes, titles)
```

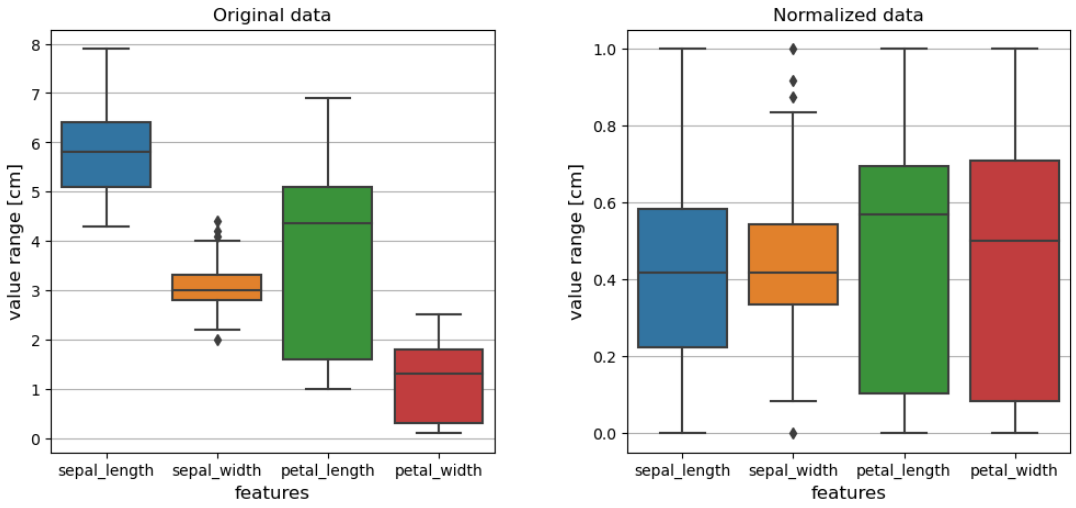


Figure 28: Boxplots comparing the original data (left) with the normalized data (right) with all features in one scale

To compare the **original data** with the **scaled data** side-by-side as **histograms** with overlaid **probability density functions**, the function `func_histograms_comp_scaling()` is implemented.

[422]:
```
from scipy.stats import norm

def func_histograms_comp_scaling(df_orig, df_scaled, features,
                                 titles, scaling_type):

    # Number of bins for the histogram
    # - bins=<integer>: defines the number of equal-width bins in the range
    # - bins=<string>: one of the binning strategies is used:
    #   'auto', 'fd', 'doane', 'scott', 'stone', 'rice', 'sturges', or 'sqrt'
    n_bins = 'auto'
    #n_bins = 10
    fig, subplots = plt.subplots(4, 2, figsize=(12, 16))
    # Set margins between subplots
    plt.subplots_adjust(wspace=0.3, hspace=0.4)

    # Make subplots iterable via 'subplots.flatten()'
    subplot_list = subplots.flatten()

    subplots_orig   = [0, 2, 4, 6]
    subplots_scaled = [1, 3, 5, 7]

    # Show histograms with ORIGINAL data,
    # so loop through list of subplots with EVEN indexes
    for feature, title, index_subplt in zip(features, titles, subplots_orig):
        subplot_list[index_subplt].hist(df_orig[feature],
                                        bins = n_bins, rwidth=0.95,
                                        density=True, alpha=0.8)

        # Fit a normal distribution to the data
        # with mean and standard deviation
        mu, std = norm.fit(df_orig[feature])

        # Plot the probability density function (PDF)
        xmin, xmax = subplot_list[index_subplt].get_xlim()
        x = np.linspace(xmin, xmax, 100)
        p = norm.pdf(x, mu, std)
        subplot_list[index_subplt].plot(x, p, 'k', linewidth=2)

        title_concat = "{} (Mean: {:.2f}, " \
                       "Std. deviation: {:.2f})".format(title, mu, std)
        subplot_list[index_subplt].set_title(title_concat)
        # Show grid
        subplot_list[index_subplt].grid(visible=True)
        # Hide grid behind the bars
        subplot_list[index_subplt].set_axisbelow(True)
        # Label x and y-axis
        subplot_list[index_subplt].set_xlabel('original value range [cm]')
        subplot_list[index_subplt].set_ylabel('frequency density')


    # Show histograms with SCALED data,
    # so loop through list of subplots with ODD indexes
    for feature, title, index_subplt in zip(features, titles, subplots_scaled):
        subplot_list[index_subplt].hist(df_scaled[feature],
                                        bins = n_bins, rwidth=0.95,
```

```
                                        density=True, alpha=0.8)

        # Fit a normal distribution to the data
        # with mean and standard deviation
        mu, std = norm.fit(df_scaled[feature])

        # Plot the probability density function (PDF)
        xmin, xmax = subplot_list[index_subplt].get_xlim()
        x = np.linspace(xmin, xmax, 100)
        p = norm.pdf(x, mu, std)
        subplot_list[index_subplt].plot(x, p, 'k', linewidth=2)

        title_concat = "{} (Mean: {:.2f}, " \
                       "Std. deviation: {:.2f})".format(title, mu, std)
        subplot_list[index_subplt].set_title(title_concat)
        # Show grid
        subplot_list[index_subplt].grid(visible=True)
        # Hide grid behind the bars
        subplot_list[index_subplt].set_axisbelow(True)
        # Label x and y-axis
        subplot_list[index_subplt].set_xlabel('{} value range [cm]'
                                              .format(scaling_type))
        subplot_list[index_subplt].set_ylabel('frequency density')

    plt.show()
```

Call the new function `func_histograms_comp_scaling()` to create the **histograms** with overlaid **probability density functions** comparing **original data** with the **normalized data** side-by-side:

[423]:
```
features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
titles =   ['Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width']

func_histograms_comp_scaling(irisdata_df, irisdata_df_norm,
                             features, titles, 'normalized')
```

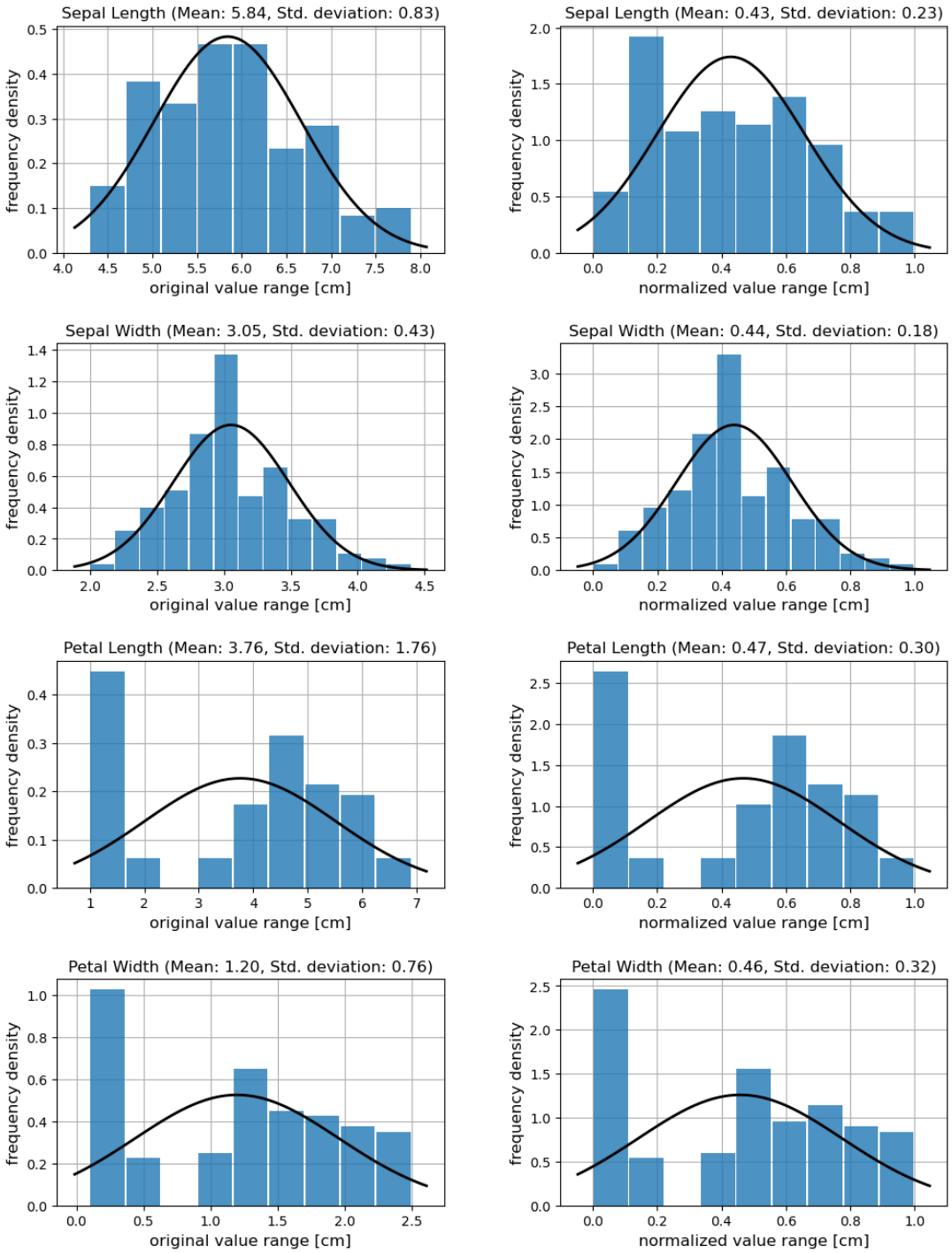


Figure 29: Histograms with overlaid probability density functions comparing original data (left) with the normalized data (right)

6.2.2 Standardization

Standardization is another scaling technique in which **values are centered around the mean** with a **unit standard deviation**. That is, the **mean** of the feature **becomes zero** and the resulting **distribution** has a **unit standard deviation** (see Bhandari 2020).

The **standardization** is calculated following this **formula**:

$$X' = \frac{X - \mu}{\sigma}$$

- μ is the **mean** of the feature values and
- σ is the **standard deviation** of the feature values.

It should be noted that the values after standardization are not limited to a certain range (unlike normalization).

For further details read here: scikit-learn: Standardization, or mean removal and variance scaling.

[424]:
```
from sklearn.preprocessing import StandardScaler

# Fit the standard scaler on raw input dataframe
# by selecting columns 1-4 with all feature rows
#std_scaler = StandardScaler().fit(irisdata_df.iloc[:, 0:4])
# by ommitting the last column with class names
std_scaler = StandardScaler().fit(irisdata_df.drop('species', axis=1))

# Transform the raw dateframe with fitted scaler
# and convert it to numpy array
#irisdata_np_std = std_scaler.transform(irisdata_df.iloc[:, 0:4])
irisdata_np_std = std_scaler.transform(irisdata_df.drop('species', axis=1))

#irisdata_np_std
```

[425]:
```
# Make a deep copy of original dataframe
irisdata_df_std = irisdata_df.copy(deep=True)

# Replace values of dataframe with standardized values from array
irisdata_df_std.iloc[:, 0:4] = irisdata_np_std

#irisdata_df_std
```

[426]:
```
irisdata_df_std.describe()
```

[426]:

	sepal_length	sepal_width	petal_length	petal_width
count	1.500000e+02	1.500000e+02	1.500000e+02	1.500000e+02
mean	-2.775558e-16	-5.140333e-16	1.154632e-16	9.251859e-16
std	1.003350e+00	1.003350e+00	1.003350e+00	1.003350e+00
min	-1.870024e+00	-2.438987e+00	-1.568735e+00	-1.444450e+00
25%	-9.006812e-01	-5.877635e-01	-1.227541e+00	-1.181504e+00
50%	-5.250608e-02	-1.249576e-01	3.362659e-01	1.332259e-01
75%	6.745011e-01	5.692513e-01	7.627586e-01	7.905908e-01
max	2.492019e+00	3.114684e+00	1.786341e+00	1.710902e+00

As in the previous section, the **original** and the **standardized data** are plotted as side-by-side **boxplots** with all **features at one scale**.

[427]:
```
titles     = ['Original', 'Standardized']
dataframes = [irisdata_df, irisdata_df_std]

func_boxplots_comp_scaling(dataframes, titles)
```

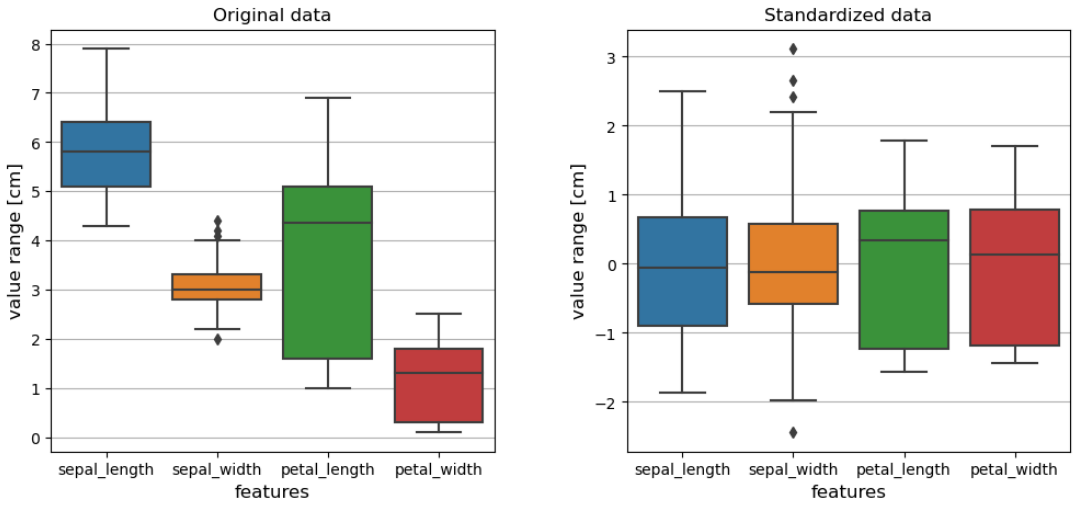


Figure 30: Boxplots comparing the original data (left) with the standardized data (right) with all features in one scale

As in the previous section, the **original** and the **standardized data** are plotted as side-by-side **histograms** with overlaid **probability density functions**.

[428]:
```
features = ['sepal_length', 'sepal_width', 'petal_length', 'petal_width']
titles =   ['Sepal Length', 'Sepal Width', 'Petal Length', 'Petal Width']

func_histograms_comp_scaling(irisdata_df, irisdata_df_std,
                             features, titles, 'standardized')
```

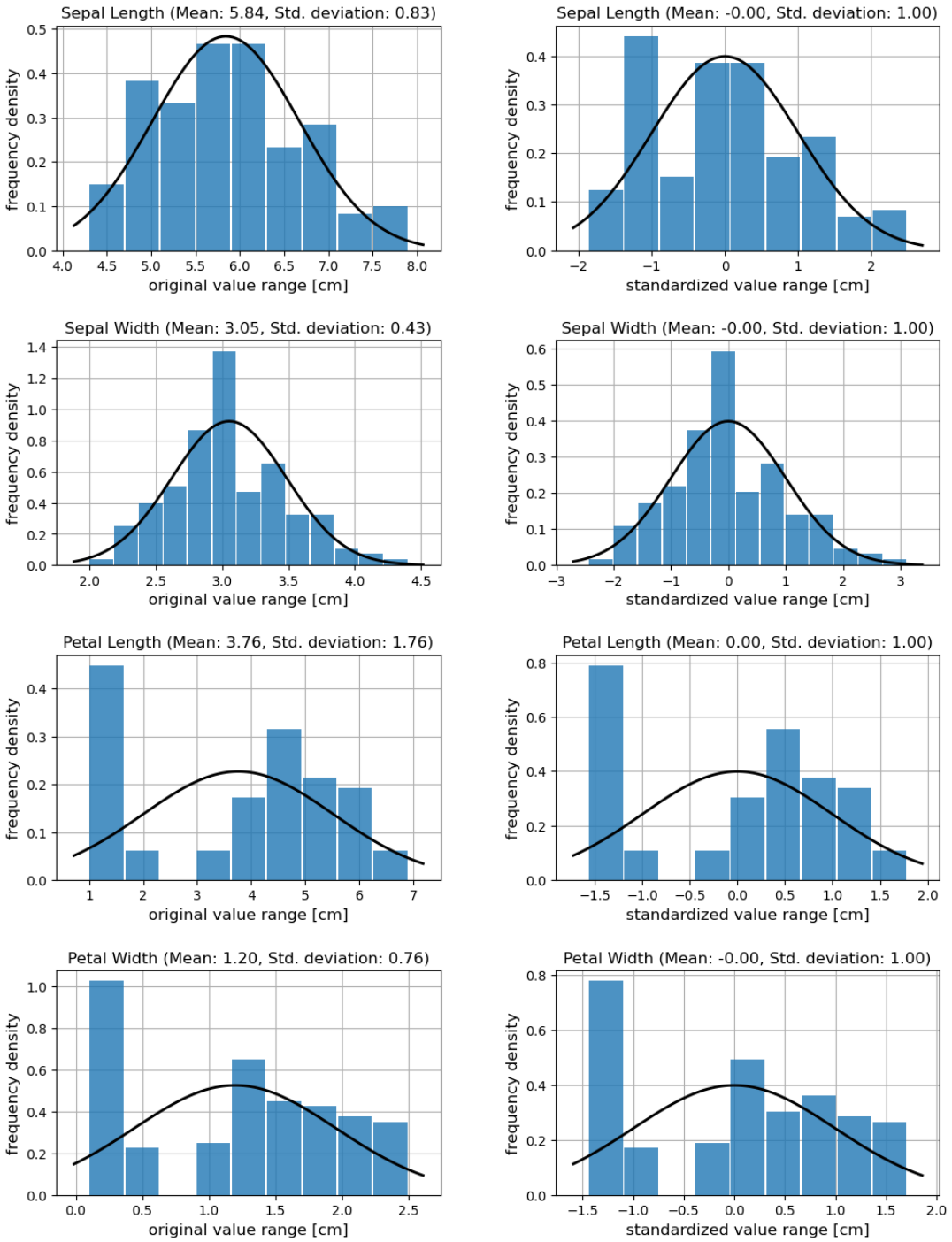


Figure 31: Histograms with overlaid probability density functions comparing original data (left) with the standardized data (right)

7 STEP 5: Carry out training, prediction and testing

To avoid errors, the Iris dataset is imported again:

[86]:
```
# Import ORIGINAL Iris dataset for classification
#irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle.csv')

# Import CLEANED Iris dataset for classification (removed duplicates)
```

```
irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle_cleaned.csv')

# Import NOISED Iris dataset for classification
#irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle_noised.csv')
```

7.1 Split the dataset

In the next very important step, the dataset is split into **2 subsets**: a **training dataset** and a **test dataset**. As the names suggest, the training dataset is used to train the ML algorithm. The test dataset is then used to check the quality of the trained ML algorithm (here the **recognition rate**). For this purpose, the **class labels** are **removed** from the training dataset - after all, these are to be predicted.

Typically, the **test dataset** should contain about **20%** of the entire dataset.

In particular, to **avoid bias** in the sorted Iris dataset due to splitting, the **order** of the data rows must be **randomized**. This is done with the parameter `shuffle=True`.

[87]:
```
from sklearn.model_selection import train_test_split

X = irisdata_df.drop('species', axis=1)
y = irisdata_df['species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,␣
  ↪shuffle=True)
```

Check that the split datasets are still balanced and that no **bias** has been created by the splitting.

For this test, the previously separated labels `y_train` must be added back to the training dataset `X_train`.

[88]:
```
# Make a deep copy of 'X_train'
X_train_bias_test_df = X_train.copy(deep=True)

# Add list of labels to test dataframe
X_train_bias_test_df['species'] = y_train

# Count unique values without missing values in a column,
# ordered descending and not normalized
X_train_bias_test_df['species'].value_counts(ascending=True, dropna=False,␣
  ↪normalize=False)
```

[88]:
```
Iris-setosa        38
Iris-virginica     39
Iris-versicolor    40
Name: species, dtype: int64
```

For training, do not use only the variables that correlate best with each other, but all of them.

Otherwise, the result of the prediction would be significantly worse. Maybe this is already an indication of **overfitting** of the ML model.

[89]:
```
# DO NOT USE THIS!!
#X_train, X_test, y_train, y_test = train_test_split(X[['sepal_length',
#                                                       'sepal_width']],
#                                                    y, test_size = 0.20)
```

7.2 Standardize the datasets

Standardize the feature values by computing the **mean**, subtracting the mean from the data points, and then dividing by the **standard deviation**:

[90]:
```
from sklearn.preprocessing import StandardScaler

std_scaler = StandardScaler()
X_train = std_scaler.fit_transform(X_train)
X_test = std_scaler.transform(X_test)

#X_train
```

7.3 Train the SVC

In this step the SVC is trained with the training data. Training means to **fit** the SVC to the **training data**.

[91]:
```
# fit the model for the data
classifier.fit(X_train, y_train)
```

[91]:
```
SVC(kernel='linear', random_state=0)
```

7.4 Make predictions

In this step the aim is to **predict the species** using unlabeled test data.

[92]:
```
y_pred = classifier.predict(X_test)
#X_test
#y_pred
```

8 STEP 6: Evaluate model's performance

Subsequently to the training of the SVC model and the classification predictions made based on the test data, this step evaluates the **quality of the classification result** using known **metrics** such as the **accuracy score**, the **cross-validation score** as well as the **confusion matrix**.

8.1 Accuracy Score

In a multilabel classification (such as the Iris dataset), this **accuracy classification score** computes the subset accuracy. It means, that the set of **labels predicted** in `y_pred` for a sample must exactly match the corresponding set of **true labels** in `y_test`. For further details see sklearn.metrics.accuracy_score.

[93]:
```
from sklearn.metrics import accuracy_score

acc_score = accuracy_score(y_test, y_pred)

print("Accuracy score: {:.2f} %".format(acc_score.mean()*100))
```

```
Accuracy score: 93.33 %
```

8.2 Classification Report

The **classification report** shows a representation of the most important **classification metrics** on a **class-by-class basis**. This gives a **better understanding** of the behaviour of the classifier than **global**

accuracy, which can mask functional weaknesses in one class of a multi-class problem (see Classification Report (https://www.scikit-yb.org/en/latest/api/classifier/classification_report.html)).

The metrics are defined in the form of **true/false positives** and **true/false negatives** as follows:

- **precision:** is the metric for the **accuracy** of a classifier. For each class, it is defined as the **ratio** of **true positives** to the **sum of true and false positives**.
- **recall:** is a metric for the **completeness** of the classifier, i.e. the ability of a classifier to find **all positive instances correctly**. For each class, it is defined as the **ratio** of **true positives** to the **sum of true positives and false negatives**.
- **f1-score:** is a **weighted harmonic mean of precision and recall**, with the best value being 1.0 and the worst 0.0.
- **support:** is the **number of actual occurrences of the class** in the specified data set. This metric is an indication of the **balance of the class distribution** in the test dataset.

[94]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        10
Iris-versicolor       1.00      0.80      0.89        10
 Iris-virginica       0.83      1.00      0.91        10

       accuracy                           0.93        30
      macro avg       0.94      0.93      0.93        30
   weighted avg       0.94      0.93      0.93        30
```

8.3 Cross-validation score

The previous evaluations by the **accuracy classification score** and the **classification report** were carried out after a **manual classification**. For this purpose, the complete Iris dataset was first split into a larger **training dataset** and a much smaller **test dataset** with the function `train_test_split()`. Then the **SVC model** was **trained** with the training dataset and **validated** with the test dataset.

For **automatic classification** with subsequent **validation** Scikit-learn provides the function `cross_val_score()`. This performs **n-times cross-validation**. First, the Iris dataset is randomly split into several different **subsets** (called *folds*). The number of subsets is determined by the parameter `cv` (here e.g. `cv = 10`). Subsequently, the **SVC model** is **trained** and **evaluated** several times in succession according to the number of subsets (here 10 times). In each run, always a different subset (fold) is used for validation, while training is performed on the remaining (here nine) folds. The **result** is an **array with ten scores** from the **evaluation**. (see Géron 2018).

The **cross-validation** method **trains** and **validates** a model over **multiple runs** according to the number of subsets. This leads to a better **understanding of model performance** over the **entire dataset** rather than just a single train/test split (see Allwright 2022).

In the following code example, the cross validation is determined over 10 runs (`cv = 10`). From the 10 resulting scores in the array a **mean** is formed and additionally the **standard deviation** is given.

[95]:
```
from sklearn.model_selection import cross_val_score

irisdata_df_wo_labels = irisdata_df.drop('species', axis=1)
irisdata_labels = irisdata_df['species']

accuracies = cross_val_score(estimator = classifier, X = irisdata_df_wo_labels,
                             y = irisdata_labels, cv = 10)
```

```
print("Cross-validation score: {:.2f} %".format(accuracies.mean()*100))
print("Standard Deviation: {:.2f} %".format(accuracies.std()*100))
```

```
Cross-validation score: 97.33 %
Standard Deviation: 4.42 %
```

8.4 Confusion matrix

The **confusion matrix** measures the **quality of predictions** from a classification model by looking at how many **predictions** are **True** and how many are **False** (see What the Confusion Matrix Measures?).

The confusion matrix (also called error matrix) is a special table layout that visualizes the performance of a classification algorithm. Each **row** of the matrix represents the instances in a **true class**, while each **column** represents the instances in a **predicted class**, or vice versa - both variants can be found in the literature. Using this matrix, it is easy to see if there has been confusion between classes during classification - which is where the name comes from.

8.4.1 Textual confusion matrix

For checking the accuracy of the model, the **confusion matrix** can be used for the **cross validation**.

By using the function `sklearn.metrics.confusion_matrix()` a **confusion matrix** of the **true Iris class labels** versus the **predicted class labels** is plotted.

[96]:
```
cm = metrics.confusion_matrix(y_test, y_pred)
print(cm)
```

```
[[10  0  0]
 [ 0  8  2]
 [ 0  0 10]]
```

8.4.2 Colored confusion matrix

The function `sklearn.metrics.ConfusionMatrixDisplay()` plots a colored confusion matrix.

[97]:
```
sns.set_style("white")

# print colored confusion matrix
cm_colored = metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)

cm_colored.figure_.suptitle("Colored Confusion Matrix")
cm_colored.figure_.set_figwidth(8)
cm_colored.figure_.set_figheight(7)

cm_colored.confusion_matrix

# save figure as PNG
plt.tight_layout()
plt.savefig('images/confusion_matrix.png', dpi=150, pad_inches=5)
plt.show()
```

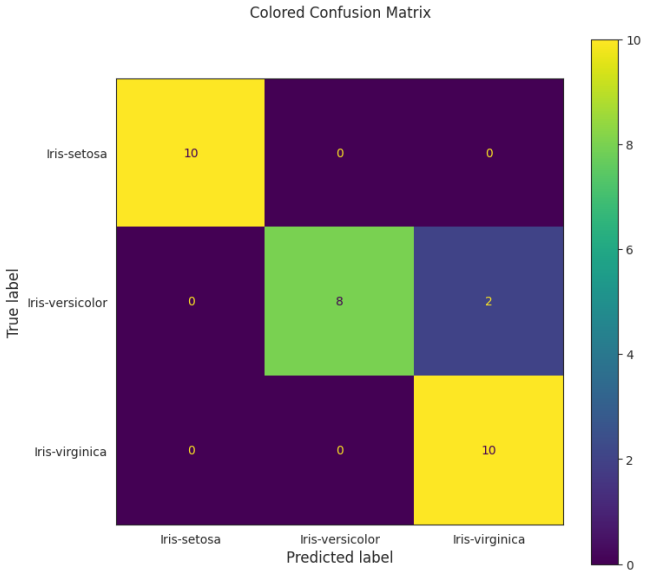


Figure 32: Checking the accuracy of the model by using the confusion matrix for cross-validation

9 STEP 7: Vary parameters of the ML model manually

This section was inspired by In Depth: Parameter tuning for SVC

In this section, the 4 SVC parameters `kernel`, `gamma`, `C` and `degree` will be introduced one by one. Furthermore, their influence on the classification result by varying these single parameters will be shown.

Disclaimer: In order to show the effects of varying the individual parameters in 2D graphs, only the best correlating variables `petal_length` and `petal_width` are used to train the SVC.

9.1 Prepare dataset

[21]:
```
# Import packages
from sklearn.svm import SVC
#from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
import numpy as np
```

To avoid errors, the Iris dataset is imported again:

[22]:
```
# Import ORIGINAL Iris dataset for classification
#irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle.csv')

# Import NOISED Iris dataset for classification
irisdata_df = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle_noised.csv')
```

Encode the class column from class strings to integer equivalents:

[23]:
```
irisdata_df_enc = irisdata_df.replace({"species":  {"Iris-setosa":0,
                                                    "Iris-versicolor":1,
                                                    "Iris-virginica":2}})
#irisdata_df_enc
```

9.1.1 Prepare datasets for parameter variation and plotting

These datasets will be used for parameter variation and plotting only. In particular, for later **2D plotting** of the effects of parameter variation, only **2 variables** of the Iris dataset can be used.

However, as seen in the previous section, this selection is very much at the expense of detection accuracy. Therefore, it is not useful to make predictions with this subset of data - it is not necessary to divide it into a training and a test dataset.

[24]:
```
# Copy only 2 feature columns
# and convert pandas.DataFrame to numpy array
X_plot = irisdata_df_enc[['petal_length', 'petal_width']].to_numpy(copy=True)
#X_plot = irisdata_df_enc[['sepal_length', 'sepal_width']].to_numpy(copy=True)
#X_plot
```

[25]:
```
# Convert pandas.DataFrame to numpy array
# and get a flat 1D copy of 2D numpy array
y_plot = irisdata_df_enc[['species']].to_numpy(copy=True).flatten()
#y_plot
```

9.1.2 Prepare dataset for prediction and evaluation

To **evaluate the recognition accuracy** by parameter variation, the complete Iris dataset with all variables must be used. To make predictions with test data, the dataset is again divided into a training and a test dataset.

[26]:
```
X = irisdata_df.drop('species', axis=1)
y = irisdata_df['species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,␣
 ↪shuffle=True)
#X_train
```

Normalize the feature values for **prediction and evaluation**. Normalisation is deliberately used here to avoid visualisation problems due to negative values.

[27]:
```
norm_scaler_pred = MinMaxScaler()
X_train = norm_scaler_pred.fit_transform(X_train)
X_test = norm_scaler_pred.transform(X_test)
```

9.2 Plotting functions

This function helps to visualize the modifications by varying the individual SVC parameters:

[28]:
```
def plotSVC(title, svc, X, y, xlabel, ylabel, subplot):
    # create a mesh to plot in
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1

    # prevent division by zero
    if x_min == 0.0:
        x_min = 0.1

    h = (x_max / x_min)/1000
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

    Z = svc.predict(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
```

```
    ax = subplot

    ax.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.6)
    ax.scatter(X[:, 0], X[:, 1], c=y, cmap=plt.cm.Paired)
    ax.set_xlabel(xlabel)
    ax.set_ylabel(ylabel)
    ax.set_xlim(xx.min(), xx.max())
    ax.set_title(title)
    ax.grid(visible=False)
```

This function cares for cross validation:

[29]:
```
def crossValSVC(X_train, y_train, kernel='rbf', gamma='scale', C=1.0, degree=3):
    # train the SVC
    svc = svm.SVC(kernel=kernel,
                  gamma=gamma,
                  C=C,
                  degree=degree).fit(X_train, y_train)
    # calculate accuracies
    accuracies = cross_val_score(estimator = svc, X = X_train,
                                 y = y_train, cv = 10)

    accuracy = accuracies.mean()*100
    return accuracy
```

This function plots the variation of the SVC parameters against the prediction accuracy to show the effect of variation and its limits regarding the phenomenon **overfitting**:

[30]:
```
def plotParamsAcc(param_list, acc_list, param_name, log_scale=False):
    fig, ax = plt.subplots(figsize=(16, 6))
    title_str = 'Variation of {} parameter '.format(param_name) \
                +'and its effect to prediction accuracy'
    plt.title(title_str)
    ax.plot(param_list, accuracy_list)
    if log_scale:
        # set the X axis scale to logarithmic
        ax.set_xscale('log')
    plt.xlabel(param_name)
    plt.ylabel('accuracy [%]')
    plt.grid(visible=True)
    plt.show()
```

9.3 Vary `kernel` of SVC

The `kernel` parameter selects the type of hyperplane that is used to separate the data. Using `linear` (linear classifier) kernel will use a linear hyperplane (a line in the case of 2D data). The `rbf` (radial basis function kernel) and `poly` (polynomial kernel) kernel use non linear hyperplanes. The **default** is `kernel=rbf`.

[31]:
```
kernels = ['linear', 'rbf', 'poly', 'sigmoid']

xlabel = 'Petal length'
ylabel = 'Petal width'

# Setup 2x2 grid for plotting
fig, subplots = plt.subplots(2, 2, figsize=(16, 12))
# Set margins between subplots
```

```
plt.subplots_adjust(wspace=0.3, hspace=0.3)

# Make subplots iterable via 'subplots.flatten()'
for kernel, subplot in zip(kernels, subplots.flatten()):
    svc_plot = svm.SVC(kernel=kernel).fit(X_plot, y_plot)
    accuracy = crossValSVC(X_train, y_train, kernel=kernel)
    title_str = 'kernel: \''+str(kernel)+'\', '+'Acc. prediction: {:.2f}%'.
  ↪format(accuracy)
    plotSVC(title_str, svc_plot, X_plot, y_plot, xlabel, ylabel, subplot)

plt.show()
```

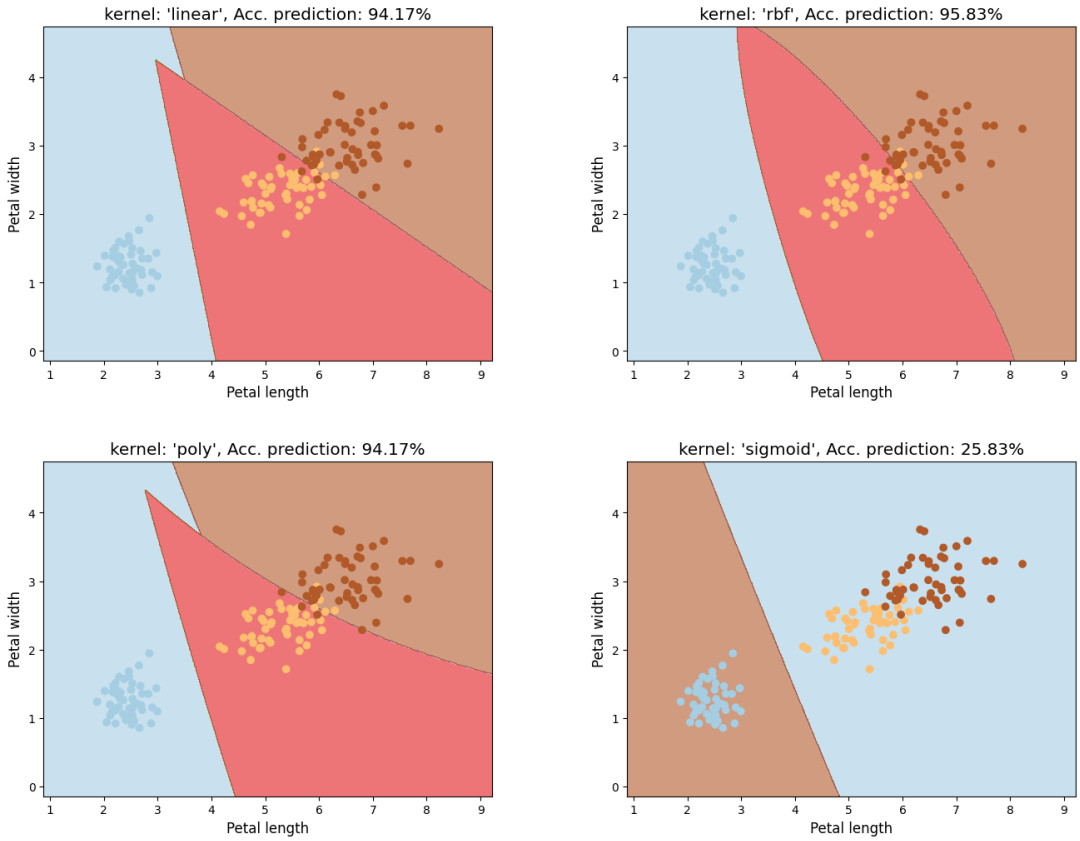


Figure 33: This group of images shows the effect on the classification by the choice of the different SVC kernels ('linear', 'rbf', 'poly' and 'sigmoid')

9.4 Vary `gamma` parameter

The `gamma` parameter is used for **non linear hyperplanes**. The higher the `gamma` float value it tries to exactly fit the training dataset. The **default** is `gamma='scale'`.

[32]:
```
gammas = [0.1, 0.3, 0.5, 1, 10, 100]

xlabel = 'Petal length'
ylabel = 'Petal width'

# Setup 2x3 grid for plotting
fig, subplots = plt.subplots(3, 2, figsize=(16, 19))
# Set margins between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.3)
```

```
# Make subplots iterable via 'subplots.flatten()'
for gamma, subplot in zip(gammas, subplots.flatten()):
    svc_plot = svm.SVC(kernel='rbf', gamma=gamma).fit(X_plot, y_plot)
    accuracy = crossValSVC(X_train, y_train, kernel='rbf', gamma=gamma)
    title_str = 'gamma: \''+str(gamma)+'\', ' \
                +'Acc. prediction: {:.2f}%'.format(accuracy)
    plotSVC(title_str, svc_plot, X_plot, y_plot, xlabel, ylabel, subplot)

plt.show()
```

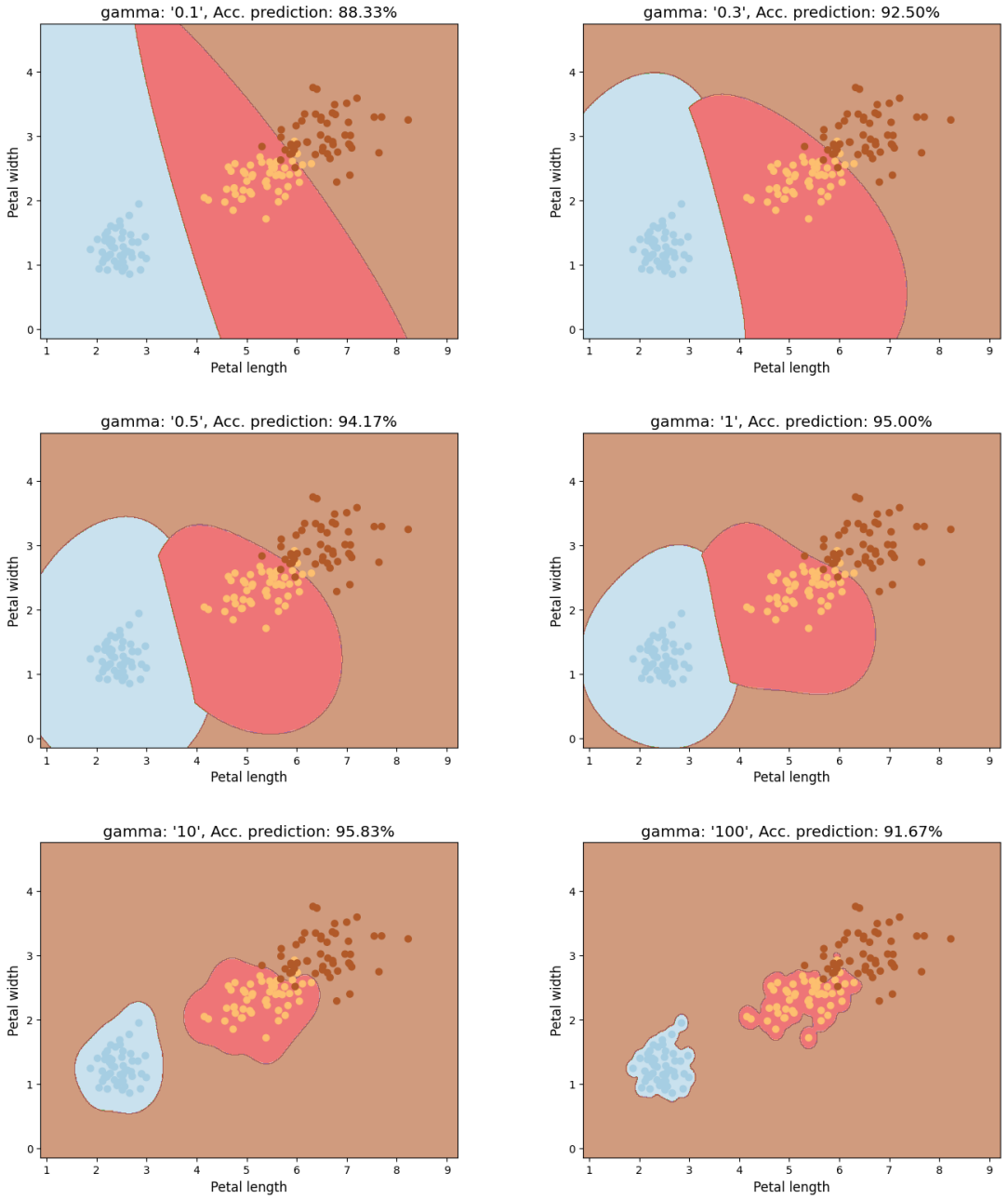


Figure 34: This group of images shows the effect on the classification by the variation of the parameter 'gamma' of the 'rbf' kernel

Show the variation of the SVC parameter `gamma` against the **prediction accuracy**.

As we can see, increasing `gamma` leads to **overfitting** as the classifier tries to perfectly fit the training data.

[33]:
```
gammas = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 10, 100, 200]

accuracy_list = list()
for gamma in gammas:
    accuracy = crossValSVC(X_train, y_train, kernel='rbf', gamma=gamma)
    accuracy_list.append(accuracy)

plotParamsAcc(gammas, accuracy_list, 'gamma', log_scale=True)
```

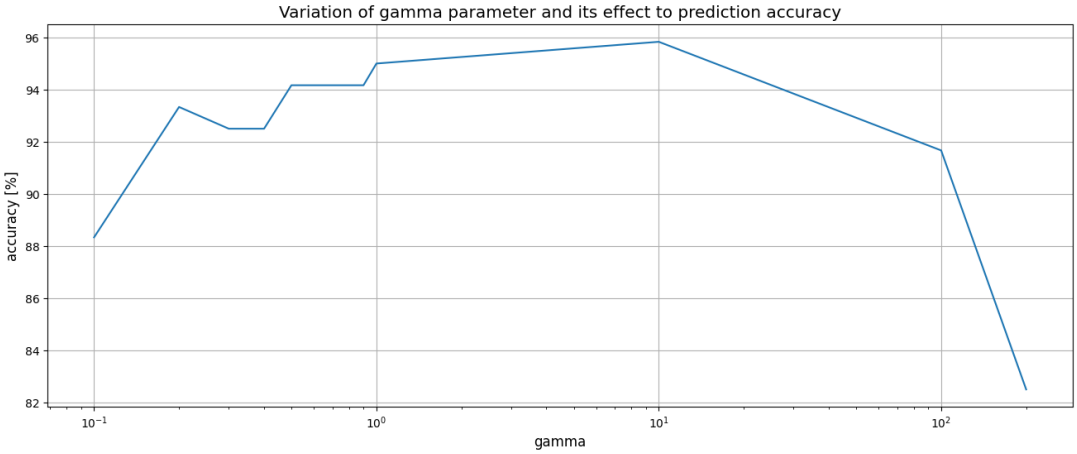


Figure 35: The plot shows the variation of the SVC parameter 'gamma' against the prediction accuracy

9.5 Vary `C` parameter

The `C` parameter is the **penalty** of the error term. It controls the trade off between smooth decision boundary and classifying the training points correctly. The **default** is `C=1.0`.

[34]:
```
cs = [0.1, 1, 5, 10, 100, 1000]

xlabel = 'Petal length'
ylabel = 'Petal width'

# Setup 2x3 grid for plotting
fig, subplots = plt.subplots(3, 2, figsize=(16, 19))
# Set margins between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.3)

# Make subplots iterable via 'subplots.flatten()'
for c, subplot in zip(cs, subplots.flatten()):
    svc_plot = svm.SVC(kernel='rbf', C=c).fit(X_plot, y_plot)
    accuracy = crossValSVC(X_train, y_train, kernel='rbf', C=c)
    title_str = 'C: \''+str(c)+'\', ' \
                +'Acc. prediction: {:.2f}%'.format(accuracy)
    plotSVC(title_str, svc_plot, X_plot, y_plot, xlabel, ylabel, subplot)

plt.show()
```

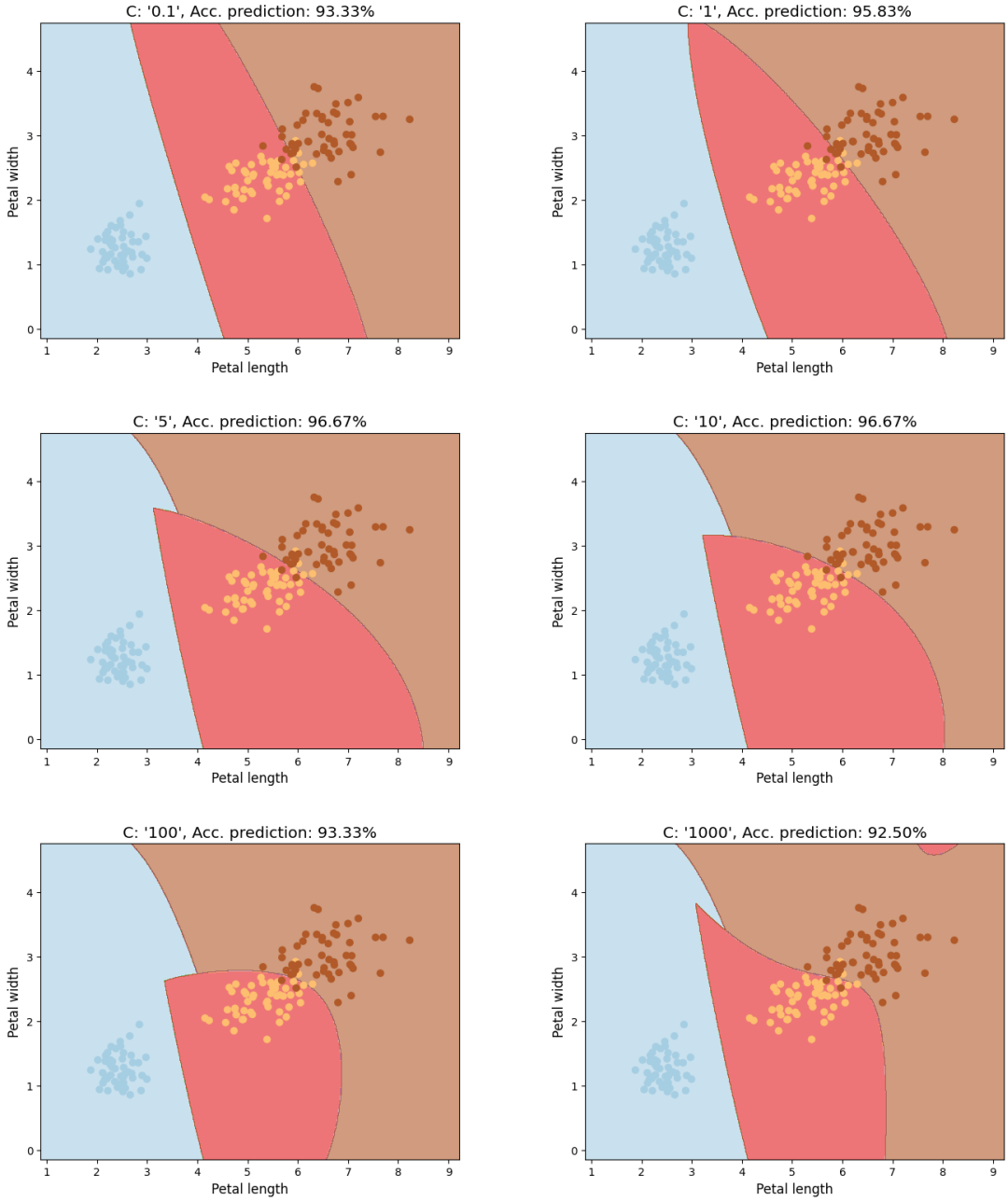


Figure 36: This group of images shows the effect on the classification by the variation of the parameter 'C' of the 'rbf' kernel

Show the variation of the SVC parameter `C` against the **prediction accuracy**.

But be careful: to high `C` values may lead to **overfitting** the training data.

[35]:
```
cs = [0.1, 1, 5, 6, 7, 8, 10, 100, 1000, 10000]

accuracy_list = list()
for c in cs:
    accuracy = crossValSVC(X_train, y_train, kernel='rbf', C=c)
    accuracy_list.append(accuracy)

plotParamsAcc(cs, accuracy_list, 'C', log_scale=True)
```

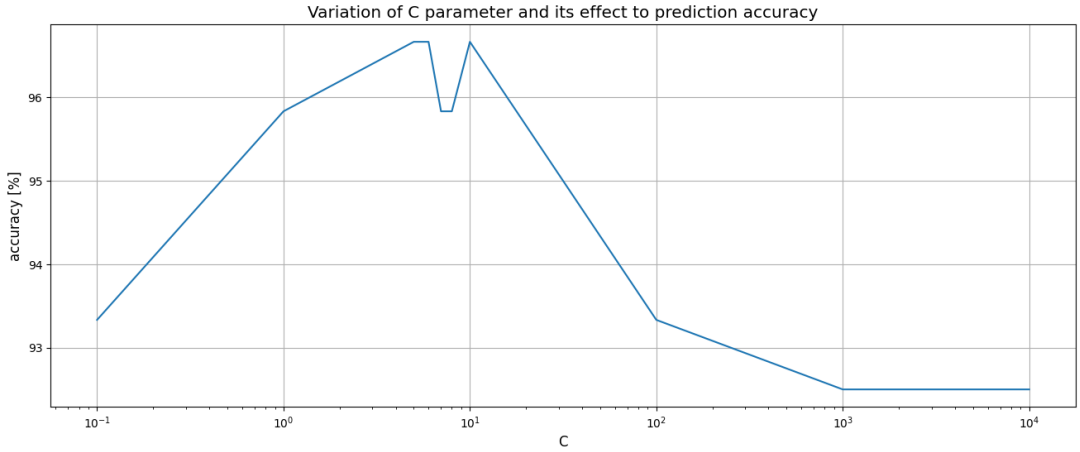


Figure 37: The plot shows the variation of the SVC parameter 'C' against the prediction accuracy

9.6 Vary `degree` parameter

The `degree` parameter is used when the `kernel` is set to `poly` and is ignored by all other kernels. It's basically the **degree of the polynomial** used to find the hyperplane to split the data. The **default** is `degree=3`.

Using `degree = 1` is the same as using a `linear` kernel. Also, increasing this parameters leads to **higher training times**.

[36]:
```
degrees = [1, 2, 3, 4, 5, 6]

xlabel = 'Petal length'
ylabel = 'Petal width'

# Setup 2x3 grid for plotting
fig, subplots = plt.subplots(3, 2, figsize=(16, 19))
# Set margins between subplots
plt.subplots_adjust(wspace=0.3, hspace=0.3)

# Make subplots iterable via 'subplots.flatten()'
for degree, subplot in zip(degrees, subplots.flatten()):
    svc_plot = svm.SVC(kernel='poly', degree=degree).fit(X_plot, y_plot)
    accuracy = crossValSVC(X_train, y_train, kernel='poly', degree=degree)
    title_str = 'degree: \''+str(degree)+'\', ' \
                +'Acc. prediction: {:.2f}%'.format(accuracy)
    plotSVC(title_str, svc_plot, X_plot, y_plot, xlabel, ylabel, subplot)

plt.show()
```

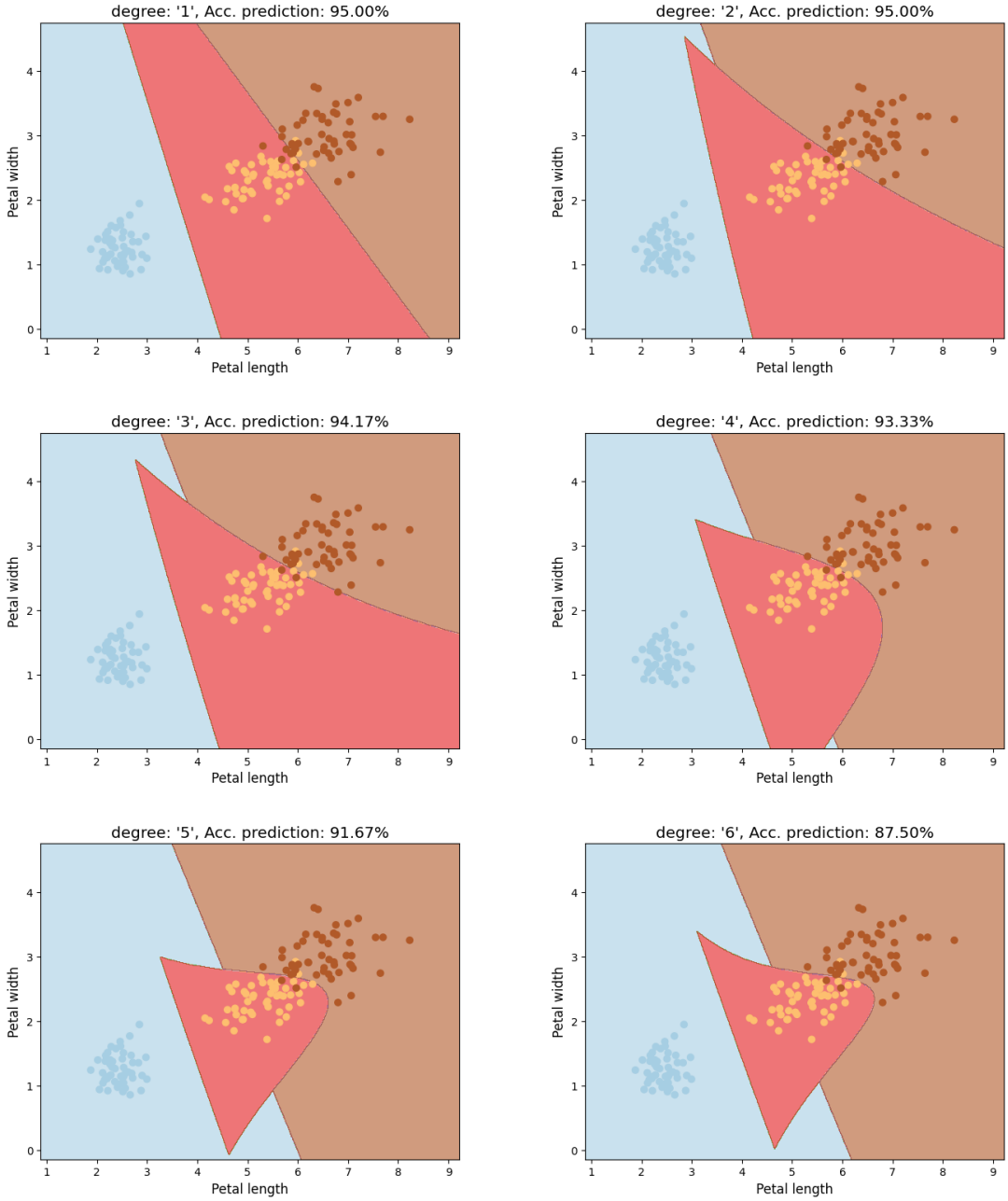


Figure 38: This group of images shows the effect on the classification by the variation of the parameter 'degree' of the 'poly' kernel

Show the variation of the SVC parameter `degree` against the **prediction accuracy**.

As we can see, increasing the `degree` of the polynomial hyperplane leads to **overfitting** the training data.

[37]:
```
degrees = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

accuracy_list = list()
for degree in degrees:
    accuracy = crossValSVC(X_train, y_train, kernel='poly', degree=degree)
    accuracy_list.append(accuracy)

plotParamsAcc(degrees, accuracy_list, 'degree', log_scale=False)
```

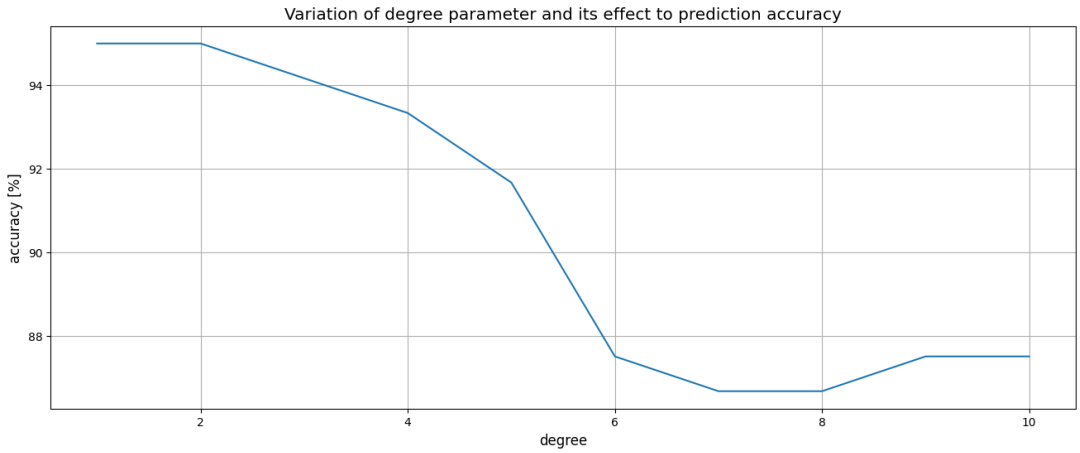


Figure 39: The plot shows the variation of the SVC parameter 'degree' against the prediction accuracy

10 STEP 8: Tune the ML model systematically

In the final step, two approaches to systematic hyper-parameter search are presented: **Grid Search** and **Randomized Search**. While the former exhaustively considers all parameter combinations for given values, the latter selects a number of candidates from a parameter space with a particular random distribution.

Sources:

- 3.2. Tuning the hyper-parameters of an estimator
  - sklearn.model_selection.GridSearchCV
  - sklearn.model_selection.RandomizedSearchCV
- Introduction to hyperparameter tuning with scikit-learn and Python
  - Abalone Dataset
- Hyperparameter tuning using Grid Search and Random Search: A Conceptual Guide

Import the necessary packages:

[15]:
```
# general packages
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.model_selection import cross_val_score
from sklearn.metrics import accuracy_score
from sklearn.metrics import classification_report
#from sklearn.svm import SVC
from sklearn import svm, metrics
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
%matplotlib inline

# additional packages for grid search
from sklearn.model_selection import RepeatedKFold
from sklearn.model_selection import GridSearchCV

# additional packages for randomized search
from sklearn.model_selection import RandomizedSearchCV
from sklearn.model_selection import RepeatedKFold

# import class MeasExecTimeOfProgram from python file MeasExecTimeOfProgramclass.py
from MeasExecTimeOfProgram_class import MeasExecTimeOfProgram
```

Set path and columns of the Iris dataset for import:

[94]:
```
# Path of the ORIGINAL Iris dataset for classification
#CSV_PATH = "./datasets/IRIS_flower_dataset_kaggle.csv"

# Path of the NOISED Iris dataset for classification
CSV_PATH = "./datasets/IRIS_flower_dataset_kaggle_noised.csv"
```

Load dataset and split it into subsets for training and testing in the ratio 80% to 20%:

[95]:
```
# load the dataset, separate the features and labels, and perform a
# training and testing split using 80% of the data for training and
# 20% for evaluation
irisdata_df = pd.read_csv(CSV_PATH)

X = irisdata_df.drop('species', axis=1)
y = irisdata_df['species']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.20,␣
 ↪shuffle=True)
```

Check that the split datasets are still balanced and that no **bias** has been created by the splitting.

For this test, the previously separated labels `y_train` must be added back to the training dataset `X_train`.

[96]:
```
# make a deep copy of 'X_train'
X_train_bias_test_df = X_train.copy(deep=True)

# add list of labels to test dataframe
X_train_bias_test_df['species'] = y_train

# count unique values without missing values in a column,
# ordered descending and normalized
X_train_bias_test_df['species'].value_counts(ascending=False, dropna=False,␣
 ↪normalize=True)
```

[96]:
```
Iris-virginica     0.341667
Iris-versicolor    0.341667
Iris-setosa        0.316667
Name: species, dtype: float64
```

Standardize the feature values by computing the **mean**, subtracting the mean from the data points, and then dividing by the **standard deviation**:

[97]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

#X_train
```

10.1 Finding a baseline

The aim of this sub-step is to establish a baseline on the Iris dataset by training a **Support Vector Classifier (SVC)** with no hyperparameter tuning.

Train the model with **no tuning of hyperparameters** to find the baseline for later improvements:

[98]:
```
classifier = svm.SVC(kernel = 'linear', random_state = 0)

# initiate measuring execution time
execTime = MeasExecTimeOfProgram()
execTime.start()

classifier.fit(X_train, y_train)

# print time delta
print('Execution time: {:.4f} ms'.format(execTime.stop()))
```

```
Execution time: 3.4351 ms
```

Evaluate our model using accuracy score:

[99]:
```
# predict labels
y_pred = classifier.predict(X_test)
```

[100]:
```
# calculate cross validation score
# HINT: do NOT use the accuracy score - it's to inaccurate!
accuracies = cross_val_score(estimator = classifier, X = X_train,
                             y = y_train, cv = 10)

print("Cross-validation score: {:.2f} %".format(accuracies.mean()*100))
print("Standard Deviation: {:.2f} %".format(accuracies.std()*100))
```

```
Cross-validation score: 95.83 %
Standard Deviation: 5.59 %
```

[101]:
```
# print classification report
print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        12
Iris-versicolor       0.90      1.00      0.95         9
 Iris-virginica       1.00      0.89      0.94         9

       accuracy                           0.97        30
      macro avg       0.97      0.96      0.96        30
   weighted avg       0.97      0.97      0.97        30
```

[102]:
```
sns.set_style("white")

# print colored confusion matrix
cm_colored = metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)

cm_colored.figure_.suptitle("Colored Confusion Matrix")
cm_colored.figure_.set_figwidth(8)
cm_colored.figure_.set_figheight(7)

cm_colored.confusion_matrix

plt.tight_layout()
plt.show()
```

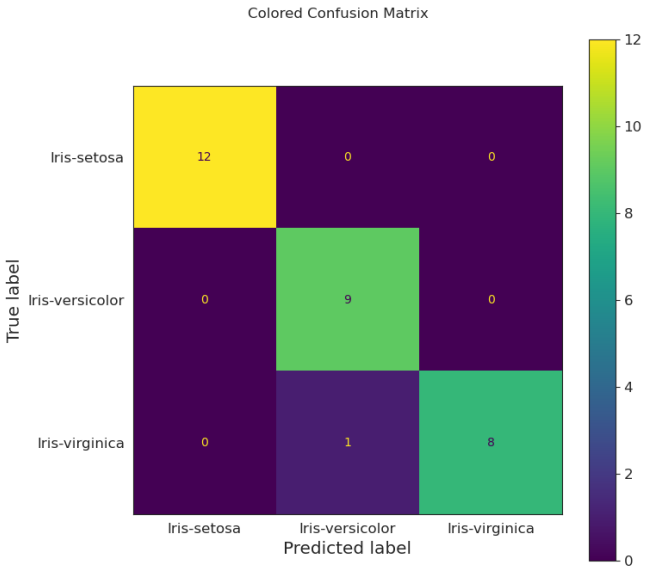


Figure 40: Confusion matrix for cross-validation of the baseline

[103]:
```
classifier.get_params()
```

[103]:
```
{'C': 1.0,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 0.0,
 'decision_function_shape': 'ovr',
 'degree': 3,
 'gamma': 'scale',
 'kernel': 'linear',
 'max_iter': -1,
 'probability': False,
 'random_state': 0,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

10.2 Add Gaussian noise to Iris dataset

Recording **datasets from real applications** is always associated with several problems. Real measured values are always subject to a certain **level of measurement noise**. Furthermore, when recording the measured values, there may be sporadic dropouts of the measurement sensors, which leads to **gaps in the dataset**. And finally, **doubles**, i.e. several identical measurements, can occur when merging several measurement series from different experiments.

These problems from the real measuring everyday life are to be shown by the example of the Iris dataset. Unfortunately, this dataset is a little "too perfect". To simulate **real measurement values**, some **Gaussian noise** with a defined **standard deviation** σ is added to the features of the Iris dataset. To simulate an **offset** due to **imperfectly calibrated measurement devices**, for example, the mean value could additionally be shifted. However, because this has **no influence on the classifiability**, it is omitted here.

[175]:
```
# Import Iris dataset for adding noise
irisdata_df_orig = pd.read_csv('./datasets/IRIS_flower_dataset_kaggle.csv')
```

First, determine the shape of the Iris dataset. The last column with the class names is omitted to get only the feature columns.

[176]:
```
# Get number of rows of the dataset
n_rows = irisdata_df_orig.shape[0]

# Get number of columns of the dataset
# Omit last column with the class names
n_cols = irisdata_df_orig.shape[1] - 1
```

Now a `numpy` array in the shape of the Iris dataset is generated. This contains **normally distributed random values** according to the **Gaussian curve** with a defined **standard deviation** σ. The **mean** of the Gaussian curve remains unchanged and is **not shifted** in the first place.

[177]:
```
# mean: "centre" of the distribution
# sigma: standard deviation (spread or "width") of the distribution
mean, sigma = 0, 0.2

# Create noise with the same dimension as the dataset
# Set 'seed' to something, to make the output of the random generator reproducible
np.random.seed(42)
irisdata_np_noise = np.random.normal(mean, sigma, (n_rows, n_cols))
```

The first 4 columns of the dataframe containing the original Iris dataset are converted to a `numpy` array.

[178]:
```
# Select columns 1-4 with all rows
# and convert it to numpy array
irisdata_np_orig = irisdata_df_orig.iloc[:, 0:4].to_numpy()
```

The array with the normally distributed random values is added to this.

[179]:
```
# Add noise to original values
irisdata_np_noised = irisdata_np_orig + irisdata_np_noise
```

Negative measured values do **not make sense** for this dataset and should therefore be avoided. Therefore, the **minimum value** over the entire array is first retrieved with the function `numpy.amin()`. If this is negative, an **integer offset** for shifting the data into the positive range is calculated from the rounded up amount of the minimum value. The function `math.ceil()` is used for this.

[180]:
```
import math

min_val = np.amin(irisdata_np_noised)
print("Minimal value of noised array: {}".format(min_val))

if (min_val < 0):
    min_val_abs = abs(min_val)

    # Round the min_val_abs upward to its nearest integer
    offset = math.ceil(min_val_abs)
else:
    offset = 0

print("Offset for array shifting: {}".format(offset))
```

```
Minimal value of noised array: -0.14617286328679105
Offset for array shifting: 1
```

The calculated **offset** is **added** to the array with the noisy measurements to **shift** the data into the **positive range**.

[181]:
```
irisdata_np_noised_shifted = irisdata_np_noised + offset
#irisdata_np_noised_shifted
```

Finally, a deep copy of the original dataframe is created. In the copy, the first 4 columns are replaced with the noisy features.

[182]:
```
# Make a deep copy of original dataframe
irisdata_df_noised = irisdata_df_orig.copy(deep=True)

# Replace values of dataframe with noisy values from array
irisdata_df_noised.iloc[:, 0:4] = irisdata_np_noised_shifted

irisdata_df_noised.describe()
```

[182]:

	sepal_length	sepal_width	petal_length	petal_width
count	150.000000	150.000000	150.000000	150.000000
mean	6.841439	4.048479	4.756247	2.197686
std	0.869432	0.473661	1.786644	0.771369
min	5.164616	2.977053	1.878503	0.853827
25%	6.131139	3.736824	2.630018	1.400227
50%	6.776529	3.967138	5.342632	2.376893
75%	7.468658	4.365999	6.113823	2.788387
max	9.037428	5.362868	8.228994	3.754139

To compare the original Iris dataset with its noisy copy, both dataframes are visualized in pairs plots.

[183]:
```
# Define a function to visualize data as pairs plots
def plotPairs(df, title):
    g = sns.pairplot(df, diag_kind="kde", hue='species',
                     palette='Dark2', height=2.5)

    g.map_lower(sns.kdeplot, levels=4, color=".2")
    # y .. padding between title and plot
    g.fig.suptitle(title, y=1.05)
    plt.show()
```

[184]:
```
title = 'Pairs plot of the ORIGINAL Iris dataset'
plotPairs(irisdata_df_orig, title)
```

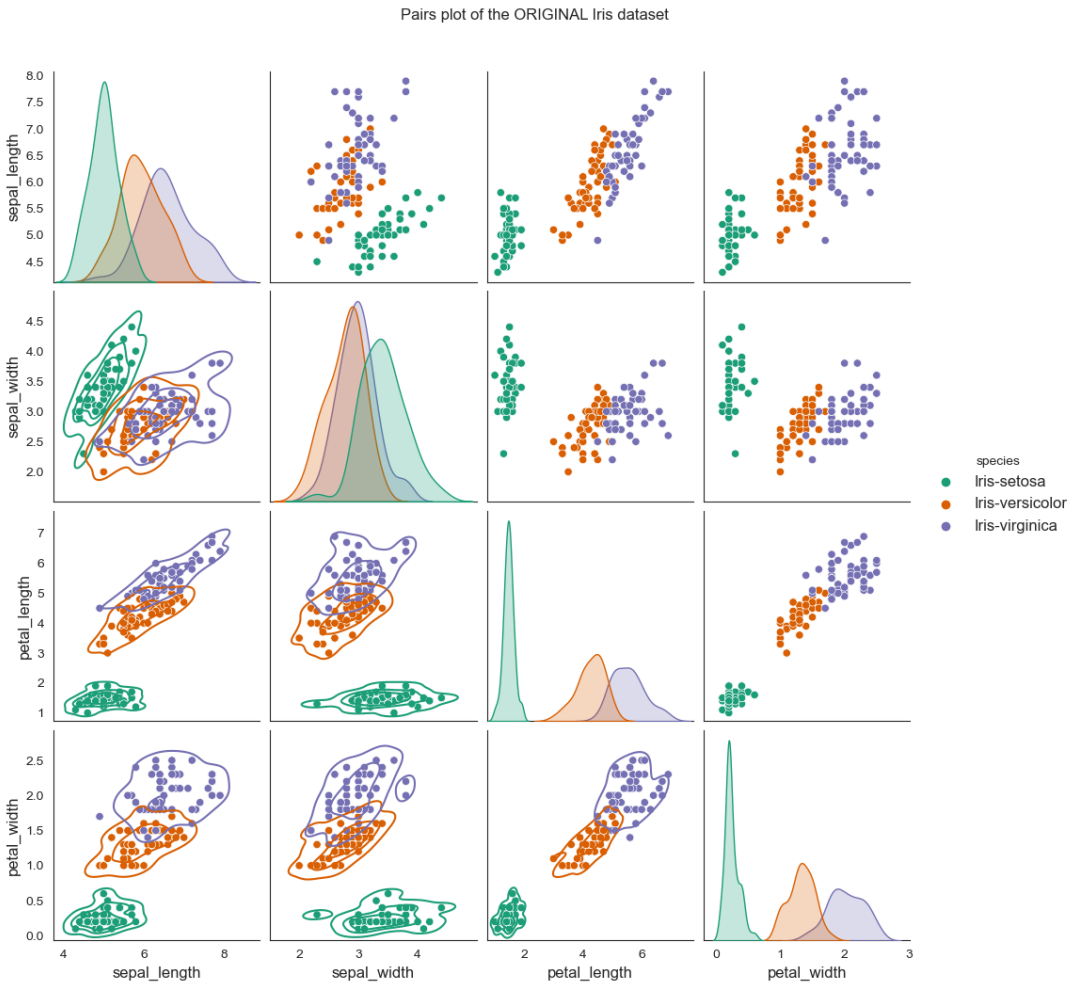


Figure 41: Pairs plot of the original Iris dataset

[185]:
```
title = 'Pairs plot of the NOISED Iris dataset'
plotPairs(irisdata_df_noised, title)
```

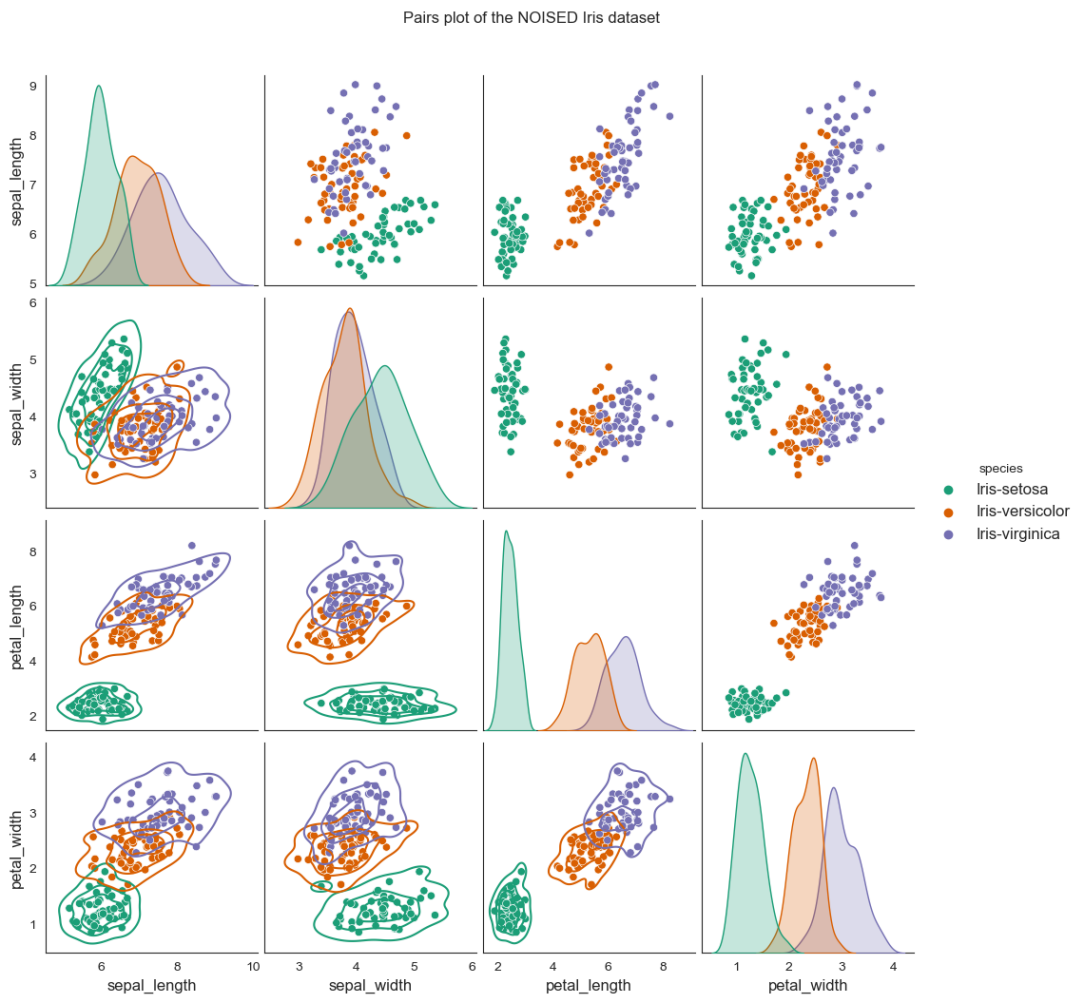


Figure 42: Pairs plot of the noised Iris dataset

Finally, the noisy Iris dataset is saved in its own CSV file.

[186]:
```
# Save noised Iris dataset to CSV file without index
csv_filepath = r'./datasets/IRIS_flower_dataset_kaggle_noised.csv'
irisdata_df_noised.to_csv(csv_filepath, sep=',', index=False)
```

10.3 Grid search

Initialize the SVC model and define the **space of the hyperparameters** to perform the **grid search** over:

[123]:
```
classifier = svm.SVC()

#kernels = ["linear", "rbf", "sigmoid", "poly"]
kernels = ["rbf", "poly"]
gammas = [0.1, 1, 10, 100, 200]
cs = [0.1, 1, 5, 10, 100, 1000]

# reduce the possible polynomial degrees to reasonable values,
# since with higher degrees the calculation time increases exponentially
#degrees = [1, 2, 3, 4, 5]
degrees = [1, 2, 3]
```

```
grid = dict(kernel=kernels, gamma=gammas, C=cs, degree=degrees)
```

Initialize a **cross-validation fold** and **perform a grid search** to tune the hyperparameters:

[124]:
```
cvFold = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

gridSearch = GridSearchCV(estimator=classifier, param_grid=grid, n_jobs=-1,
                          cv=cvFold, scoring="accuracy")

# initiate measuring execution time
execTime = MeasExecTimeOfProgram()
execTime.start()

searchResults = gridSearch.fit(X_train, y_train)

# Print execution time delta
execTime_sec = execTime.stop()/1000
execTime_str = time.strftime('%H h, %M min, %S sec', time.gmtime(execTime_sec))
#print('Execution time: {:.3f} s'.format(execTime_sec))
print('Execution time: {}'.format(execTime_str))
```

```
Execution time: 00 h, 16 min, 28 sec
```

Extract the best model and evaluate it:

[125]:
```
# predict labels by best model
bestModel = searchResults.best_estimator_

y_pred = bestModel.predict(X_test)
```

[126]:
```
# calculate cross validation score from the best model
# HINT: do NOT use the accuracy score - it's to inaccurate!
accuracies = cross_val_score(estimator = bestModel, X = X_train,
                             y = y_train, cv = 10)

print("Cross-validation score: {:.2f} %".format(accuracies.mean()*100))
print("Standard Deviation: {:.2f} %".format(accuracies.std()*100))
```

```
Cross-validation score: 97.50 %
Standard Deviation: 3.82 %
```

[127]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        12
Iris-versicolor       0.90      1.00      0.95         9
 Iris-virginica       1.00      0.89      0.94         9

       accuracy                           0.97        30
      macro avg       0.97      0.96      0.96        30
   weighted avg       0.97      0.97      0.97        30
```

[128]:
```
sns.set_style("white")

# print colored confusion matrix
cm_colored = metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)

cm_colored.figure_.suptitle("Colored Confusion Matrix")
cm_colored.figure_.set_figwidth(8)
cm_colored.figure_.set_figheight(7)

cm_colored.confusion_matrix

plt.tight_layout()
plt.show()
```

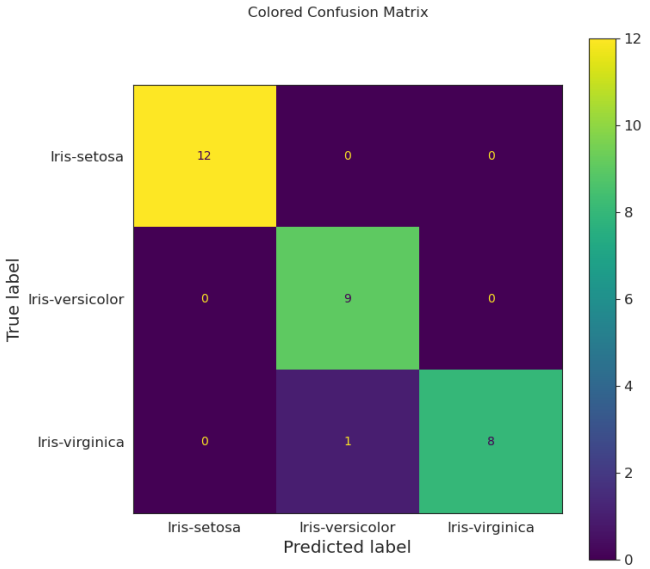


Figure 43: Confusion matrix for cross-validation after the grid search has been performed

[129]:
```
bestModel.get_params()
```

[129]:
```
{'C': 0.1,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
 'coef0': 0.0,
 'decision_function_shape': 'ovr',
 'degree': 1,
 'gamma': 100,
 'kernel': 'poly',
 'max_iter': -1,
 'probability': False,
 'random_state': None,
 'shrinking': True,
 'tol': 0.001,
 'verbose': False}
```

10.4 Randomized search

Initialize the SVC model and define the **space of the hyperparameters** to perform the **randomized search** over:

[104]:
```
classifier = svm.SVC()

#kernels = ["linear", "rbf", "sigmoid", "poly"]
kernels = ["rbf", "poly"]
gammas = [0.1, 1, 10, 100, 200]
cs = [0.1, 1, 5, 10, 100, 1000]

# reduce the possible polynomial degrees to reasonable values,
# since with higher degrees the calculation time increases exponentially
#degrees = [1, 2, 3, 4, 5]
degrees = [1, 2, 3]

grid = dict(kernel=kernels, gamma=gammas, C=cs, degree=degrees)
```

Initialize a **cross-validation fold** and **perform a randomized search** to tune the hyperparameters:

[117]:
```
cvFold = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

randomSearch = RandomizedSearchCV(estimator=classifier, n_jobs=-1,
                                  cv=cvFold, param_distributions=grid,
                                  scoring="accuracy")

# initiate measuring execution time
execTime = MeasExecTimeOfProgram()
execTime.start()

searchResults = randomSearch.fit(X_train, y_train)

# Print execution time delta
execTime_sec = execTime.stop()/1000
execTime_str = time.strftime('%H h, %M min, %S sec', time.gmtime(execTime_sec))
#print('Execution time: {:.3f} s'.format(execTime_sec))
print('Execution time: {}'.format(execTime_str))
```

```
Execution time: 00 h, 01 min, 04 sec
```

Extract the best model and evaluate it:

[118]:
```
# predict labels by best model
bestModel = searchResults.best_estimator_

y_pred = bestModel.predict(X_test)
```

[119]:
```
# calculate cross validation score from the best model
# HINT: do NOT use the accuracy score - it's to inaccurate!
accuracies = cross_val_score(estimator = bestModel, X = X_train,
                             y = y_train, cv = 10)

print("Cross-validation score: {:.2f} %".format(accuracies.mean()*100))
print("Standard Deviation: {:.2f} %".format(accuracies.std()*100))
```

```
Cross-validation score: 97.50 %
Standard Deviation: 3.82 %
```

[120]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
                precision    recall  f1-score   support

    Iris-setosa       1.00      1.00      1.00        12
Iris-versicolor       0.90      1.00      0.95         9
 Iris-virginica       1.00      0.89      0.94         9

       accuracy                           0.97        30
      macro avg       0.97      0.96      0.96        30
   weighted avg       0.97      0.97      0.97        30
```

[121]:
```
sns.set_style("white")

# print colored confusion matrix
cm_colored = metrics.ConfusionMatrixDisplay.from_predictions(y_test, y_pred)

cm_colored.figure_.suptitle("Colored Confusion Matrix")
cm_colored.figure_.set_figwidth(8)
cm_colored.figure_.set_figheight(7)

cm_colored.confusion_matrix

plt.tight_layout()
plt.show()
```

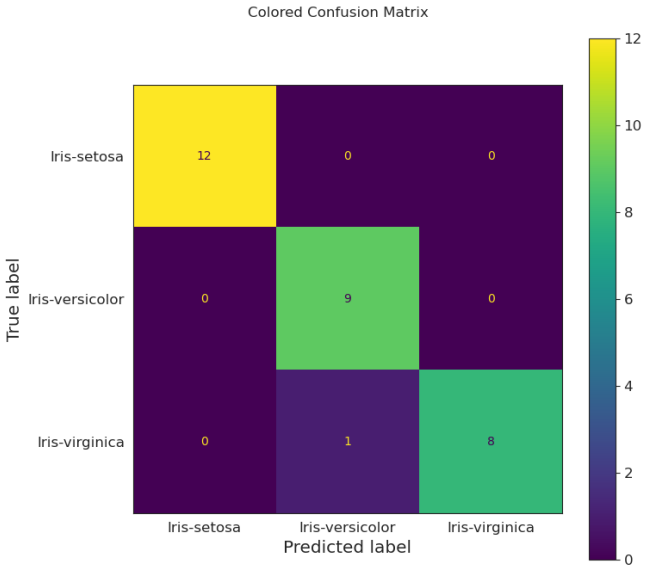


Figure 44: Confusion matrix for cross-validation after the randomized search has been performed

[122]:
```
bestModel.get_params()
```

[122]:
```
{'C': 10,
 'break_ties': False,
 'cache_size': 200,
 'class_weight': None,
```

```
'coef0': 0.0,
'decision_function_shape': 'ovr',
'degree': 1,
'gamma': 1,
'kernel': 'poly',
'max_iter': -1,
'probability': False,
'random_state': None,
'shrinking': True,
'tol': 0.001,
'verbose': False}
```

11 Summary and outlook

11.1 English summary

In November 2022, the **Artificial Intelligence Conference** will be hosted by the German Social Accident Insurance (DGUV) in Dresden. This tutorial is to be presented to interested ML novices in the technical occupational safety and health of the social accident insurance institutions as part of a separate **Getting Started Workshop**.

In the **tutorial**, the **typical workflow** in **machine learning (ML)** was demonstrated systematically and step-by-step using the very familiar **Iris dataset**. The reasons for choosing a ready-made dataset are that an ML novice could first become familiar with ML algorithms, data analysis tools and software libraries as well as programming systems. The task was to distinguish, i.e. classify, three different Iris species based on the **dimensions** (width and length) of their **petals and sepals**. The dataset contains **50 measured individuals per species**.

For the classification of the dataset the very powerful **Support Vector Classifier (SVC)** was used. Although there is a very rich selection of other powerful ML algorithms suitable for the classification task at hand here, the SVC algorithm was deliberately chosen for the target group of the workshop for a comprehensible introduction. Its working principle is easily understandable for ML newcomers as well as in the time frame given for the workshop.

The main sections of the tutorial represent the individual steps in a typical ML workflow. In **Step 0**, specific guidance was given on the selection of hardware and software suitable for machine learning. After introductory notes regarding **community support** to be considered, a distinction was made between training and application systems in the explanations of **hardware selection**. Regarding the **software selection**, the programming languages and programming environments (so-called IDEs) popular with ML developers were introduced and advantages and disadvantages were mentioned in each case. In addition, cloud-hosted IDEs were shown, which eliminate the need to purchase suitable hardware and install the necessary software packages to a certain extent.

As stated above, in **Step 1** the ready-made and very beginner-friendly **Iris dataset** was imported into Python, citing the sources of acquisition. One of the most important and extensive steps in the entire ML process was **Step 2**. In this, the dataset imported in Step 1 was explored using typical data analysis tools. In addition to **exploring** the **data structure** as well as the **inner correlations** in the dataset, **errors** such as gaps, duplications, or obvious misentries had to be found and corrected if possible. This was enormously important so that the classification could later provide plausible results. Since the Iris dataset had no gaps or duplications, another alternative dataset was used to demonstrate the tools.

In **Step 3**, a very brief introduction to the world of artificial intelligence and machine learning was given. The introduction was supported by a **taxonomy of different types of learning** and the listing of selected ML algorithms. A **decision graph** was used to justify the choice of the Support Vector Classifier (SVC) for the classification task at hand. Afterwards, the basic working principle of the SVC including the so-called **kernel trick** was explained. Finally, a corresponding SVC model was implemented.

In **Step 4** the dataset was preprocessed for the actual classification by SVC. Depending on the selected ML algorithm as well as the data structure, it could be necessary to prepare the data before training,

e.g., by **standardization** or **normalization**. For the Iris dataset used, standardization was sufficient to align the value ranges of the features.

After splitting the dataset into a **training and test dataset**, the SVC model was trained with the training dataset in **step 5**. Subsequently, **classification predictions** were made with the trained SVC model using the test data.

In **Step 6**, the quality of the classification result was evaluated using known **metrics** such as the **cross validation score** and the **confusion matrix**.

The classification in step 5 was initially carried out with default values for the so-called **hyper parameters** of the SVC. Therefore, in **step 7** the meaning of the different parameters was explained. Then, their influence on the classification result was demonstrated by **manually varying** the individual hyper-parameters.

In the final **Step 8**, two approaches to systematic hyper-parameter search were presented: **grid search** and **randomized search**. While the former considered exhaustively all parameter combinations for given values, the latter approach selected a number of candidates from a parameter space with a particular random distribution. In order to be able to evaluate whether the two search methods led to improved results, classification was first carried out using default values for the hyper-parameters. The so-called **baseline** was determined. It turned out that the Iris dataset already gave much too good results at default values (recognition rates around 100%) to expect significant improvements by systematic parameter search. Therefore, **Gaussian noise** was added to the dataset in a second step, which allowed to simulate **measurement noise** of real ML applications at the same time.

Using a custom Python class, the **run times** for the two search methods were determined. As expected, the grid search took by far the longest time compared to the randomized search. However, only local optima for the hyper-parameters were always identified for the randomized search, which also led to different results for each run. The quality of the classification in terms of recognition rate was again determined using the metrics presented in step 6. While the grid search always produced better results compared to the baseline, these varied for the randomized search and were each slightly better or worse than those of the baseline.

As an outlook, the current tutorial could be extended by replacing the Iris dataset with the much more modern **Penguin dataset** (original package for **R**: Horst, Hill, and Gorman 2020 as well as adapted package for **Python**: Nakhaee 2021). However, this would mean a fundamental reworking of all the steps in this tutorial.

As explained above, the tutorial so far is limited to the introduction of the ML algorithms, the tools for data analysis, and the Python libraries and programming systems. In a further step, concrete hints could be given on how to build an own **real ML application** and how to generate a suitable dataset for classification.

11.2 German summary

Im November 2022 wird die **Fachtagung "Künstliche Intelligenz"** durch die Deutsche Gesetzliche Unfallversicherung (DGUV) in Dresden ausgerichtet. Dabei soll das vorliegende Tutorial interessierten ML-Neulingen im technischen Arbeitsschutz der gesetzlichen Unfallversicherungsträger im Rahmen eines eigenen **Getting-Started-Workshop** präsentiert werden.

Im **Tutorial** wurde systematisch und Schritt-für-Schritt der **typische Arbeitsablauf** beim **maschinellen Lernen (ML)** anhand des sehr bekannten **Iris-Datensatzes** demonstriert. Es wurde deshalb auf einen fertigen Datensatz zurückgegriffen, damit sich ein ML-Neuling zunächst mit den ML-Algorithmen, den Werkzeugen zur Datenanalyse sowie den Software-Bibliotheken und Programmiersystemen vertraut machen kann. Die Aufgabe bestand darin, drei verschiedene Iris-Arten anhand der **Abmessungen** (Breite und Länge) ihrer **Kron- und Kelchblätter** zu unterscheiden, d. h. zu klassifizieren. Der Datensatz enthält jeweils **50 vermessene Individuen pro Art**.

Zur Klassifikation des Datensatzes kam der sehr leistungsfähige **Support Vector Classifier (SVC)** zum Einsatz. Obwohl es eine sehr reichhaltige Auswahl anderer leistungsfähiger und für die hier vorliegende Klassifikationsaufgabe geeigneter ML-Algorithmen gibt, wurde für einen verständlichen Einstieg bewusst

der SVC-Algorithmus für die Zielgruppe des Workshops gewählt. Dessen Arbeitsweise ist sowohl für ML-Neulinge als auch in dem für den Workshop vorgegebenen Zeitrahmen leicht vermittelbar.

Die Hauptabschnitte des Tutorials repräsentieren die einzelnen Schritte in einem typischen ML-Arbeitsablauf. Im **Schritt 0** wurden konkrete Hinweise für die Auswahl der für das maschinelle Lernen geeigneten Hardware und Software gegeben. Nach einführenden Hinweisen hinsichtlich des zu berücksichtigenden **Community Supports** wurde bei den Ausführungen zur **Hardware-Auswahl** zwischen Trainings- und Applikationssystem unterschieden. Hinsichtlich der **Software-Auswahl** wurden die bei ML-Entwicklern beliebten Programmiersprachen und Programmierumgebungen (sog. IDEs) vorgestellt und jeweils Vor- und Nachteile genannt. Darüber hinaus wurden Cloud-gehostete IDEs gezeigt, die die Anschaffung geeigneter Hardware und die Installation der notwendigen Software-Pakete bis zu einem gewissen Grad überflüssig machen.

Wie oben begründet, wurde im **Schritt 1** der fertige und sehr einsteigerfreundliche **Iris-Datensatz** in Python importiert unter Nennung der Bezugsquellen. Mit der wichtigste und umfangreichste Schritt im gesamten ML-Prozess ist **Schritt 2**. In diesem wurde der in Schritt 1 importierte Datensatz mit Hilfe typischer Datenanalyse-Werkzeuge untersucht. Neben der **Erkundung** der **Datenstruktur** sowie der **inneren Zusammenhänge (sog. Korrelationen)** im Datensatz mussten auch **Fehler** wie z. B. Lücken, Dopplungen oder offensichtliche Fehleingaben gefunden und nach Möglichkeit behoben werden. Dies war enorm wichtig, damit die Klassifikation später plausible Ergebnisse liefern konnte. Da der Iris-Datensatz keine Lücken und Dopplungen aufwies, wurde zur Demonstration der Werkzeuge auf einen anderen alternativen Datensatz ausgewichen.

Im **Schritt 3** wurde zunächst eine sehr kurze Einführung in die Welt der künstlichen Intelligenz und des maschinellen Lernens gegeben. Unterstützt wurde die Einführung durch eine **Taxonomie der verschiedenen Lernarten** und der Nennung ausgewählter ML-Algorithmen. Anhand eines **Entscheidungsgraphes** wurde die Wahl des Support Vector Classifiers (SVC) für die vorliegende Klassifikationsaufgabe begründet. Danach wurde das grundsätzliche Funktionsprinzip des SVC einschließlich des sog. **Kernel-Tricks** erläutert. Abschließend wurde ein entsprechendes SVC-Modell implementiert.

Im **Schritt 4** wurde der Datensatz für die eigentliche Klassifikation per SVC vorbereitet. Je nach gewähltem ML-Algorithmus sowie der Datenstruktur konnte es erforderlich sein, dass die Daten vor dem Training aufbereitet werden mussten, z. B. durch **Standardisierung** oder **Normalisierung**. Für den verwendeten Iris-Datensatz genügte eine Standardisierung, um die Wertebereiche der Features aneinander anzugleichen.

Nach der Aufteilung des Datensatzes in einen **Trainings- und Testdatensatz**, wurde das SVC-Modell im **Schritt 5** mit dem Trainingsdatensatz trainiert. Anschließend wurden mit dem trainierten SVC-Modell anhand der Testdaten **Klassifikationsvorhersagen** getroffen.

Im **Schritt 6** wurde die Güte des Klassifikationsergebnisses unter Verwendung bekannter **Metriken** wie z. B. dem **Kreuzvalidierungskennwert (eng. Cross Validation Score)** und der **Konfusionsmatrix** evaluiert.

Die Klassifikation im Schritt 5 wurde zunächst mit Standardwerten für die sogenannten **Hyper-Parameter** des SVC durchgeführt. Daher wurde im **Schritt 7** die Bedeutung der verschiedenen Parameter erklärt. Danach wurde ihr Einfluss auf das Klassifikationsergebnis durch **manuelle Variation** der einzelnen Hyper-Parameter demonstriert.

Im abschließenden **Schritt 8** wurden zwei Ansätze zur systematischen Hyper-Parameter-Suche vorgestellt: **Rastersuche (eng. Grid Search)** und **Zufallssuche (eng. Randomized Search)**. Während bei ersterer für gegebene Werte alle Parameterkombinationen erschöpfend betrachtet wurden, wurde beim zweiten Ansatz eine Anzahl von Kandidaten aus einem Parameterraum mit einer bestimmten zufälligen Verteilung ausgewählt. Um bewerten zu können, ob die beiden Suchmethoden zu verbesserten Ergebnissen führten, wurde zuerst mit Standardwerten für die Hyper-Parameter klassifiziert. Es wurde hierbei die sogenannte **Basislinie** ermittelt. Dabei zeigte sich, dass der Iris-Datensatz schon bei Standardwerten viel zu gute Ergebnisse lieferte (Erkennungsraten um 100%), um durch systematische Parameter-Suche deutliche Verbesserungen zu erwarten. Daher wurde dem Datensatz in einem zweiten Schritt **Gaußsches Rauschen** hinzugefügt, wodurch gleichzeitig **Messrauschen** realer ML-Anwendungen simuliert werden konnte.

Unter Verwendung einer eigenen Python-Klasse wurden die **Durchlaufzeiten** für die beiden Suchmethoden ermittelt. Erwartungsgemäß dauerte die Rastersuche mit deutlichem Abstand am längsten gegenüber

der Zufallssuche. Allerdings wurden bei der Zufallssuche stets nur lokale Optima für die Hyper-Parameter ermittelt, die auch bei jedem Durchlauf zu anderen Ergebnissen führten. Die Güte der Klassifikation hinsichtlich der Erkennungsrate wurde wieder mit Hilfe der in Schritt 6 vorgestellten Metriken ermittelt. Während die Rastersuche stets bessere Ergebnisse im Vergleich zur Basislinie lieferte, schwankten diese bei der Zufallssuche und waren jeweils geringfügig besser oder schlechter als die der Basislinie.

Ausblickend könnte das vorliegende Tutorial erweitert werden, indem der Iris-Datensatz gegen den deutlich moderneren **Pinguin-Datensatz** ausgetauscht wird (originales Paket für **R**: Horst, Hill, and Gorman 2020 sowie adaptiertes Paket für **Python**: Nakhaee 2021). Dies würde allerdings eine grundsätzliche Überarbeitung sämtlicher Schritte des Tutorials bedeuten.

Wie oben erläutert, beschränkt sich das Tutorial bisher auf die Vorstellung der ML-Algorithmen, die Werkzeuge zur Datenanalyse sowie die Python-Bibliotheken und Programmiersysteme. In einem weiteren Schritt könnten konkrete Hinweise gegeben werden, wie eine eigene **reale ML-Applikation** aufgebaut und ein geeigneter Datensatz für eine Klassifikation erzeugt werden kann.

12 Acknowledgments

Vor einem reichlichen Jahr wurde ich eingeladen, im Vorbereitungskomitee für die DGUV-Fachtagung "Künstliche Intelligenz" mitwirken zu dürfen. Mein Vorschlag, einen eigenen Getting-Started-Workshop für interessierte ML-Neulinge zu gestalten, wurde dort sehr positiv aufgenommen. Das hat mich für die Ausarbeitung des vorliegenden Tutorials sehr motiviert.

Mein besonderer Dank gilt Herrn Prof. André Steimers, der mit langen und sehr interessanten Fachgesprächen, dem Lesen von Rohfassungen und seiner konstruktiven Kritik viel Zeit investierte.

Weiterhin danke ich meinen Kollegen des Dresdener Prüflabors dafür, dass sie sich jederzeit trotz sehr hohem Prüfaufkommen Zeit für meine themenbezogene Fachsimpelei genommen haben. Insbesondere konnte ich während dieser Gespräche meine Gedankengänge und Formulierungen auf Verständlichkeit und Nachvollziehbarkeit prüfen.

Abschließend möchte ich meiner Lebensgefährtin danken, dass sie erste Textentwürfe kritisch Korrektur gelesen hat und mir ansonsten den Rücken freigehalten hat - auch wenn ich nach Feierabend oder an den Wochenenden programmiert und geschrieben habe. Unserem zweijährigen Sohn danke ich für seine Geduld mit Papa. Er hätte sicherlich das ein oder andere Mal lieber "Die Sendung mit der Maus" statt seltsamer Grafiken mit mir auf dem Rechner angeschaut.

Dresden, 25.10.2022

13 References

Online references

Adhikari, Daisy (Mar. 1, 2021). *Is Octave Good for Machine Learning?* English. Data Science Nerd. URL: https://datasciencenerd.com/is-octave-good-for-machine-learning/ (visited on 09/08/2022) (cit. on pp. 10, 12).

Adler, Rasmus (Sept. 6, 2019). *Autonom oder vielleicht doch nur hochautomatisiert? Was ist z.B. der Unterschied zwischen autonomem Fahren und hochautomatisiertem Fahren?* German. Ed. by Fraunhofer IESE. URL: https://www.iese.fraunhofer.de/blog/autonom-oder-vielleicht-doch-nur-hochautomatisiert-was-ist-eigentlich-der-unterschied/ (visited on 09/06/2022) (cit. on pp. 3, 6).

– (June 10, 2021). *Autonome Systeme: großes Potenzial für die digitale Zukunft.* German. Ed. by Fraunhofer IESE. URL: https://www.iese.fraunhofer.de/blog/autonome-systeme/ (visited on 09/06/2022) (cit. on pp. 3, 6).

Allan, Alasdair (June 24, 2019). *Benchmarking Machine Learning on the New Raspberry Pi 4, Model B.* English. URL: https://www.hackster.io/news/benchmarking-machine-learning-on-the-new-raspberry-pi-4-model-b-88db9304ce4 (visited on 08/31/2022) (cit. on p. 10).

Allwright, Stephen (June 26, 2022). *Using cross_val_score in sklearn, simply explained*. English. URL: https://stephenallwright.com/cross_val_score-sklearn/ (visited on 10/18/2022) (cit. on p. 80).

Baba, Arshid Aslam (Mar. 22, 2018). *Iris Flower Dataset. Iris flower data set used for multi-class classification*. English. URL: https://www.kaggle.com/datasets/arshid/iris-flower-dataset (visited on 08/31/2022) (cit. on p. 24).

Bhandari, Aniruddha (Apr. 3, 2020). *Feature Scaling for Machine Learning: Understanding the Difference Between Normalization vs. Standardization*. English. URL: https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/ (visited on 10/09/2022) (cit. on pp. 69, 70, 74).

Das, Tamal (Aug. 16, 2022). *Google Colab: Everything you Need to Know*. English. URL: https://geekflare.com/google-colab/ (visited on 09/14/2022) (cit. on p. 20).

Exterman, Dori (June 7, 2021). *CUDA vs OpenCL: Which to Use for GPU Programming*. English. URL: https://www.incredibuild.com/blog/cuda-vs-opencl-which-to-use-for-gpu-programming (visited on 08/31/2022) (cit. on p. 10).

Fraj, Mohtadi Ben (May 5, 2018). *In Depth: Parameter tuning for SVC*. English. URL: https://medium.com/all-things-ai/in-depth-parameter-tuning-for-svc-758215394769 (visited on 09/22/2022) (cit. on p. 66).

Gupta, Sakshi (Oct. 6, 2021). *What is the best language for machine learning?* English. Springboard. URL: https://www.springboard.com/blog/data-science/best-language-for-machine-learning/ (visited on 09/08/2022) (cit. on pp. 10, 13).

Heidrich, Jens, Daniel Schneider, and Andreas Jedlitschka (Mar. 4, 2021). *Dependable AI / Verlässliche KI – Ein Überblick*. German. Ed. by Fraunhofer IESE. URL: https://www.iese.fraunhofer.de/blog/dependable-ai/ (visited on 09/06/2022) (cit. on pp. 4, 7).

Horst, Allison Marie, Alison Presmanes Hill, and Kristen B Gorman (July 16, 2020). *palmerpenguins: Palmer Archipelago (Antarctica) penguin data*. English. R package version 0.1.0. DOI: 10.5281/zenodo.3960218. URL: https://allisonhorst.github.io/palmerpenguins/ (visited on 10/04/2022) (cit. on pp. 104, 106).

IndianTechWarrior, ed. (June 7, 2022). *Does TensorFlow Support OpenCL?* English. IndianTechWarrior. URL: https://indiantechwarrior.com/does-tensorflow-support-opencl/ (visited on 08/31/2022) (cit. on p. 10).

Iwanski, Luke (June 1, 2016). *TensorFlow for OpenCL using SYCL*. English. Codeplay Software Ltd. URL: https://www.codeplay.com/portal/blogs/2016/06/01/tensorflow-for-opencl-using-sycl.html (visited on 08/31/2022) (cit. on p. 10).

Johnson, Leigh (June 24, 2019). *Portable Computer Vision: TensorFlow 2.0 on a Raspberry Pi*. English. URL: https://towardsdatascience.com/portable-computer-vision-tensorflow-2-0-on-a-raspberry-pi-part-1-of-2-84e318798ce9 (visited on 08/31/2022) (cit. on p. 10).

Lell, Daud (Apr. 10, 2019). *Einführung in Machine Learning mit Python – Support Vector Machines*. English. URL: https://www.ancud.de/support-vector-machine-svn/ (visited on 09/22/2022) (cit. on pp. 65, 66).

Misal, Disha (July 30, 2021). *5 Alternatives To Google Colab For Data Scientists*. English. URL: https://analyticsindiamag.com/5-alternatives-to-google-colab-for-data-scientists/ (visited on 09/14/2022) (cit. on p. 20).

Nakhaee, Muhammad Chenariyan (Dec. 17, 2021). *Palmerpenguins as Python package*. English. URL: https://github.com/mcnakhaee/palmerpenguins (visited on 10/04/2022) (cit. on pp. 104, 106).

NVIDIA.Developer (July 20, 2022). *CUDA GPUs - Compute Capability*. English. Ed. by NVIDIA Corporation. URL: https://developer.nvidia.com/cuda-gpus (visited on 08/31/2022) (cit. on p. 9).

Politiek, Rients (Aug. 16, 2022a). *Deep learning with Raspberry Pi and alternatives in 2022*. English. Q-engineering. URL: https://qengineering.eu/deep-learning-with-raspberry-pi-and-alternatives.html (visited on 08/31/2022) (cit. on p. 10).

– (June 10, 2022b). *Install OpenCL on Raspberry Pi 3 B+*. English. Q-engineering. URL: https://qengineering.eu/install-opencl-on-raspberry-pi-3.html (visited on 08/31/2022) (cit. on p. 10).

Rapp, Rachel (Mar. 18, 2021). *Best Google Colab Alternatives in 2021*. English. URL: https://blog.paperspace.com/best-google-colab-alternatives/ (visited on 09/14/2022) (cit. on pp. 19, 20).

Romer, Paul (Apr. 13, 2018). *Jupyter, Mathematica, and the Future of the Research Paper*. English. URL: https://paulromer.net/jupyter-mathematica-and-the-future-of-the-research-paper/ (visited on 09/08/2022) (cit. on p. 16).

Siow, Eugene (Dec. 17, 2020). *Free GPUs for Machine Learning: Google Colab vs Paperspace Gradient*. English. URL: https://news.machinelearning.sg/posts/google_colab_vs_paperspace_gradient/ (visited on 09/14/2022) (cit. on p. 21).

Somers, James (Apr. 5, 2018). *The Scientific Paper Is Obsolete*. English. The Atlantic. URL: https://www.theatlantic.com/science/archive/2018/04/the-scientific-paper-is-obsolete/556676/ (visited on 09/08/2022) (cit. on p. 16).

Wikipedia: Autonomie (June 4, 2022). *Autonomie*. German. Ed. by Wikipedia, The Free Encyclopedia. URL: https://de.wikipedia.org/w/index.php?title=Autonomie&oldid=224750608 (visited on 09/06/2022) (cit. on pp. 4, 6).

Wikipedia: CUDA (Aug. 28, 2022). *CUDA*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=CUDA&oldid=1107147697 (visited on 08/30/2022) (cit. on p. 9).

Wikipedia: IDE (Sept. 1, 2022). *Integrated development environment*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Integrated_development_environment&oldid=1107871547 (visited on 09/08/2022) (cit. on p. 15).

Wikipedia: Iris dataset (Aug. 8, 2022). *Iris flower data set*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Iris_flower_data_set&oldid=1103056245 (visited on 08/31/2022) (cit. on p. 24).

Wikipedia: Iris setosa (Aug. 5, 2022). *Iris setosa*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Iris_setosa&oldid=1102525458 (visited on 09/01/2022) (cit. on p. 25).

Wikipedia: Iris versicolor (June 4, 2022). *Iris versicolor*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Iris_versicolor&oldid=1091474628 (visited on 09/01/2022) (cit. on p. 25).

Wikipedia: Iris virginica (June 21, 2022). *Iris virginica*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Iris_virginica&oldid=1094181148 (visited on 09/01/2022) (cit. on p. 25).

Wikipedia: OpenCL (Aug. 21, 2022). *OpenCL*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=OpenCL&oldid=1105615914 (visited on 08/30/2022) (cit. on p. 9).

Wikipedia: OSAL (Apr. 28, 2022). *OSAL*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Operating_system_abstraction_layer&oldid=1085096210 (visited on 09/01/2022) (cit. on p. 9).

Wikipedia: Petal (May 15, 2022). *Petal*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Petal&oldid=1087986268 (visited on 09/01/2022) (cit. on p. 25).

Wikipedia: Raspi (Aug. 28, 2022). *Raspberry Pi*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Raspberry_Pi&oldid=1107184087 (visited on 08/31/2022) (cit. on p. 10).

Wikipedia: Reinforcement learning (Sept. 16, 2022). *Reinforcement learning*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Reinforcement_learning&oldid=1110645217 (visited on 09/22/2022) (cit. on p. 64).

Wikipedia: Semi-supervised learning (Sept. 20, 2022). *Semi-supervised learning*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Semi-supervised_learning&oldid=1111251364 (visited on 09/22/2022) (cit. on p. 64).

Wikipedia: Sepal (May 23, 2022). *Sepal*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Sepal&oldid=1089457643 (visited on 09/01/2022) (cit. on p. 25).

Wikipedia: Supervised learning (Aug. 31, 2022). *Supervised learning*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Supervised_learning&oldid=1107692217 (visited on 09/22/2022) (cit. on p. 63).

Wikipedia: Support-vector machine (Sept. 1, 2022). *Support-vector machine*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Support-vector_machine&oldid=1107938510 (visited on 09/22/2022) (cit. on p. 65).

Wikipedia: SYCL (Aug. 1, 2022). *SYCL*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=SYCL&oldid=1101810571 (visited on 08/31/2022) (cit. on p. 10).

Wikipedia: Unsupervised learning (Aug. 28, 2022). *Unsupervised learning*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Unsupervised_learning&oldid=1107188105 (visited on 09/22/2022) (cit. on p. 64).

Wikipedia: VM (Aug. 17, 2022). *Virtual machine*. English. Ed. by Wikipedia, The Free Encyclopedia. URL: https://en.wikipedia.org/w/index.php?title=Virtual_machine&oldid=1104946655 (visited on 08/31/2022) (cit. on p. 10).

Wilimitis, Drew (Dec. 12, 2018). *The Kernel Trick in Support Vector Classification*. English. URL: https://towardsdatascience.com/the-kernel-trick-c98cdbcaeb3f (visited on 09/23/2022) (cit. on pp. 66, 67).

Zhang, Grace (Nov. 11, 2018). *What is the kernel trick? Why is it important?* English. URL: https://medium.com/@zxr.nju/what-is-the-kernel-trick-why-is-it-important-98a98db0961d (visited on 09/23/2022) (cit. on p. 67).

Books, technical reports and others

DIN EN 61508-3 (Feb. 2011). *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme – Teil 3: Anforderungen an Software (IEC 61508-3:2010); Deutsche Fassung EN 61508-3:2010*. deutsch. Tech. rep. DKE Deutsche Kommission Elektrotechnik Elektronik Informationstechnik im DIN und VDE. URL: https://www.beuth.de/de/norm/din-en-61508-3/135505701 (visited on 09/06/2022) (cit. on pp. 4, 7).

DIN EN ISO 13849-1 (June 2016). *Sicherheit von Maschinen – Sicherheitsbezogene Teile von Steuerungen – Teil 1: Allgemeine Gestaltungsleitsätze (ISO 13849-1:2015); Deutsche Fassung EN ISO 13849-1:2015*. deutsch. Tech. rep. DIN Deutsches Institut für Normung e. V. URL: http://www.beuth.de/de/norm/din-en-iso-13849-1/230387878 (visited on 09/06/2022) (cit. on pp. 4, 7).

Dumitrescu, Roman et al. (Jan. 28, 2018). *Studie "Autonome Systeme"*. Tech. rep. Expertenkommission Forschung und Innovation (EFI), Berlin. URL: https://www.econstor.eu/handle/10419/175555 (visited on 09/06/2022) (cit. on pp. 4, 6).

Fisher, R. A. (Sept. 1936). "The use of multiple measurements in taxonomic problems". In: vol. 7. 2, pp. 179–188. DOI: https://doi.org/10.1111/j.1469-1809.1936.tb02137.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1469-1809.1936.tb02137.x (visited on 08/31/2022) (cit. on p. 24).

Géron, Aurélien (2018). *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow. Konzepte, Tools und Techniken für intelligente Systeme*. German. Trans. English by Kristian Rother. 1. Auflage. Heidelberg: O'Reilly, p. 552. ISBN: 978-3-96010-114-7. URL: https://katalog.ub.uni-heidelberg.de/cgi-bin/titel.cgi?katkey=68275614 (visited on 08/30/2022) (cit. on pp. 69, 80).

Gonzalez Viejo, Claudia et al. (Nov. 1, 2019). "Emerging technologies based on artificial intelligence to assess quality and consumer preference of beverages". In: *Beverages* 5. DOI: 10.3390/beverages5040062. URL: https://www.researchgate.net/publication/336375517_Emerging_technologies_based_on_artificial_intelligence_to_assess_quality_and_consumer_preference_of_beverages (visited on 09/21/2022) (cit. on pp. 62, 63).

Hunter, J. D. (June 18, 2007). "Matplotlib: A 2D graphics environment". In: *Computing in Science & Engineering* 9.3, pp. 90–95. DOI: 10.1109/MCSE.2007.55. URL: https://matplotlib.org/stable/users/project/history.html (visited on 09/09/2022) (cit. on p. 14).

ISO/IEC 22989 (July 2022). *Information technology - Artificial intelligence - Artificial intelligence concepts and terminology*. Tech. rep. ISO/IEC JTC 1/SC 42 - Artificial Intelligence. URL: https://webstore.iec.ch/publication/77839 (visited on 09/21/2022) (cit. on p. 62).

Jürgensohn, Thomas et al. (Apr. 30, 2021). *Rechtliche Rahmenbedingungen für die Bereitstellung autonomer und KI-Systeme*. German. Tech. rep. BAuA, Bundesanstalt für Arbeitsschutz und Arbeitsmedizin. DOI: doi:10.21934/baua:bericht20210423. URL: http://www.baua.de/dok/8859246 (visited on 09/06/2022) (cit. on pp. 4, 5, 7).

Kadner, Susanne et al. (Mar. 20, 2017). *Fachforum autonome Systeme – Chancen und Risiken für Wirtschaft, Wissenschaft und Gesellschaft*. Tech. rep. acatech – Deutsche Akademie der Technikwissenschaften. URL: https://www.acatech.de/publikation/fachforum-autonome-systeme-chancen-und-risiken-fuer-wirtschaft-wissenschaft-und-gesellschaft-abschlussbericht/ (visited on 09/06/2022) (cit. on pp. 4, 6).

Karimi, Kamran, Neil G. Dickson, and Firas Hamze (May 16, 2011). "A Performance Comparison of CUDA and OpenCL". In: DOI: https://doi.org/10.48550/arXiv.1005.2581. URL: https://arxiv.org/abs/1005.2581 (visited on 08/31/2022) (cit. on p. 10).

Kuhn, Thomas and Peter Liggesmeyer (Jan. 30, 2019). "Autonome Systeme – Potenziale und Herausforderungen: Perspektiven, Herausforderungen und Grenzen der Künstlichen Intelligenz in der Arbeitswelt". In: *Autonome Systeme und Arbeit*. transcript Verlag, pp. 27–46. ISBN: 978-3-8376-4395-4. DOI: 10.14361/9783839443958-003. URL: https://www.researchgate.net/publication/338265062_Autonome_Systeme_Potenziale_und_Herausforderungen_Perspektiven_Herausforderungen_und_Grenzen_der_Kunstlichen_Intelligenz_in_der_Arbeitswelt (visited on 09/06/2022) (cit. on pp. 4, 6).

Raschka, Sebastian and Vahid Mirjalili (2018). *Machine Learning mit Python und Scikit-learn und TensorFlow. Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics*. German. Trans. English by Knut Lorenzen. 2nd ed. mitp Verlags GmbH & Co. KG, Frechen, p. 577. ISBN: 978-3-95845-733-1. URL: https://katalog.ub.uni-heidelberg.de/cgi-bin/titel.cgi?katkey=68206280 (visited on 08/30/2022) (cit. on p. 40).

Waskom, Michael L. (Apr. 6, 2021). "seaborn: statistical data visualization". In: *Journal of Open Source Software* 6.60, p. 3021. DOI: 10.21105/joss.03021. URL: https://joss.theoj.org/papers/10.21105/joss.03021 (visited on 09/09/2022) (cit. on p. 14).