INTRODUCTION TO MACHINE LEARNING

Dr. Kamakshaiah Musunuru

About the Author

Dr. M. Kamakshaiah is a open source software evangelist, enterprise solutions architect and academic of data science and analytics. His teaching interests are IT practices in business, MIS, Data Science, Business Analytics including functional analytics related to marketing, finance, HRM, quality and operations. He also teaches theoretical concepts like multivariate analytics, numerical simulations & optimization, machine learning & AI using few programming languages like R, Python, Java. Internet of Things and big data analytics (Hadoop) are his all time favorites for teaching.

He has taught abroad for two years and credits two country visits. He has developed few free and open source software solutions meant for corporate practitioners, academics, scholars and students engaging in data science and analytics. All his software applications are available from his Github portal https://github.com/Kamakshaiah.

Foreword

I started writing this book for one simple reason that I got to teach machine learning for business analytics students. Actually I started preserving every little code, chunk by chunk, as and when I struck with this subject. Though I thought it as notes at first but the stuff that I gathered became rather more significant to compile it as a book.

I turned into data analyst due to my being accidentally exposed to open source software when I was in Ethiopia. I got to use GNU/Linux to keep my personal computer away from virus infection at the university. I crashed Windows a couple of times when I was working in office at the University, eventually I found GNU/Linux as a wonderful solution to address virus issues. My first experience of GNU/Linux, perhaps, is *Jaunty Jackalope*, as I suppose. This must be Ubuntu 9.04, an LTS version, if I am not wrong. However, my involvement in GNU/Linux became a serious affair though *Karmic Koala*. [1] I still remember few of my colleagues used to visit my home for OS installation in those days of stay at Ethiopia.

My first encounter with analytics software is R. R is *lingua franca* of statistics. I taught "business statistics" several times during

[1] *Karmic Koala* is Ubuntu 9.10. Visit https://wiki.ubuntu.com/Releases for more information on Ubuntu releases.

my stint as academic. Once, I notice this little yet uppercase R letter over there while I was browsing for statistical software in Ubuntu *package manager*. R changed not only my understanding of Statistics but the very way of learning the same. I addicted to R so much so that for every pretty little calculations I used to refer R manuals. I must be the first academic to introduce R in south Indian business management curriculum, yet remained unknown to others. Today, R is one of the best tools for statistical analysis and practice of quantitative techniques.

I came across Python just as in same way as I discovered R in Ubuntu. Python was one of the default programming languages for many things in Ubuntu. At first I did not know the power of Python for everything I learned it was only by self study. All my learning is from online resources. However, I could produce thousands of students majored in data analysts through my formal classes. All that I learned and taught is only by my self study, I mean, through very informal personal learning.

I am writing all this not to show myself as a valiant learner, but to demonstrate how can a novice and a naive enthusiast like me can learn programming tools like R and Python. Today, I am offering a couple of courses to teach Hadoop and IoT, that is all by passion. So, I would like to give confidence to the reader that you don't need any formal learning in computer science or IT to learn data science and adopt it as a profession. However, you need tons and tons of patience and passion.

This book is meant for beginners of machine learning and practice. It has 5 chapters each section represents a unique concept of data analytics. This book may be useful for both practitioners and academics to acquire knowledge of machine learning accompanied with Python practice. The first chapter *introduction*, deals with very short information related to basics of machine learning. Chapter I has information related to installation of Python in Linux, Windows and few other OSes. Chapter 2 deals with *Supervised learning* and this chapter has information re-

lated to various supervised machine learning algorithms such as ... Chapter 3, *Unsuptervised learning* deals with algorithms such as ... Chapter 4, *Deep lerning*, deals with different types of techniques related to ...

As far as coding is concerned; those code sections where there exist left-bar such as the below

```
statement 1
statement 2
statement 3
```

represents a ***script***. A script is a plain code file in which there exists program statements. There are other code sections where each statement is preceded by python prompt (>>>). These code sections are meant for testing. These sections are useful either to evaluate a Python statement or provide evidence for logic.

All the code chunks used in this book are provided though my github portal with a project name *PfDSaA*. Feel free to visit the site https://github.com/Kamakshaiah/PfDSaA and download required *.py* files for practice.

One last note is that this book is meant for both data scientists and analysts. The learner need to have basics of Python. However, little logical thinking together with passion and patience are required. The book is written in such a way that even a person having no knowledge on Python can pick up and become maverick at the end of reading. This book covers from very basic details ranging from installation to creating packages. I am not covering very advanced concepts such as software and applications development due to one reason to keep this book reasonable for both beginners and advanced users.

Happy reading $\cdots$

Author
Dr. M. Kamakshaiah

Contents

Chapter 1

Introduction

Machine learning (ML) is a field of inquiry devoted to understanding and building methods that "learn", that is, methods that leverage data to improve performance on some set of tasks. [1] It is seen as a part of artificial intelligence. Machine learning algorithms build a model based on sample data, known as training data, in order to make predictions or decisions without being explicitly programmed to do so. Machine learning algorithms are used in a wide variety of applications, such as in medicine, email filtering, speech recognition, and computer vision, where it is difficult or unfeasible to develop conventional algorithms to perform the needed tasks.

A subset of machine learning is closely related to computational statistics, which focuses on making predictions using computers, but not all machine learning is statistical learning. The study of mathematical optimization delivers methods, theory and application domains to the field of machine learning. Data mining is a related field of study, focusing on exploratory data analysis through unsupervised learning. [2] Some implementations of machine learning use data and neural networks in a way that

mimics the working of a biological brain. [3] In its application across business problems, machine learning is also referred to as predictive analytics.

Machine learning programs can perform tasks without being explicitly programmed to do so. It involves computers learning from data provided so that they carry out certain tasks. For simple tasks assigned to computers, it is possible to program algorithms telling the machine how to execute all steps required to solve the problem at hand; on the computer's part, no learning is needed. For more advanced tasks, it can be challenging for a human to manually create the needed algorithms. In practice, it can turn out to be more effective to help the machine develop its own algorithm, rather than having human programmers specify every needed step. [4]

The discipline of machine learning employs various approaches to teach computers to accomplish tasks where no fully satisfactory algorithm is available. In cases where vast numbers of potential answers exist, one approach is to label some of the correct answers as valid. This can then be used as training data for the computer to improve the algorithm(s) it uses to determine correct answers. For example, to train a system for the task of digital character recognition, the MNIST dataset of handwritten digits has often been used.

1.1 Machine

The word "machine" is very critical because machine learning for machine is the main entity that process data and accomplish most of the tasks required for learning. In the most general sense, a machine can be defined as below.

> "A machine is an apparatus using mechanical power and having several parts, each with a definite function and together performing a particular task."

Although, the word *machine* is used in generic parlance, all over this text, but machine learning is beign done through a machine called computer. In fact, machine learning is a field of science that deals with computer algorithms and using those algorithms for training and testing data for consistent results related to prediction. In the domain of computer science, there are two types of machine that are vogue in usage. The first is the machine defined by *von Neumann* architecture. The second, *Turing's B-Type* machines.

1.1.1 *von Neumann* architecture

The von Neumann architecture, also known as the von Neumann model or Princeton architecture, is a computer architecture based on a 1945 description by John von Neumann and others in the First Draft of a Report on the EDVAC. [5] [1] That document describes a design architecture for an electronic digital computer with these components:

- A processing unit that contains an arithmetic logic unit and processor registers
- A control unit that contains an instruction register and program counter
- Memory that stores data and instructions
- External mass storage
- Input and output mechanisms

The term "von Neumann architecture" has evolved to mean any stored-program computer in which an instruction fetch and a data operation cannot occur at the same time because they share a common bus. This is referred to as the *von Neumann bottleneck* and often limits the performance of the system.

[1] EDVAC stands for *Electronic Discrete Variable Automatic Computer*

1.1.2 Turing's B-type machines

Also known as *unorganized machine* is a concept mentioned in a 1948 report in which Alan Turing suggested that the infant human cortex was what he called an "unorganized machine". [6]

Turing defined the class of unorganized machines as largely random in their initial construction, but capable of being trained to perform particular tasks. Turing's unorganized machines were in fact very early examples of randomly connected, binary neural networks, and Turing claimed that these were the simplest possible model of the nervous system.

In his 1948 paper Turing defined two examples of his unorganized machines. The first were A-type machines — these being essentially randomly connected networks of NAND logic gates. The second were called B-type machines, which could be created by taking an A-type machine and replacing every inter-node connection with a structure called a connection modifier — which itself is made from A-type nodes. The purpose of the connection modifiers were to allow the B-type machine to undergo "appropriate interference, mimicking education" in order to organize the behavior of the network to perform useful work. Before the term genetic algorithm was coined, Turing even proposed the use of what he called a genetic search to configure his unorganized machines. Turing claimed that the behavior of B-type machines could be very complex when the number of nodes in the network was large, and stated that the "picture of the cortex as an unorganized machine is very satisfactory from the point of view of evolution and genetics". Turing had been interested in the possibility of simulating neural systems for at least the previous two years. [2]

[2] Please read few of his lines that he shared while in correspondence with William Ross Ashby in 1946 at http://www.rossashby.info/letters/turing.html

1.2 Learning

Learning is the process of acquiring new understanding, knowledge, behaviors, skills, values, attitudes, and preferences. [7] The ability to learn is possessed by humans, animals, and some machines; there is also evidence for some kind of learning in certain plants.[8] Some learning is immediate, induced by a single event (e.g. being burned by a hot stove), but much skill and knowledge accumulate from repeated experiences. The changes induced by learning often last a lifetime, and it is hard to distinguish learned material that seems to be "lost" from that which cannot be retrieved.[9]

1.2.1 Human learning

Human learning starts at birth (it might even start before in terms of an embryo's need for both interaction with, and freedom within its environment within the womb.) and continues until death as a consequence of ongoing interactions between people and their environment. The nature and processes involved in learning are studied in many established fields (including educational psychology, neuropsychology, experimental psychology, cognitive sciences, and pedagogy), as well as emerging fields of knowledge (e.g. with a shared interest in the topic of learning from safety events such as incidents/accidents, or in collaborative learning health systems).[10][11] Research in such fields has led to the identification of various sorts of learning. For example, learning may occur as a result of habituation, or classical conditioning, operant conditioning or as a result of more complex activities such as play, seen only in relatively intelligent animals. Learning may occur consciously or without conscious awareness. Learning that an aversive event can't be avoided or escaped may result in a condition called learned helplessness. There is evidence for human behavioral learning prenatally, in which habituation has been observed as early as 32 weeks into gestation, indicating that the central nervous system is suffi-

ciently developed and primed for learning and memory to occur very early on in development.

Play has been approached by several theorists as a form of learning. Children experiment with the world, learn the rules, and learn to interact through play. Lev Vygotsky agrees that play is pivotal for children's development, since they make meaning of their environment through playing educational games. For Vygotsky, however, play is the first form of learning language and communication, and the stage where a child begins to understand rules and symbols.[12] This has led to a view that learning in organisms is always related to semiosis, and often associated with representational systems/activity.[13]

1.2.2 Machine learning

The term machine learning was coined in 1959 by *Arthur Samuel*, an IBM employee and pioneer in the field of computer gaming and artificial intelligence. [14] [15] Also the synonym self-teaching computers were used in this time period. [16]

By the early 1960s an experimental "learning machine" with punched tape memory, called *Cybertron*, had been developed by *Raytheon Company* to analyze sonar signals, electrocardiograms and speech patterns using rudimentary reinforcement learning. It was repetitively trained by a human operator/teacher to recognize patterns and equipped with a "goof" button to cause it to re-evaluate incorrect decisions. A representative book on research into machine learning during the 1960s was *Nilsson's* book on Learning Machines, dealing mostly with machine learning for pattern classification. Interest related to pattern recognition continued into the 1970s, as described by Duda and Hart in 1973. [17] In 1981 a report was given on using teaching strategies so that a neural network learns to recognize 40 characters (26 letters, 10 digits, and 4 special symbols) from a computer terminal.

Tom M. Mitchell provided a widely quoted, more formal defi-

nition of the algorithms studied in the machine learning field: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P if its performance at tasks in T, as measured by P, improves with experience E." [18] This definition of the tasks in which machine learning is concerned offers a fundamentally operational definition rather than defining the field in cognitive terms. This follows Alan Turing's proposal in his paper "Computing Machinery and Intelligence", in which the question "Can machines think?" is replaced with the question "Can machines do what we (as thinking entities) can do?".

Modern day machine learning has two objectives, one is to classify data based on models which have been developed, the other purpose is to make predictions for future outcomes based on these models. A hypothetical algorithm specific to classifying data may use computer vision of moles coupled with supervised learning in order to train it to classify the cancerous moles. A machine learning algorithm for stock trading may inform the trader of future potential predictions.

1.3 Associated fields

Machine learning greatly a collection or at least intersection of various domains of knowledge. For instance, there are views that machine learning is subset of artificial intelligence.[19] Few see it as integration an intersection of artificial intelligence, data mining and statistics.[20]

1.3.1 Artificial intelligence

As a scientific endeavor, machine learning grew out of the quest for artificial intelligence. In the early days of AI as an academic discipline, some researchers were interested in having machines learn from data. They attempted to approach the problem with various symbolic methods, as well as what was then termed "neu-

ral networks"; these were mostly *perceptrons* and other models that were later found to be reinventions of the generalized linear models of statistics.[21] Probabilistic reasoning was also employed, especially in automated medical diagnosis. [22]

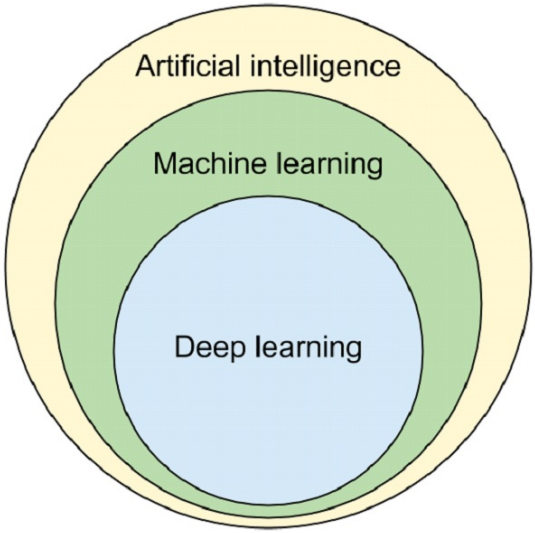

Figure 1.1: AI vs ML

However, an increasing emphasis on the logical, knowledge-based approach caused a rift between AI and machine learning. Probabilistic systems were plagued by theoretical and practical problems of data acquisition and representation.? By 1980, expert systems had come to dominate AI, and statistics was out of favor. [23] Work on symbolic/knowledge-based learning did continue within AI, leading to inductive logic programming, but the more statistical line of research was now outside the field of AI proper, in pattern recognition and information retrieval. Neural networks research had been abandoned by AI and computer science around the same time. This line, too, was contin-

ued outside the AI/CS field, as "connectionism", by researchers from other disciplines including *Hopfield, Rumelhart* and *Hinton*. Their main success came in the mid-1980s with the reinvention of *backpropagation*.

Machine learning (ML), reorganized as a separate field, started to flourish in the 1990s. The field changed its goal from achieving artificial intelligence to tackling solvable problems of a practical nature. It shifted focus away from the symbolic approaches it had inherited from AI, and toward methods and models borrowed from statistics, fuzzy logic, and probability theory.

The difference between ML and AI is frequently misunderstood. ML learns and predicts based on passive observations, whereas AI implies an agent interacting with the environment to learn and take actions that maximize its chance of successfully achieving its goals. [24]

As of 2020, many sources continue to assert that ML remains a subfield of AI. Others have the view that not all ML is part of AI, but only an "intelligent subset" of ML should be considered AI.

1.3.2 Data mining

Machine learning and data mining often employ the same methods and overlap significantly, but while machine learning focuses on prediction, based on known properties learned from the training data, data mining focuses on the discovery of (previously) unknown properties in the data (this is the analysis step of knowledge discovery in databases). Data mining uses many machine learning methods, but with different goals; on the other hand, machine learning also employs data mining methods as "unsupervised learning" or as a *preprocessing* step to improve learner accuracy. Much of the confusion between these two research communities comes from the basic assumptions they work with. *In machine learning, performance is usually evaluated with re-*

spect to the ability to reproduce known knowledge, while in knowledge discovery and data mining (KDD) the key task is the discovery of previously unknown knowledge. Evaluated with respect to known knowledge, an uninformed (unsupervised) method will easily be outperformed by other supervised methods, while in a typical KDD task, supervised methods cannot be used due to the unavailability of training data.

1.3.3 Optimization

Machine learning also has intimate ties to optimization. Many learning problems are formulated as minimization of some loss function on a training set of examples. Loss functions express the discrepancy between the predictions of the model being trained and the actual problem instances. For example, in classification, one wants to assign a label to instances, and models are trained to correctly predict the preassigned labels of a set of examples.

1.3.4 Generalization

The difference between optimization and machine learning arises from the goal of generalization. While optimization algorithms can minimize the loss on a training set, machine learning is concerned with minimizing the loss on unseen samples. Characterizing the generalization of various learning algorithms is an active topic of current research, especially for deep learning algorithms.

1.3.5 Statistics

Machine learning and statistics are closely related fields in terms of methods, but distinct in their principal goal. Statistics draws population inferences from a sample, while machine learning finds generalizable predictive patterns. [25] According to *Michael I. Jordan*, the ideas of machine learning, from methodological principles to theoretical tools, have had a long pre-history in statistics.[26] He also suggested the term data science as a place-

holder to call the overall field. *Leo Breiman* distinguished two statistical modeling paradigms. Data model and algorithmic model, wherein "algorithmic model" means more or less the machine learning algorithms like Random forest. Some statisticians have adopted methods from machine learning, leading to a combined field that they call statistical learning.[27]

1.4 Categories of Machine Learning

Machine learning approaches are traditionally divided into three broad categories, depending on the nature of the "signal" or "feedback" available to the learning system:

1. *Supervised learning*: The computer is presented with example inputs and their desired outputs, given by a "teacher", and the goal is to learn a general rule that maps inputs to outputs.

2. *Unsupervised learning*: No labels are given to the learning algorithm, leaving it on its own to find structure in its input. Unsupervised learning can be a goal in itself (discovering hidden patterns in data) or a means towards an end (feature learning).

3. *Reinforcement learning*: A computer program interacts with a dynamic environment in which it must perform a certain goal (such as driving a vehicle or playing a game against an opponent). As it navigates its problem space, the program is provided feedback that's analogous to rewards, which it tries to maximize.

1.4.1 Supervised learning

Supervised learning algorithms build a mathematical model of a set of data that contains both the inputs and the desired outputs.[28] The data is known as training data, and consists of a set

of training examples. Each training example has one or more inputs and the desired output, also known as a supervisory signal. In the mathematical model, each training example is represented by an array or vector, sometimes called a feature vector, and the training data is represented by a matrix. Through iterative optimization of an objective function, supervised learning algorithms learn a function that can be used to predict the output associated with new inputs.[29] An optimal function will allow the algorithm to correctly determine the output for inputs that were not a part of the training data. An algorithm that improves the accuracy of its outputs or predictions over time is said to have learned to perform that task.[30]

Types of supervised-learning algorithms include active learning, classification and regression.[31] Classification algorithms are used when the outputs are restricted to a limited set of values, and regression algorithms are used when the outputs may have any numerical value within a range. As an example, for a classification algorithm that filters emails, the input would be an incoming email, and the output would be the name of the folder in which to file the email.

Similarity learning is an area of supervised machine learning closely related to regression and classification, but the goal is to learn from examples using a similarity function that measures how similar or related two objects are. It has applications in ranking, recommendation systems, visual identity tracking, face verification, and speaker verification.

1.4.2 Unsupervised learning

Unsupervised learning algorithms take a set of data that contains only inputs, and find structure in the data, like grouping or clustering of data points. The algorithms, therefore, learn from test data that has not been labeled, classified or categorized. Instead of responding to feedback, unsupervised learning algorithms identify commonalities in the data and react based

on the presence or absence of such commonalities in each new piece of data. A central application of unsupervised learning is in the field of density estimation in statistics, such as finding the probability density function. [32] Though unsupervised learning encompasses other domains involving summarizing and explaining data features.

Cluster analysis is the assignment of a set of observations into subsets (called clusters) so that observations within the same cluster are similar according to one or more predesignated criteria, while observations drawn from different clusters are dissimilar. Different clustering techniques make different assumptions on the structure of the data, often defined by some similarity metric and evaluated, for example, by internal compactness, or the similarity between members of the same cluster, and separation, the difference between clusters. Other methods are based on estimated density and graph connectivity.

1.4.3 Semi-supervised learning

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. In weakly supervised learning, the training labels are noisy, limited, or imprecise; however, these labels are often cheaper to obtain, resulting in larger effective training sets.

1.4.4 Reinforcement learning

Reinforcement learning is an area of machine learning concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. Due

to its generality, the field is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics and genetic algorithms. In machine learning, the environment is typically represented as a Markov decision process (MDP). Many reinforcement learning algorithms use dynamic programming techniques.[33] Reinforcement learning algorithms do not assume knowledge of an exact mathematical model of the MDP, and are used when exact models are infeasible. Reinforcement learning algorithms are used in autonomous vehicles or in learning to play a game against a human opponent.

1.5 Machine learning process

Machine learning is the process of imparting intelligence to machines seems daunting and impossible. But it is actually really easy. It can be broken down into 7 major steps.

1.5.1 Collecting Data

Machines initially learn from the data that you give them. It is of the utmost importance to collect reliable data so that machine learning model can find the correct patterns. The quality of the data that was fed in to the machine will determine how accurate the model is. Incorrect or outdated data, gives rise to wrong outcomes or predictions which are not relevant.

It is always better to procure data from a reliable source, as it will directly affect the outcome of the model. Good data is relevant, contains very few missing and repeated values, and has a good representation of the various subcategories/classes present. Given the problem that needs to be solved, then investigation to obtain data need to be done. The quality and quantity of information is very important since it will directly impact how well or badly the model will work. At times data may be available

from existing database or it needs to be created from scratch. If it is a small project you can create a spreadsheet that will later be easily exported as a CSV file. It is also common to use the web scraping technique to automatically collect information from various sources such as APIs.

1.5.2 Preparing the Data

Once data is collected visualizing data becomes a priority. Correlations between the different characteristics need to be assessed. It will be necessary to make a selection of characteristics since these characteristics will impact the execution times and the results. PCA is useful for reducing dimensions if necessary. Additionally, balance the amount of data for each class so that it is significant as the learning may be biased towards a type of response and when your model tries to generalize knowledge it will fail. You must also separate the data into two groups: one for *training* and the other for *model evaluation* which can be divided approximately in a ratio of 80/20 but it can vary depending on the case and the volume of data. At this stage, data needs to be processed through normalization, eliminating duplicates, and making error corrections.

1.5.3 Choose the model

A machine learning model determines the output you get after running a machine learning algorithm on the collected data. It is important to choose a model which is relevant to the task at hand. Over the years, scientists and engineers developed various models suited for different tasks like speech recognition, image recognition, prediction, etc. Apart from this, model needs to be chosen based on type of data i.e. categorical & non-categorical, numerical & non-numerical etc.

There are several models available for machine learning practice. Mostly they are algorithms available for different methods

such as classification, prediction, linear regression, clustering, K-Nearest Neighbor, Deep Learning, Neural Networks, and more. There are various models to be used depending on the data that is going to be processed such as images, sound, text, and numerical values. In the following table, there are few models and their applications that can be applied in projects.

Model	Applications
Logistic Regression	Price prediction
Fully connected networks	Classification
Convolutional Neural Networks	Image processing
Recurrent Neural Networks	Voice recognition
Random Forest	Fraud Detection
Reinforcement Learning	Learning by trial and error
Generative Models	Image creation
K-means	Segmentation
k-Nearest Neighbors	Recommendation systems
Bayesian Classifiers	Spam and noise filtering

Table 1.1: ML Algorithms and applications

1.5.4 Training the Model

Training is the most important step in machine learning. In training, data will be passed to the machine learning model to find patterns and make predictions. It results in the model learning from the data so that it can accomplish the task set. Over time, with training, the model gets better at predicting.

Data need to be trained smoothly to see an incremental improvement in the prediction rate. Remember to initialize the weights of your model randomly. The weights are the values that multiply or affect the relationships between the inputs and outputs. These weights are automatically adjusted by the selected algorithm while getting trained.

1.5.5 Evaluating the Model

After training the model, the model need to be tested for the performance. This is done by testing the performance of the model on previously unseen data. The unseen data used is the testing set that was split earlier. If testing was done on the same data which is used for training, that will not get an accurate measure, as the model is already used to the data, and finds the same patterns in it, as it previously did. This will give you disproportionately high accuracy. When used on testing data, it is possible to get accurate measure of how the model will perform and its speed. If the accuracy is less than or equal to 50%, that model will not be useful since it would be like tossing a coin to make decisions. If the precision is 90% or more, it means the model is a best one to rely upon.

1.5.6 Parameter Tuning

Once the model was created and evaluated, it should be verfied that whether the accuracy can be improved or not. This is done by tuning the parameters present in your model. Parameters are the variables in the model that the programmer generally decides. The accuracy can be maximum at a particular value of the parameter. Parameter tuning refers to finding these values.

At times a model can be overfitting or underfitting if the evaluation did not obtain good predictions and precision is not the minimum desired. It is possible to increase the number of training times as in *epochs*. Another important parameter is the one known as the "learning rate", which is usually a value that multiplies the gradient to gradually bring it closer to the global or local minimum to minimize the cost of the function.

Increasing your values by 0.1 units from 0.001 is not the same as this can significantly affect the model execution time. Indicate the maximum error allowed for the model. At times it can take a few minutes to hours, and even days, to train teh machine.

These parameters are often called *Hyperparameters*. This "tuning" is still more of an art than a science and will improve as you experiment. There are usually many parameters to adjust and when combined they can trigger all the options. Each algorithm has its own parameters to adjust. To name a few more, in Artificial Neural Networks (ANNs) architecture the number of hidden layers need to be decided and gradually to be tested with how many neurons each layer can accommodate. This will be a work of great effort and patience to give good results.

1.5.7 Making Predictions

In the end, the model will be used on the data to make predictions accurately. Usually, predictions are not a matter of concern, but accuracies will be calculated based on *confusion matrix*. A confusion matrix, also known as an error matrix, is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa. The name stems from the fact that whether the system is confusing two classes (i.e. commonly mislabeling one as another).

1.6 Train, validate & test data

In machine learning, a common task is the study and construction of algorithms that can learn from and make predictions on data.[34] Such algorithms function by making data-driven predictions or decisions, through building a mathematical model from input data. These input data used to build the model are usually divided in multiple data sets. In particular, three data sets are commonly used in different stages of the creation of the model: *training, validation* and *test* sets.

The model is initially fit on a training data set, which is a set of examples used to fit the parameters (e.g. weights of connections between neurons in artificial neural networks) of the model.[35] The model (e.g. a naive Bayes classifier) is trained on the training data set using a supervised learning method, for example using optimization methods such as gradient descent or stochastic gradient descent. In practice, the training data set often consists of pairs of an input vector (or scalar) and the corresponding output vector (or scalar), where the answer key is commonly denoted as the target (or label). The current model is run with the training data set and produces a result, which is then compared with the target, for each input vector in the training data set. Based on the result of the comparison and the specific learning algorithm being used, the parameters of the model are adjusted. The model fitting can include both variable selection and parameter estimation.

Successively, the fitted model is used to predict the responses for the observations in a second data set called the validation data set. The validation data set provides an unbiased evaluation of a model fit on the training data set while tuning the model's hyperparameters (e.g. the number of hidden units layers and layer widths in a neural network). Validation datasets can be used for regularization by early stopping (stopping training when the error on the validation data set increases, as this is a sign of over-fitting to the training data set). This simple procedure is complicated in practice by the fact that the validation dataset's error may fluctuate during training, producing multiple local minima. This complication has led to the creation of many *ad hoc* rules for deciding when over-fitting has truly begun.

Finally, the test data set is a data set used to provide an unbiased evaluation of a final model fit on the training data set. If the data in the test data set has never been used in training (for example in cross-validation), the test data set is also called a holdout data set. The term "validation set" is sometimes used instead of "test set" in some literature (e.g., if the original data

set was partitioned into only two subsets, the test set might be referred to as the validation set). Deciding the sizes and strategies for data set division in training, test and validation sets is very dependent on the problem and data available.

1.6.1 Training data set

A training data set is a data set of examples used during the learning process and is used to fit the parameters (e.g., weights) of, for example, a classifier.[36] For classification tasks, a supervised learning algorithm looks at the training data set to determine, or learn, the optimal combinations of variables that will generate a good predictive model.[37] The goal is to produce a trained (fitted) model that generalizes well to new, unknown data. The fitted model is evaluated using "new" examples from the held-out datasets (validation and test datasets) to estimate the model s accuracy in classifying new data. To reduce the risk of issues such as over-fitting, the examples in the validation and test datasets should not be used to train the model. Most approaches that search through training data for empirical relationships tend to overfit the data, meaning that they can identify and exploit apparent relationships in the training data that do not hold in general.

1.6.2 Validation data set

A validation data set is a data-set of examples used to tune the hyperparameters (i.e. the architecture) of a classifier. It is sometimes also called the development set or the "dev set". An example of a hyperparameter for artificial neural networks includes the number of hidden units in each layer. [38] It, as well as the testing set (as mentioned below), should follow the same probability distribution as the training data set.

In order to avoid overfitting, when any classification parameter needs to be adjusted, it is necessary to have a validation data set

in addition to the training and test datasets. For example, if the most suitable classifier for the problem is sought, the training data set is used to train the different candidate classifiers, the validation data set is used to compare their performances and decide which one to take and, finally, the test data set is used to obtain the performance characteristics such as *accuracy, sensitivity, specificity, F-measure*, and so on. The validation data set functions as a hybrid: it is training data used for testing, but neither as part of the low-level training nor as part of the final testing.

1.6.3 Test data set

A test data set is a data set that is independent of the training data set, but that follows the same probability distribution as the training data set. If a model fit to the training data set also fits the test data set well, minimal overfitting has taken place (see figure below). A better fitting of the training data set as opposed to the test data set usually points to over-fitting.

A test set is therefore a set of examples used only to assess the performance (i.e. generalization) of a fully specified classifier. To do this, the final model is used to predict classifications of examples in the test set. Those predictions are compared to the examples' true classifications to assess the model's accuracy.

In a scenario where both validation and test datasets are used, the test data set is typically used to assess the final model that is selected during the validation process. In the case where the original data set is partitioned into two subsets (training and test datasets), the test data set might assess the model only once (e.g., in the holdout method). Note that some sources advise against such a method. However, when using a method such as cross-validation, two partitions can be sufficient and effective since results are averaged after repeated rounds of model training and testing to help reduce bias and variability.

1.6.4 Cross validation

Cross-validation, sometimes called rotation estimation or out-of-sample testing, is any of various similar model validation techniques for assessing how the results of a statistical analysis will generalize to an independent data set. Cross-validation is a resampling method that uses different portions of the data to test and train a model on different iterations. It is mainly used in settings where the goal is prediction, and one wants to estimate how accurately a predictive model will perform in practice. In a prediction problem, a model is usually given a dataset of known data on which training is run (training dataset), and a dataset of unknown data (or first seen data) against which the model is tested (called the validation dataset or testing set). The goal of cross-validation is to test the model's ability to predict new data that was not used in estimating it, in order to flag problems like overfitting or selection bias and to give an insight on how the model will generalize to an independent dataset (i.e., an unknown dataset, for instance from a real problem).

One round of cross-validation involves partitioning a sample of data into complementary subsets, performing the analysis on one subset (called the training set), and validating the analysis on the other subset (called the validation set or testing set). To reduce variability, in most methods multiple rounds of cross-validation are performed using different partitions, and the validation results are combined (e.g. averaged) over the rounds to give an estimate of the model's predictive performance. In summary, cross-validation combines (averages) measures of fitness in prediction to derive a more accurate estimate of model prediction performance.

Method

Assume a model with one or more unknown parameters, and a data set to which the model can be fit (the training data set). The fitting process optimizes the model parameters to make the

model fit the training data as well as possible. If an independent sample of validation data is taken from the same population as the training data, it will generally turn out that the model does not fit the validation data as well as it fits the training data. The size of this difference is likely to be large especially when the size of the training data set is small, or when the number of parameters in the model is large. Cross-validation is a way to estimate the size of this effect.

In linear regression, there exist real response values *y1, ..., yn*, and n p-dimensional vector covariates *x1, ..., xn*. The components of the vector *xi* are denoted *xi1, ..., xip*. If least squares is used to fit a function in the form of a hyperplane $\hat{y} = a + \beta^T x$ to the data $(xi, yi)1 \leq i \leq n$, then the fit can be assessed using the mean squared error (MSE). The MSE for given estimated parameter values a and β on the training set $(xi, yi)1 \leq i \leq n$ is defined as:

$$\begin{aligned}\text{MSE} &= \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - a - \boldsymbol{\beta}^T \mathbf{x}_i)^2 \\ &= \frac{1}{n}\sum_{i=1}^{n}(y_i - a - \beta_1 x_{i1} - \cdots - \beta_p x_{ip})^2\end{aligned}$$

If the model is correctly specified, it can be shown under mild assumptions that the expected value of the MSE for the training set is $(n?p?1)/(n+p+1) < 1$ times the expected value of the MSE for the validation set (the expected value is taken over the distribution of training sets). Thus, a fitted model and computed MSE on the training set will result in an optimistically biased assessment of how well the model will fit an independent data set. This biased estimate is called the in-sample estimate of the fit, whereas the cross-validation estimate is an out-of-sample estimate.

Since in linear regression it is possible to directly compute the factor $(n?p?1)/(n+p+1)$ by which the training MSE underestimates the validation MSE under the assumption that the model

specification is valid, cross-validation can be used for checking whether the model has been overfitted, in which case the MSE in the validation set will substantially exceed its anticipated value. (Cross-validation in the context of linear regression is also useful in that it can be used to select an optimally regularized cost function.) In most other regression procedures (e.g. logistic regression), there is no simple formula to compute the expected out-of-sample fit. Cross-validation is, thus, a generally applicable way to predict the performance of a model on unavailable data using numerical computation in place of theoretical analysis.

1.6.5 Overfitting/Underfitting a Model

As we know, in machine learning the data is usually split into two subsets: *training data* and *testing data*, and fit our model on the train data, in order to make predictions on the test data. As a result, there are two issues while fitting a model. They are *overfitting* or *underfitting*. These issues must be tackled in order to maintain accuracy

Overfitting

Overfitting means that model is beign trained "too well". This usually happens when the model is too complex (i.e. too many features/variables compared to the number of observations). This model will be very accurate on the training data but will probably be very not accurate on untrained or new data. It is because this model is not generalized leading to bad results or prediction. Basically, when this happens, the model learns or describes the "noise" in the training data instead of the actual relationships between variables in the data. This noise, obviously, is not part in of any new dataset, and cannot be applied to it.

Underfitting

In contrast to overfitting, when a model is underfitted, it means that the model does not fit the training data and therefore misses the trends in the data. It also means the model cannot be generalized to new data. This is usually the result of a very simple model. This happens due to lack of enough predictors/independent variables. It could also happen when, for example, we fit a linear model (like linear regression) to data that is not linear. It almost goes without saying that this model will have poor predictive ability (on training data and can t be generalized to other data). It is worth noting the underfitting is not as prevalent as overfitting. Nevertheless, avoiding both of those problems in data analysis yields good outcomes.

1.6.6 Python practice

Scikit-Learn library and specifically the `train_test_split` method is used to split arrays or matrices into random train and test subsets. Following code snippet shows as to how packages can be imported. The another package is *numpy* which is used to simulate data sets required for our task.

```
>>> import numpy as np
>>> from sklearn.model_selection import train_test_split
>>> X, y = np.arange(10).reshape((5, 2)), range(5)
>>> X
array([[0, 1],
       [2, 3],
       [4, 5],
       [6, 7],
       [8, 9]])
>>> list(y)
[0, 1, 2, 3, 4]
```

In the above snippet there are two packages that were imported namely *numpy* and *sklearn*. The package *numpy* is useful for data simulations. The function `train_test_split` from the

package *sklearn* is useful for train, test and split tasks. Below code snippet performs the required tasks.

```
>>> X_train, X_test, y_train, y_test = train_test_split(X
                                ↪ , y, test_size=0.33,
                                ↪ random_state=42)
>>> X_train
array([[4, 5],
       [0, 1],
       [6, 7]])
>>> X_test
array([[2, 3],
       [8, 9]])
>>> y_train
[2, 0, 3]
>>> y_test
[1, 4]
```

Let's try fitting data using Linear Regression and then test our data with cross validations.

```
from sklearn.linear_model import LinearRegression

lm = LinearRegression()
model = lm.fit(X_train, y_train)
predictions = lm.predict(X_test)
print(model.score(X_test, y_test))
print(predictions)
```

The output would be

```
>>> model.score(X_test, y_test)
1.0

>>> predictions
array([1., 4.])
```

Compare `predictions` with `y_test` values. Perfect predictions. Isn't it? There are methods to know about intercept and coefficients. That is not required for this task. [39]. As far as cross validation is concerned, it is not necessary, because it is a perfect

fit as we know (score is 1). However, it can be done as shown below for practice

```
>>> scores = cross_val_score(model, X, y, cv=2)
>>> print("Cross-validated scores:", scores)
Cross-validated scores: [1. 1.]
>>> predictions = cross_val_predict(model, X, y, cv=2)
>>> for i in predictions:
...     print(i)
...
3.885780586188048e-16
1.0000000000000002
2.0000000000000004
3.0
4.0
```

Perfect predictions.

Exercises

1. Learn data simulations using Python. There are number of ways to make data sets quickly using Python programming. Appendix 1 [6] shows few methods to perform data simulations. Create few data variables such as numeric and non-numeric and try to create data sets using *pandas* library.
2. Take a sample data set with certain valid variables. Perform linear regression and explain regression analysis with the help of summary measures. [For instance, you can use *diabetes* data set available from *sklearn* library.]
3. Perform test, split and train tasks on simulated data sets.
4. Plot regression fits using your own data sets using simulation methods.
5. Demonstrate model fit, model score and model predictions using imported data sets. Explain cross validation and accuracy of models.
6. Use any data sets of your interest and calculate AIC, BIC for regression models.

Notes

[1]Mitchell, Tom (1997). Machine Learning. New York: McGraw Hill. ISBN 0-07-042807-7.

[2]Friedman, Jerome H. (1998). "Data Mining and Statistics: What's the connection?". Computing Science and Statistics. 29 (1): 39.

[3]Zhou, Victor (2019-12-20). "Machine Learning for Beginners: An Introduction to Neural Networks". Medium. Retrieved 2021-08-15.

[4]Ethem Alpaydin (2020). Introduction to Machine Learning (Fourth ed.). MIT. pp. xix, 13, 1318. ISBN 978-0262043793.

[5]von Neumann, John (1945), First Draft of a Report on the EDVAC (PDF), archived from the original (PDF) on March 14, 2013, retrieved August 24, 2011

[6]Turing's 1948 paper has been reprinted as Turing AM. Intelligent Machinery. In: Ince DC, editor. Collected works of AM Turing — Mechanical Intelligence. Elsevier Science Publishers, 1992.

[7]Richard Gross, Psychology: The Science of Mind and Behaviour 6E, Hachette UK, ISBN 978-1-4441-6436-7.

[8]Karban, R. (2015). Plant Learning and Memory. In: Plant Sensing and Communication. Chicago and London: The University of Chicago Press, pp. 31–44,

[9]Daniel L. Schacter; Daniel T. Gilbert; Daniel M. Wegner (2011) [2009]. Psychology, 2nd edition. Worth Publishers. p. 264

[10]Sujan, M. A., Huang, H., Braithwaite, J. (2017). Learning from incidents in health care: critique from a Safety-II perspective. Safety Science, 99, 115-121.

[11]Hartley, DM, Seid, M. Collaborative learning health systems: Science and practice. Learn Health Sys. 2021; 5(3):e10286. https://doi.org/10.1002/lrh2.10286

[12]Sheridan, Mary; Howard, Justine; Alderson, Dawn (2010). Play in Early Childhood: From Birth to Six Years. Oxon: Routledge. ISBN 978-1-136-83748-7.

[13]Hutchins, E., 2014. The cultural ecosystem of human cognition. Philosophical Psychology 27(1), 34-49.

[14]Samuel, Arthur (1959). "Some Studies in Machine Learning Using the Game of Checkers". IBM Journal of Research and Development. 3 (3): 210 229.

[15]R. Kohavi and F. Provost, "Glossary of terms," Machine Learning, vol. 30, no. 2 3, pp. 271 274, 1998.

[16]Gerovitch, Slava (9 April 2015). "How the Computer Got Its Revenge on the Soviet Union". Nautilus. Retrieved 19 September 2021.

[17]Duda, R., Hart P. Pattern Recognition and Scene Analysis, Wiley Interscience, 1973

[18]Mitchell, T. (1997). Machine

Learning. McGraw Hill. p. 2. ISBN 978-0-07-042807-2.

[19]Ku. Chhaya A. Khanzode and Dr. Ravindra D. Sarode, Advantages and Disadvantages of Artificial Intelligence and Machine Learning: A Literature Review, International Journal of Library Information Science, 9(1), 2020, pp. 30-36.

[20]Delveen, Shereen. Survey on Classification Algorithms for Data Mining:(Comparison and Evaluation). Computer Engineering and Intelligent Systems. Vol.4, No.8, 2013

[21]Sarle, Warren (1994). "Neural Networks and statistical models". CiteSeerX 10.1.1.27.699.

[22]Russell, Stuart; Norvig, Peter (2003) [1995]. Artificial Intelligence: A Modern Approach (2nd ed.). Prentice Hall. ISBN 978-0137903955.

[23]Langley, Pat (2011). "The changing science of machine learning". Machine Learning. 82 (3): 275 279. doi:10.1007/s10994-011-5242-y

[24]Alpaydin, Ethem (2010). Introduction to Machine Learning. MIT Press. p. 9. ISBN 978-0-262-01243-0.

[25]Bzdok, Danilo; Altman, Naomi; Krzywinski, Martin (2018). "Statistics versus Machine Learning". Nature Methods. 15 (4): 233234.

[26]Michael I. Jordan (2014-09-10). "statistics and machine learning". reddit. Retrieved 2014-10-01.

[27]Gareth James; Daniela Witten; Trevor Hastie; Robert Tibshirani (2013). An Introduction to Statistical Learning. Springer. p. vii.

[28]Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (Third ed.). Prentice Hall. ISBN 9780136042594.

[29]Mohri, Mehryar; Rostamizadeh, Afshin; Talwalkar, Ameet (2012). Foundations of Machine Learning. The MIT Press. ISBN 9780262018258.

[30]Mitchell, T. (1997). Machine Learning. McGraw Hill. p. 2. ISBN 978-0-07-042807-2.

[31]Alpaydin, Ethem (2010). Introduction to Machine Learning. MIT Press. p. 9. ISBN 978-0-262-01243-0.

[32]Jordan, Michael I.; Bishop, Christopher M. (2004). "Neural Networks". In Allen B. Tucker (ed.). Computer Science Handbook, Second Edition (Section VII: Intelligent Systems). Boca Raton, Florida: Chapman Hall/CRC Press LLC. ISBN 978-1-58488-360-9.

[33]van Otterlo, M.; Wiering, M. (2012). Reinforcement learning and markov decision processes. Reinforcement Learning. Adaptation, Learning, and Optimization. Vol. 12. pp. 3 42. doi:10.1007/978-3-642-27645-3$_1$.$ISBN978 - 3 - 642 - 27644 - 6$.

[34]Ron Kohavi; Foster Provost (1998). "Glossary of terms". Machine Learning. 30: 271 274.

[35]James, Gareth (2013). An Introduction to Statistical Learning: with Applications in R. Springer. p. 176.

[36]Ripley, B.D. (1996) Pattern Recognition and Neural Networks, Cambridge: Cambridge University Press, p. 354

[37]Larose, D. T.; Larose, C. D. (2014). Discovering knowledge in data : an introduction to data mining. Hoboken: Wiley.

[38]Ripley, B.D. (1996) Pattern Recognition and Neural Networks, Cambridge: Cambridge University Press, p. 354

[39]There are ways to access measures. Suppose, if the model object is `fit` The intercept and coefficients can be retrieved using `fit.intercept_` and `fit.coef_`

Chapter 2

Supervised Learning

Supervised learning (SL) is the machine learning task of learning a function that maps an input to an output based on example input-output pairs. It infers a function from labeled training data consisting of a set of training examples. In supervised learning, each example is a pair consisting of an input object (typically a vector) and a desired output value (also called the supervisory signal). A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. An optimal scenario will allow for the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way (see inductive bias). This statistical quality of an algorithm is measured through the so-called generalization error.[40] [41]

To solve a given problem of supervised learning, one has to perform the following steps:

1. *Determine the type of training examples.* Before doing

anything else, the user should decide what kind of data is to be used as a training set. In the case of handwriting analysis, for example, this might be a single handwritten character, an entire handwritten word, an entire sentence of handwriting or perhaps a full paragraph of handwriting.

2. *Gather a training set.* The training set needs to be representative of the real-world use of the function. Thus, a set of input objects is gathered and corresponding outputs are also gathered, either from human experts or from measurements.

3. *Determine the input feature representation of the learned function.* The accuracy of the learned function depends strongly on how the input object is represented. Typically, the input object is transformed into a feature vector, which contains a number of features that are descriptive of the object. The number of features should not be too large, because of the curse of dimensionality; but should contain enough information to accurately predict the output.

4. *Determine the structure of the learned function and corresponding learning algorithm.* For example, the engineer may choose to use support-vector machines or decision trees.

5. *Complete the design.* Run the learning algorithm on the gathered training set. Some supervised learning algorithms require the user to determine certain control parameters. These parameters may be adjusted by optimizing performance on a subset (called a validation set) of the training set, or via cross-validation.

6. *Evaluate the accuracy of the learned function.* After parameter adjustment and learning, the performance of the resulting function should be measured on a test set that is separate from the training set.

2.1 Algorithms

The most widely used learning algorithms are:

1. Support-vector machines
2. Linear regression
3. Logistic regression
4. Naive Bayes
5. Linear discriminant analysis
6. Decision trees
7. K-nearest neighbor algorithm
8. Neural networks (Multilayer perceptron)
9. Similarity learning

2.2 Applications

1. Bioinformatics
2. Cheminformatics
3. Quantitative structure activity relationship
4. Database marketing
5. Handwriting recognition
6. Information retrieval
7. Learning to rank
8. Information extraction recognition in computer vision character recognition detection
9. Pattern recognition
10. Speech recognition

11. Supervised learning is a special case of downward causation in biological systems
12. Landform classification using satellite imagery

2.3 Logistic regression

In statistics logistic regression is used to model the probability of a certain class or event. I will be focusing more on the basics and implementation of the model, and not go too deep into the math part. Logistic regression is similar to linear regression because both of these involve estimating the values of parameters used in the prediction equation based on the given training data. Linear regression predicts the value of some continuous, dependent variable. Whereas logistic regression predicts the probability of an event or class that is dependent on other factors. Thus the output of logistic regression always lies between 0 and 1. Because of this property it is commonly used for classification purpose.

2.3.1 Logistic Model

Consider a model with features *x1, x2, x3 ... xn*. Let the binary output be denoted by *Y*, that can take the values 0 or 1. Let *p* be the probability of *Y = 1*, we can denote it as *p = P(Y=1)*. The mathematical relationship between these variables can be denoted as:

$$\log_b \frac{p}{1-p} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n \qquad (2.1)$$

Here the term *p/(1-p)* is known as the odds and denotes the likelihood of the event taking place. Thus *ln(p/(1-p))* is known as the log odds and is simply used to map the probability that lies between 0 and 1 to a range between $(-\infty, +\infty)$. The terms *b0, b1, b2 ...* are parameters (or weights) that we will estimate during training. So this is just the basic math behind what we

are going to do. We are interested in the probability p in this equation. So we simplify the equation to obtain the value of p:

1. The log term ln on the LHS can be removed by raising the RHS as a power of e:

$$\frac{p}{1-p} = b^{\beta_0+\beta_1 x_1+\beta_2 x_2}. \tag{2.2}$$

2. Now we can easily simplify to obtain the value of p:

$$p = \frac{b^{\beta_0+\beta_1 x_1+\beta_2 x_2}}{b^{\beta_0+\beta_1 x_1+\beta_2 x_2}+1} = \frac{1}{1+b^{-(\beta_0+\beta_1 x_1+\beta_2 x_2)}} = S_b(\beta_0+\beta_1 x_1+\beta_2 x_2). \tag{2.3}$$

This actually turns out to be the equation of the Sigmoid Function which is widely used in other machine learning applications. The Sigmoid Function is given by:

$$\sigma(t) = \frac{e^t}{e^t+1} = \frac{1}{1+e^{-t}} \tag{2.4}$$

Now we will be using the above derived equation to make our predictions. Before that we will train our model to obtain the values of our parameters *b0, b1, b2 ...* that result in least error. This is where the error or loss function comes in.

2.3.2 Loss Function

The loss is basically the error in our predicted value. In other words it is a difference between our predicted value and the actual value. We will be using the L2 Loss Function to calculate the error. Theoretically you can use any function to calculate the error. This function can be broken down as:

1. Let the actual value be y_i. Let the value predicted using our model be denoted as $\hat{y}_i$. Find the difference between the actual and predicted value.

2. Square this difference.

3. Find the sum across all the values in training data.

$$L = \sum_{i=1}^{n} (y + i - \bar{y_i})^2 \tag{2.5}$$

Now that we have the error, we need to update the values of our parameters to minimize this error. This is where the "learning" actually happens, since our model is updating itself based on it s previous output to obtain a more accurate output in the next step. Hence with each iteration our model becomes more and more accurate. We will be using the Gradient Descent Algorithm to estimate our parameters. Another commonly used algorithm is the *Maximum Likelihood Estimation*.

2.3.3 The Gradient Descent Algorithm

You might know that the partial derivative of a function at it s minimum value is equal to *0*. So gradient descent basically uses this concept to estimate the parameters or weights of our model by minimizing the loss function. For simplicity, for the rest of this tutorial let us assume that our output depends only on a single feature x. So we can rewrite our equation as:

$$y_i = p = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x)}} \tag{2.6}$$

Thus we need to estimate the values of weights *b0* and *b1* using our given training data.

1. Initially let *b0=0* and *b1=0*. Let L be the learning rate. The learning rate controls by how much the values of b0 and b1 are updated at each step in the learning process. Here let *L=0.001*.

2. Calculate the partial derivative with respect to b0 and b1. The value of the partial derivative will tell us how far the loss function is from it s minimum value. It is a measure of how much our weights need to be updated to attain minimum or ideally *0* error. In case you have more than one feature, you need to calculate the partial derivative for each weight *b0, b1 ... bn* where n is the number of features.

$$D_{b0} = -2\sum_{i=1}^{n}(y_i - \bar{y_i}) \times y_i \times (1 - \bar{y_i}) D_{b1} = -2\sum_{i=1}^{n}(y_i - \bar{y_i}) \times y_i \times (1 - \bar{y_i}) \times x_i \tag{2.7}$$

3. Next we update the values of *b0* and *b1*:

$$b_0 = b_0 - L \times D_{b0} b_1 = b_1 - L \times D_{b1} \tag{2.8}$$

4. We repeat this process until our loss function is a very small value or ideally reaches 0 (meaning no errors and 100% accuracy). The number of times we repeat this learning process is known as iterations or epochs.

2.3.4 Python practice

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegressionCV
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegressionCV(cv=5, random_state=0).fit(
                                ↪ X, y)
>>> clf.predict(X[:2, :])
array([0, 0])
>>> clf.predict_proba(X[:2, :]).shape
(2, 3)
>>> clf.score(X, y)
0.98...
```

2.4 K-Nearest Neighbors Algorithm (k-NN)

The k-nearest neighbors algorithm (k-NN) is a non-parametric classification method first developed by *Evelyn Fix* and *Joseph Hodges* in 1951, and later expanded by *Thomas Cover*. [42] [43] It is used for classification and regression. In both cases, the input consists of the k closest training examples in data set. The output depends on whether *k-NN* is used for classification or regression:

1. In k-NN classification, the output is a class membership. An object is classified by a plurality vote of its neighbors, with the object being assigned to the class most common among its k nearest neighbors (k is a positive integer, typically small). If $k = 1$, then the object is simply assigned to the class of that single nearest neighbor.

2. In k-NN regression, the output is the property value for the object. This value is the average of the values of k nearest neighbors.

k-NN is a type of classification where the function is only approximated locally and all computation is deferred until function evaluation. Since this algorithm relies on distance for classification, if the features represent different physical units or come in vastly different scales then normalizing the training data can improve its accuracy dramatically. [44] [45]

Both for classification and regression, a useful technique can be to assign weights to the contributions of the neighbors, so that the nearer neighbors contribute more to the average than the more distant ones. For example, a common weighting scheme consists in giving each neighbor a weight of 1/d, where d is the distance to the neighbor. The neighbors are taken from a set of objects for which the class (for k-NN classification) or the object property value (for k-NN regression) is known. This can

be thought of as the training set for the algorithm, though no explicit training step is required. A peculiarity of the k-NN algorithm is that it is sensitive to the local structure of the data.

Suppose we have pairs $(X_1, Y_1), (X_2, Y_2), \ldots, (X_n, Y_n)$ taking values in $R^d \times \{1, 2\}$, where Y is the class label of X, so that $X|Y = r \sim P_r$ for $r = 1, 2$ (and probability distributions P_r. Given some norm $\|\cdot\|$ on R^d and a point $x \in R^d$, let $(X_{(1)}, Y_{(1)}), \ldots, (X_{(n)}, Y_{(n)})$ be a reordering of the training data such that $\|X_{(1)} - x\| \leq \cdots \leq \|X_{(n)} - x\|$.

2.4.1 Algorithm

The training examples are vectors in a multidimensional feature space, each with a class label. The training phase of the algorithm consists only of storing the feature vectors and class labels of the training samples. In the classification phase, k is a user-defined constant, and an unlabeled vector (a query or test point) is classified by assigning the label which is most frequent among the k training samples nearest to that query point.

A commonly used distance metric for continuous variables is Euclidean distance. For discrete variables, such as for text classification, another metric can be used, such as the overlap metric (or Hamming distance). In the context of gene expression microarray data, for example, k-NN has been employed with correlation coefficients, such as Pearson and Spearman, as a metric. Often, the classification accuracy of k-NN can be improved significantly if the distance metric is learned with specialized algorithms such as Large Margin Nearest Neighbor or Neighbourhood components analysis.

A drawback of the basic "majority voting" classification occurs when the class distribution is skewed. That is, examples of a more frequent class tend to dominate the prediction of the new example, because they tend to be common among the k nearest neighbors due to their large number. One way to overcome this

problem is to weight the classification, taking into account the distance from the test point to each of its k nearest neighbors. The class (or value, in regression problems) of each of the k nearest points is multiplied by a weight proportional to the inverse of the distance from that point to the test point. Another way to overcome skew is by abstraction in data representation. For example, in a self-organizing map (SOM), each node is a representative (a center) of a cluster of similar points, regardless of their density in the original training data. K-NN can then be applied to the SOM.

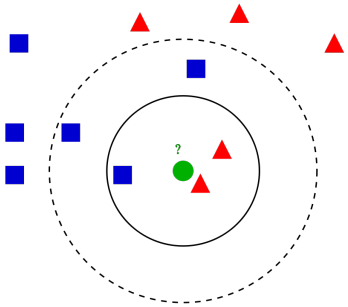

Figure 2.1: Example of k-NN classification. The test sample (green dot) should be classified either to blue squares or to red triangles. If $k = 3$ (solid line circle) it is assigned to the red triangles because there are 2 triangles and only 1 square inside the inner circle. If $k = 5$ (dashed line circle) it is assigned to the blue squares (3 squares vs. 2 triangles inside the outer circle).

2.4.2 Python practice

Demonstrate the resolution of a regression problem using a k-Nearest Neighbor and the interpolation of the target using both barycenter and constant weights.

```
import numpy as np
import matplotlib.pyplot as plt
```

```
from sklearn import neighbors

np.random.seed(0)
X = np.sort(5 * np.random.rand(40, 1), axis=0)
T = np.linspace(0, 5, 500)[:, np.newaxis]
y = np.sin(X).ravel()

# Add noise to targets
y[::5] += 1 * (0.5 - np.random.rand(8))

# Fit regression model
n_neighbors = 5

for i, weights in enumerate(['uniform', 'distance']):
    knn = neighbors.KNeighborsRegressor(n_neighbors,
                                        ↪ weights=weights)
    y_ = knn.fit(X, y).predict(T)

    plt.subplot(2, 1, i + 1)
    plt.scatter(X, y, color='darkorange', label='data')
    plt.plot(T, y_, color='navy', label='prediction')
    plt.axis('tight')
    plt.legend()
    plt.title("KNeighborsRegressor (k = %i, weights = '%s
                                        ↪ ')" % (n_neighbors,
                                        ↪ weights))

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

Figure 2.2: KNN for both *Uniform* and *distance* based weights.

2.5 Decision Trees

Decision Trees (DTs) are a non-parametric supervised learning method used for classification and regression. The goal is to create a model that predicts the value of a target variable by learning simple decision rules inferred from the data features. A tree can be seen as a piecewise constant approximation. For

instance, in the example below, decision trees learn from data to approximate a sine curve with a set of if-then-else decision rules. The deeper the tree, the more complex the decision rules and the fitter the model.

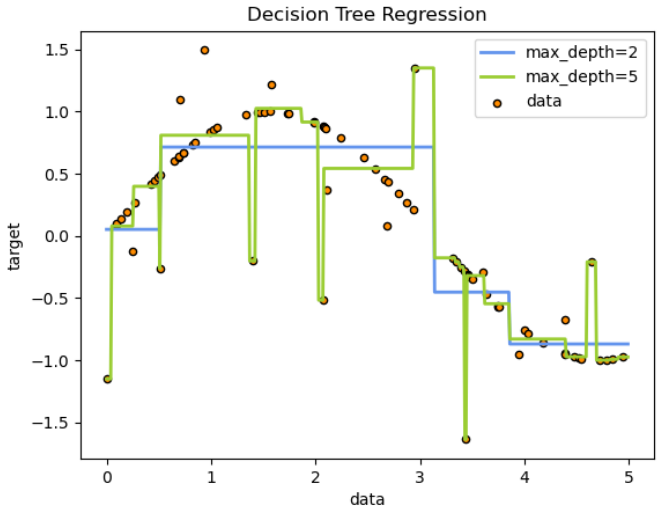

Figure 2.3: Decision Tree Regression

Some advantages of decision trees are:

1. Simple to understand and to interpret. Trees can be visualised.

2. Requires little data preparation. Other techniques often require data normalisation, dummy variables need to be created and blank values to be removed. Note however that this module does not support missing values.

3. The cost of using the tree (i.e., predicting data) is logarithmic in the number of data points used to train the

tree.

4. Able to handle both numerical and categorical data. However scikit-learn implementation does not support categorical variables for now. Other techniques are usually specialised in analysing datasets that have only one type of variable. See algorithms for more information.

5. Able to handle multi-output problems.

6. Uses a white box model. If a given situation is observable in a model, the explanation for the condition is easily explained by boolean logic. By contrast, in a black box model (e.g., in an artificial neural network), results may be more difficult to interpret.

7. Possible to validate a model using statistical tests. That makes it possible to account for the reliability of the model.

8. Performs well even if its assumptions are somewhat violated by the true model from which the data were generated.

The disadvantages of decision trees include:

1. Decision-tree learners can create over-complex trees that do not generalise the data well. This is called overfitting. Mechanisms such as pruning, setting the minimum number of samples required at a leaf node or setting the maximum depth of the tree are necessary to avoid this problem.

2. Decision trees can be unstable because small variations in the data might result in a completely different tree being generated. This problem is mitigated by using decision trees within an ensemble.

3. Predictions of decision trees are neither smooth nor continuous, but piecewise constant approximations as seen in the above figure. Therefore, they are not good at extrapolation.

4. The problem of learning an optimal decision tree is known to be NP-complete under several aspects of optimality and even for simple concepts. Consequently, practical decision-tree learning algorithms are based on heuristic algorithms such as the greedy algorithm where locally optimal decisions are made at each node. Such algorithms cannot guarantee to return the globally optimal decision tree. This can be mitigated by training multiple trees in an ensemble learner, where the features and samples are randomly sampled with replacement.

5. There are concepts that are hard to learn because decision trees do not express them easily, such as XOR, parity or multiplexer problems.

6. Decision tree learners create biased trees if some classes dominate. It is therefore recommended to balance the dataset prior to fitting with the decision tree.

2.5.1 Classification

`DecisionTreeClassifier` is a class capable of performing multi-class classification on a dataset. As with other classifiers, this class takes as input two arrays: an array X, sparse or dense, of shape $(n\_samples, n\_features)$ holding the training samples, and an array Y of integer values, shape $(n\_samples,)$, holding the class labels for the training samples:

```
>>> from sklearn import tree
>>> X = [[0, 0], [1, 1]]
>>> Y = [0, 1]
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, Y)
```

After being fitted, the model can then be used to predict the class of samples:

```
>>> clf.predict([[2., 2.]])
array([1])
```

In case that there are multiple classes with the same and highest probability, the classifier will predict the class with the lowest index amongst those classes. As an alternative to outputting a specific class, the probability of each class can be predicted, which is the fraction of training samples of the class in a leaf:

```
>>> clf.predict_proba([[2., 2.]])
array([[0., 1.]])
```

`DecisionTreeClassifier` is capable of both binary (where the labels are *[-1, 1]*) classification and multiclass (where the labels are *[0,..., K-1]*) classification. Using the Iris dataset, we can construct a tree as follows:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn import tree
>>> iris = load_iris()
>>> X, y = iris.data, iris.target
>>> clf = tree.DecisionTreeClassifier()
>>> clf = clf.fit(X, y)
```

Once trained, you can plot the tree with the `plot_tree` function:

```
>>> tree.plot_tree(clf)
```

The method `export_graphviz` in *Graphviz* package is useful to export the tree. If you use the conda package manager, the graphviz binaries and the python package can be installed with `conda install python-graphviz`. [46] Alternatively binaries for graphviz can be downloaded from the graphviz project homepage, and the Python wrapper installed from pypi with *pip install graphviz*. Below is an example graphviz export of the above tree trained on the entire iris dataset; the results are saved in an output file *iris.pdf*:

```
>>> import graphviz
>>> dot_data = tree.export_graphviz(clf, out_file=None)
>>> graph = graphviz.Source(dot_data)
>>> graph.render("iris")
```

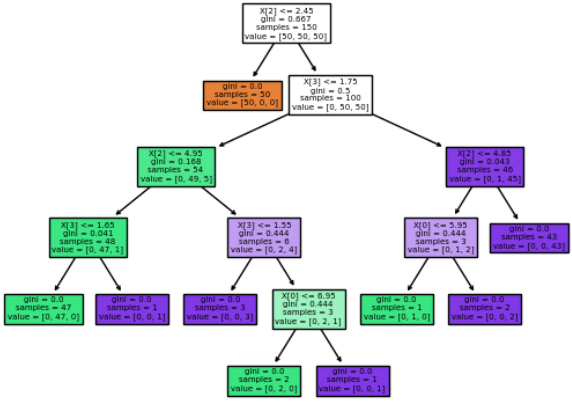

Figure 2.4: Decision Fit on Iris Data Set

The `export_graphviz` exporter also supports a variety of aesthetic options, including coloring nodes by their class (or value for regression) and using explicit variable and class names if desired. Jupyter notebooks also render these plots inline automatically:

```
>>> dot_data = tree.export_graphviz(clf, out_file=None,
...                         feature_names=iris.feature_names
                                ↪ ,
...                         class_names=iris.target_names,
...                         filled=True, rounded=True,
...                         special_characters=True)
>>> graph = graphviz.Source(dot_data)
>>> graph
```

2.5.2 Regression

Decision trees can also be applied to regression problems, using the `DecisionTreeRegressor` class. As in the classification setting, the fit method will take as argument arrays X and y, only that in this case y is expected to have floating point values instead of integer values:

```
>>> from sklearn import tree
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> clf = tree.DecisionTreeRegressor()
>>> clf = clf.fit(X, y)
>>> clf.predict([[1, 1]])
array([0.5])
```

2.6 Support Vector Machines

Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection. The advantages of support vector machines are:

1. Effective in high dimensional spaces.
2. Still effective in cases where number of dimensions is greater than the number of samples.
3. Uses a subset of training points in the decision function (called support vectors), so it is also memory efficient.
4. *Versatile*: different Kernel functions can be specified for the decision function. Common kernels are provided, but it is also possible to specify custom kernels.

The disadvantages of support vector machines include:

1. If the number of features is much greater than the number of samples, avoid over-fitting in choosing Kernel functions and regularization term is crucial.

2. SVMs do not directly provide probability estimates, these are calculated using an expensive five-fold cross-validation (see Scores and probabilities, below).

The support vector machines in scikit-learn support both dense (`numpy.ndarray` and convertible to that by `numpy.asarray`) and sparse (any `scipy.sparse`) sample vectors as input. However, to use an SVM to make predictions for sparse data, it must have been fit on such data. For optimal performance, use C-ordered `numpy.ndarray` (dense) or `scipy.sparse.csr_matrix` (sparse) with `dtype=float64`.

2.6.1 Classification

SVC, NuSVC and *LinearSVC* are classes capable of performing binary and multi-class classification on a dataset. *SVC* and *NuSVC* are similar methods, but accept slightly different sets of parameters and have different mathematical formulations (see section Mathematical formulation). On the other hand, *LinearSVC* is another (faster) implementation of Support Vector Classification for the case of a linear kernel. Note that *LinearSVC* does not accept parameter kernel, as this is assumed to be linear. It also lacks some of the attributes of *SVC* and *NuSVC*, like `support_`. As other classifiers, SVC, NuSVC and *LinearSVC* takes two input arrays. An array X with a shape of $(n\_samples, n\_features)$ holding the training samples, and an array y of class labels (strings or integers), of shape $(n\_samples)$.

```
>>> from sklearn import svm
>>> X = [[0, 0], [1, 1]]
>>> y = [0, 1]
>>> clf = svm.SVC()
>>> clf.fit(X, y)
SVC()
```

After being fitted, the model can then be used to predict new values:

```
>>> clf.predict([[2., 2.]])
array([1])
```

SVMs decision function (detailed in the Mathematical formulation) depends on some subset of the training data, called the support vectors. Some properties of these support vectors can be found in attributes *support_vectors_*, *support_* and *n_support_*:

```
>>> # get support vectors
>>> clf.support_vectors_
array([[0., 0.],
       [1., 1.]])
>>> # get indices of support vectors
>>> clf.support_
array([0, 1]...)
>>> # get number of support vectors for each class
>>> clf.n_support_
array([1, 1]...)
```

2.6.2 Regression

The method of Support Vector Classification can be extended to solve regression problems. This method is called Support Vector Regression. The model produced by support vector classification (as described above) depends only on a subset of the training data, because the cost function for building the model does not care about training points that lie beyond the margin. Analogously, the model produced by Support Vector Regression depends only on a subset of the training data, because the cost function ignores samples whose prediction is close to their target.

There are three different implementations of Support Vector Regression: *SVR, NuSVR* and *LinearSVR. LinearSVR* provides a faster implementation than *SVR* but only considers the linear kernel, while *NuSVR* implements a slightly different formulation than *SVR* and *LinearSVR.* As with classification classes, the fit method will take as argument vectors *X, y*, only that in this

case y is expected to have floating point values instead of integer values:

```
>>> from sklearn import svm
>>> X = [[0, 0], [2, 2]]
>>> y = [0.5, 2.5]
>>> regr = svm.SVR()
>>> regr.fit(X, y)
SVR()
>>> regr.predict([[1, 1]])
array([1.5])
```

2.7 Naïve Bayes Algorithm

Naive Bayes methods are a set of supervised learning algorithms based on applying Bayes' theorem with the "naive" assumption of conditional independence between every pair of features given the value of the class variable. Bayes' theorem states the following relationship, given class variable y and dependent feature vector *x1* through *xn*, :

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y)P(x_1, \ldots, x_n \mid y)}{P(x_1, \ldots, x_n)} \tag{2.9}$$

Using the naive conditional independence assumption that

$$P(x_i|y, x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n) = P(x_i|y), \tag{2.10}$$

for all i, this relationship is simplified to

$$P(y \mid x_1, \ldots, x_n) = \frac{P(y) \prod_{i=1}^{n} P(x_i \mid y)}{P(x_1, \ldots, x_n)} \tag{2.11}$$

Since $P(x_1, \ldots, x_n)$ is constant given the input, we can use the following classification rule:

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow \tag{2.12}$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

$$P(y \mid x_1, \ldots, x_n) \propto P(y) \prod_{i=1}^{n} P(x_i \mid y)$$

$$\Downarrow \tag{2.13}$$

$$\hat{y} = \arg\max_y P(y) \prod_{i=1}^{n} P(x_i \mid y),$$

and we can use Maximum A Posteriori (MAP) estimation to estimate $P(y)$ and $P(x_i \mid y)$; the former is then the relative frequency of class in the training set.

The different naive Bayes classifiers differ mainly by the assumptions they make regarding the distribution of $P(x_i \mid y)$.

In spite of their apparently over-simplified assumptions, naive Bayes classifiers have worked quite well in many real-world situations, famously document classification and spam filtering. They require a small amount of training data to estimate the necessary parameters. (For theoretical reasons why naive Bayes works well, and on which types of data it does, see the references below.)

Naive Bayes learners and classifiers can be extremely fast compared to more sophisticated methods. The decoupling of the class conditional feature distributions means that each distribution can be independently estimated as a one dimensional distribution. This in turn helps to alleviate problems stemming from the curse of dimensionality.

On the flip side, although naive Bayes is known as a decent classifier, it is known to be a bad estimator, so the probability outputs from *predict_proba* are not to be taken too seriously.

2.7.1 Gaussian Naive Bayes

`GaussianNB` implements the Gaussian Naive Bayes algorithm for classification. The likelihood of the features is assumed to be Gaussian:

$$P(x_i \mid y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.14}$$

The parameters σ_y and μ_y are estimated using maximum likelihood.

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.naive_bayes import GaussianNB
>>> X, y = load_iris(return_X_y=True)
>>> X_train, X_test, y_train, y_test = train_test_split(X
                              ↪ , y, test_size=0.5,
                              ↪ random_state=0)
>>> gnb = GaussianNB()
>>> y_pred = gnb.fit(X_train, y_train).predict(X_test)
>>> print("Number of mislabeled points out of a total %d
                              ↪ points : %d"
...       % (X_test.shape[0], (y_test != y_pred).sum()))
Number of mislabeled points out of a total 75 points : 4
```

There are different types of Naive Bayes algorithms. Following is the description

2.8 Random Forest

Random forests or random decision forests is an ensemble learning method for classification, regression and other tasks that operates by constructing a multitude of decision trees at training

Type	Desc.
Multinomial Naive Bayes	for multinomially distributed data
Complement Naive Bayes	suits for imbalanced data sets
Bernoulli Naive Bayes	data that is distributed according to multivariate Bernoulli distributions
Categorical Naive Bayes	for categorically distributed data
Out-of-core naive Bayes model fitting	large scale classification problems for which the full training set might not fit in memory.

Table 2.1: Different types of Naive Bayes algorithms

time. For classification tasks, the output of the random forest is the class selected by most trees. For regression tasks, the mean or average prediction of the individual trees is returned. Random decision forests correct for decision trees' habit of overfitting to their training set. Random forests generally outperform decision trees, but their accuracy is lower than gradient boosted trees. However, data characteristics can affect their performance.

The first algorithm for random decision forests was created in 1995 by Tin Kam Ho using the random subspace method, which, in Ho's formulation, is a way to implement the "stochastic discrimination" approach to classification proposed by Eugene Kleinberg.

A random forest is a meta estimator that fits a number of decision tree classifiers on various sub-samples of the dataset and uses averaging to improve the predictive accuracy and control over-fitting. The sub-sample size is controlled with the *max_samples* parameter if *bootstrap=True* (default), otherwise the whole dataset is used to build each tree.

Random forests are frequently used as "blackbox" models in businesses, as they generate reasonable predictions across a wide range of data while requiring little configuration.

The default values for the parameters controlling the size of the trees (e.g. *max_depth*, *min_samples_leaf*, etc.) lead to fully grown and unpruned trees which can potentially be very large on some data sets. To reduce memory consumption, the complexity and size of the trees should be controlled by setting those parameter values.

The features are always randomly permuted at each split. Therefore, the best found split may vary, even with the same training data, $max\_features = n\_features$ and $bootstrap = False$, if the improvement of the criterion is identical for several splits enumerated during the search of the best split. To obtain a deterministic behaviour during fitting, *random_state* has to be fixed.

```
>>> from sklearn.ensemble import RandomForestClassifier
>>> from sklearn.datasets import make_classification
>>> X, y = make_classification(n_samples=1000, n_features
                               ↪ =4,
...                              n_informative=2,
                               ↪ n_redundant=0,
...                              random_state=0, shuffle=
                               ↪ False)
>>> clf = RandomForestClassifier(max_depth=2,
                               ↪ random_state=0)
>>> clf.fit(X, y)
RandomForestClassifier(...)
>>> print(clf.predict([[0, 0, 0, 0]]))
[1]
```

2.9 Rule based learning

2.9.1 Apriori Algorithm

Notes

[40]Stuart J. Russell, Peter Norvig (2010) Artificial Intelligence: A Modern Approach, Third Edition, Prentice Hall ISBN 9780136042594.

[41]Cabannes, Vivien; Rudi, Alessandro; Bach, Francis (2021). "Fast rates in Structured Prediction". COLT. arXiv:2102.00760.

[42]Fix, Evelyn; Hodges, Joseph L. (1951). Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties (PDF) (Report). USAF School of Aviation Medicine, Randolph Field, Texas.

[43]Altman, Naomi S. (1992). "An introduction to kernel and nearest-neighbor nonparametric regression" (PDF). The American Statistician. 46 (3): 175 185. doi:10.1080/00031305.1992.10475879. hdl:1813/31637.

[44]Piryonesi S. Madeh; El-Diraby Tamer E. (2020-06-01). "Role of Data Analytics in Infrastructure Asset Management: Overcoming Data Size and Quality Problems". Journal of Transportation Engineering, Part B: Pavements. 146 (2): 04020022.

[45]Hastie, Trevor. (2001). The elements of statistical learning : data mining, inference, and prediction : with 200 full-color illustrations. Tibshirani, Robert., Friedman, J. H. (Jerome H.). New York: Springer. ISBN 0-387-95284-5. OCLC 46809224.

[46]Visit https://www.graphviz.org/ for more details.

Chapter 3

Unsupervised Learning

Unsupervised learning (UL) is a type of algorithm that learns patterns from untagged data. The hope is that, through mimicry, the machine is forced to build a compact internal representation of its world and then generate imaginative content. In contrast to supervised learning (SL) where data is tagged by a human, e.g. as "car" or "fish" etc, UL exhibits self-organization that captures patterns as neuronal predilections or probability densities. The other levels in the supervision spectrum are reinforcement learning where the machine is given only a numerical performance score as its guidance, and semi-supervised learning where a smaller portion of the data is tagged. Two broad methods in UL are Neural Networks and Probabilistic Methods. [47]

3.1 Approaches

Some of the most common algorithms used in unsupervised learning include: (1) Clustering, (2) Anomaly detection, (3) Neural Networks, and (4) Approaches for learning latent variable models. Each approach uses several methods as follows.

1. *Clustering methods* include: hierarchical clustering, k-means, mixture models, DBSCAN, and OPTICS algorithm
2. *Anomaly detection methods* include: Local Outlier Factor, and Isolation Forest
3. Approaches for learning latent variable models such as Expectation maximization algorithm (EM), Method of moments, and Blind signal separation techniques (Principal component analysis, Independent component analysis, Non-negative matrix factorization, Singular value decomposition)
4. *Neural Networks methods* include: Autoencoders, Deep Belief Nets, Hebbian Learning, Generative adversarial networks, and Self-organizing map

3.2 Clustering

Cluster analysis was originated in anthropology by Driver and Kroeber in 1932 and introduced to psychology by Joseph Zubin in 1938 and Robert Tryon in 1939 and famously used by Cattell beginning in 1943 for trait theory classification in personality psychology. [48] [49] [50] [51]

Cluster analysis or clustering is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups (clusters). It is a main task of exploratory data analysis, and a common technique for statistical data analysis, used in

many fields, including pattern recognition, image analysis, information retrieval, bioinformatics, data compression, computer graphics and machine learning.

Cluster analysis itself is not one specific algorithm, but the general task to be solved. It can be achieved by various algorithms that differ significantly in their understanding of what constitutes a cluster and how to efficiently find them. Popular notions of clusters include groups with small distances between cluster members, dense areas of the data space, intervals or particular statistical distributions. Clustering can therefore be formulated as a multi-objective optimization problem. The appropriate clustering algorithm and parameter settings (including parameters such as the distance function to use, a density threshold or the number of expected clusters) depend on the individual data set and intended use of the results. Cluster analysis as such is not an automatic task, but an iterative process of knowledge discovery or interactive multi-objective optimization that involves trial and failure. It is often necessary to modify data preprocessing and model parameters until the result achieves the desired properties.

Besides the term clustering, there are a number of terms with similar meanings, including automatic classification, numerical taxonomy, botryology, typological analysis, and community detection. The subtle differences are often in the use of the results: while in data mining, the resulting groups are the matter of interest, in automatic classification the resulting discriminative power is of interest.

3.2.1 Algorithms for clustering

Clustering algorithms can be categorized based on their cluster model. The following overview will only list the most prominent examples of clustering algorithms, as there are possibly over 100 published clustering algorithms. Not all provide models for their clusters and can thus not easily be categorized. An overview of

algorithms explained in Wikipedia can be found in the list of statistics algorithms.

There is no objectively "correct" clustering algorithm, but as it was noted, "clustering is in the eye of the beholder." The most appropriate clustering algorithm for a particular problem often needs to be chosen experimentally, unless there is a mathematical reason to prefer one cluster model over another. An algorithm that is designed for one kind of model will generally fail on a data set that contains a radically different kind of model. For example, k-means cannot find non-convex clusters. [52] Following are some of the algorithms available for clustering.

1. *Connectivity models*: for example, hierarchical clustering builds models based on distance connectivity.
2. *Centroid models*: for example, the k-means algorithm represents each cluster by a single mean vector.
3. *Distribution models*: clusters are modeled using statistical distributions, such as multivariate normal distributions used by the expectation-maximization algorithm.
4. *Density models*: for example, DBSCAN and OPTICS defines clusters as connected dense regions in the data space.
5. *Subspace models*: in biclustering (also known as co-clustering or two-mode-clustering), clusters are modeled with both cluster members and relevant attributes.
6. *Group models*: some algorithms do not provide a refined model for their results and just provide the grouping information.
7. *Graph-based models*: a clique, that is, a subset of nodes in a graph such that every two nodes in the subset are connected by an edge can be considered as a prototypical form of cluster. Relaxations of the complete connectivity requirement (a fraction of the edges can be missing) are known as quasi-cliques, as in the HCS clustering algorithm.

8. *Signed graph models*: Every path in a signed graph has a sign from the product of the signs on the edges. Under the assumptions of balance theory, edges may change sign and result in a bifurcated graph. The weaker "clusterability axiom" (no cycle has exactly one negative edge) yields results with more than two clusters, or subgraphs with only positive edges.

9. *Neural models:* the most well known unsupervised neural network is the self-organizing map and these models can usually be characterized as similar to one or more of the above models, and including subspace models when neural networks implement a form of Principal Component Analysis or Independent Component Analysis.

A "clustering" is essentially a set of such clusters, usually containing all objects in the data set. Additionally, it may specify the relationship of the clusters to each other, for example, a hierarchy of clusters embedded in each other. Clusterings can be roughly distinguished as:

1. *Hard clustering*: each object belongs to a cluster or not

2. *Soft clustering (fuzzy clustering)*: each object belongs to each cluster to a certain degree (for example, a likelihood of belonging to the cluster)

There are also finer distinctions possible, for example:

1. *Strict partitioning clustering*: each object belongs to exactly one cluster

2. *Strict partitioning clustering with outliers*: objects can also belong to no cluster, and are considered outliers Overlapping clustering (also: alternative clustering, multi-view clustering): objects may belong to more than one cluster; usually involving hard clusters

3. *Hierarchical clustering*: objects that belong to a child cluster also belong to the parent cluster

4. *Subspace clustering*: while an overlapping clustering, within a uniquely defined subspace, clusters are not expected to overlap

3.3 K-Means clustering

Also known as centroid-based clustering. In centroid-based clustering clusters are represented by a central vector, which may not necessarily be a member of the data set. When the number of clusters is fixed to k, k-means clustering gives a formal definition as an optimization problem: find the k cluster centers and assign the objects to the nearest cluster center, such that the squared distances from the cluster are minimized.

The optimization problem itself is known to be NP-hard, and thus the common approach is to search only for approximate solutions. A particularly well known approximate method is *Lloyd's algorithm*, often just referred to as "k-means algorithm". It does however only find a local optimum, and is commonly run multiple times with different random initializations. Variations of k-means often include such optimizations as choosing the best of multiple runs, but also restricting the centroids to members of the data set (k-medoids), choosing medians (k-medians clustering), choosing the initial centers less randomly (k-means++) or allowing a fuzzy cluster assignment (fuzzy c-means).

Most k-means-type algorithms require the number of clusters - k - to be specified in advance, which is considered to be one of the biggest drawbacks of these algorithms. Furthermore, the algorithms prefer clusters of approximately similar size, as they will always assign an object to the nearest centroid. This often leads to incorrectly cut borders of clusters (which is not surprising since the algorithm optimizes cluster centers, not cluster borders).

K-means has a number of interesting theoretical properties. First,

it partitions the data space into a structure known as a Voronoi diagram. Second, it is conceptually close to nearest neighbor classification, and as such is popular in machine learning. Third, it can be seen as a variation of model based clustering, and Lloyd's algorithm as a variation of the Expectation-maximization algorithm for this model discussed below.

Centroid-based clustering problems such as *k-means* and *k-medoids* are special cases of the uncapacitated, metric facility location problem, a canonical problem in the operations research and computational geometry communities. In a basic facility location problem (of which there are numerous variants that model more elaborate settings), the task is to find the best warehouse locations to optimally service a given set of consumers. One may view "warehouses" as cluster centroids and "consumer locations" as the data to be clustered. This makes it possible to apply the well-developed algorithmic solutions from the facility location literature to the presently considered centroid-based clustering problem.

3.4 K-Means clustering using Python

Clustering of unlabeled data can be performed with the module `sklearn.cluster`. Each clustering algorithm comes in two variants: a *class,* that implements the `fit` method to learn the clusters on train data, and a *function*, that, given train data, returns an array of integer labels corresponding to the different clusters. For the class, the labels over the training data can be found in the `labels_` attribute.

3.4.1 clustering algorithms in *scikit-learn*

1. K-Means
2. Affinity propagation
3. Mean-shift

4. Spectral clustering
5. Ward hierarchical clustering
6. Agglomerative clustering
7. DBSCAN
8. OPTICS
9. Gaussian mixtures
10. BIRCH

The *KMeans* algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares (see below). This algorithm requires the number of clusters to be specified. It scales well to large number of samples and has been used across a large range of application areas in many different fields. The k-means algorithm divides a set of samples into disjoint clusters , each described by the mean of the samples in the cluster. The means are commonly called the cluster centroids ; note that they are not, in general, points from , although they live in the same space. The K-means algorithm aims to choose centroids that minimize the inertia, or within-cluster sum-of-squares criterion:

$$\sum_{i=0}^{n} \frac{min}{\mu_{j \in c}} (||x_i - \mu_i||^2) \tag{3.1}$$

Inertia can be recognized as a measure of how internally coherent clusters are. It suffers from various drawbacks. [53]. The k-means problem is solved using either *Lloyd s* or *Elkan s* algorithm. **The average complexity is given by *O(k n T)*, where *n* is the number of samples and *T* is the number of iteration.** The worst case complexity is given by $O(n^{k+2/p})$ with n = samples, p = features. In practice, the k-means algorithm is very fast (one of the fastest clustering algorithms available), but it falls in

local minima. That s why it can be useful to restart it several times. If the algorithm stops before fully converging (because of `tol` or `max_iter`), `labels_` and `cluster_centers_` will not be consistent, i.e. the `cluster_centers_` will not be the means of the points in each cluster. Also, the estimator will reassign `labels_` after the last iteration to make `labels_` consistent with predict on the training set.

```
>>> from sklearn.cluster import KMeans
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [10, 2], [10, 4], [10, 0]])
>>> kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
>>> kmeans.labels_
array([1, 1, 1, 0, 0, 0], dtype=int32)
>>> kmeans.predict([[0, 0], [12, 3]])
array([1, 0], dtype=int32)
>>> kmeans.cluster_centers_
array([[10.,  2.],
       [ 1.,  2.]])
```

3.5 Hierarchical clustering

Hierarchical clustering is a method of cluster analysis which seeks to build a hierarchy of clusters. Strategies for hierarchical clustering generally fall into two types:

1. *Agglomerative*: This is a "bottom-up" approach: each observation starts in its own cluster, and pairs of clusters are merged as one moves up the hierarchy.

2. *Divisive*: This is a "top-down" approach: all observations start in one cluster, and splits are performed recursively as one moves down the hierarchy.

In general, the merges and splits are determined in a greedy manner. The results of hierarchical clustering are usually presented in a *dendrogram*. [54] [55]

The standard algorithm for hierarchical agglomerative clustering (HAC) has a time complexity of $\mathcal{O}(n^3)$ and requires $\Omega(n^2)$ memory, which makes it too slow for even medium data sets. However, for some special cases, optimal efficient agglomerative methods (of complexity $\mathcal{O}(n^2)$) are known: SLINK for single-linkage and CLINK for complete-linkage clustering. With a heap, the runtime of the general case can be reduced to $\mathcal{O}(n^2 \log n)$, an improvement on the aforementioned bound of $\mathcal{O}(n^3)$, at the cost of further increasing the memory requirements. In many cases, the memory overheads of this approach are too large to make it practically usable. Except for the special case of single-linkage, none of the algorithms (except exhaustive search in $\mathcal{O}(2^n)$ can be guaranteed to find the optimum solution. Divisive clustering with an exhaustive search is $\mathcal{O}(2^n)$, but it is common to use faster heuristics to choose splits, such as k-means. [56] [57]

In order to decide which clusters should be combined (for agglomerative), or where a cluster should be split (for divisive), a measure of dissimilarity between sets of observations is required. In most methods of hierarchical clustering, this is achieved by use of an appropriate metric (a measure of distance between pairs of observations), and a linkage criterion which specifies the dissimilarity of sets as a function of the pairwise distances of observations in the sets.

The choice of an appropriate metric will influence the shape of the clusters, as some elements may be relatively closer to one another under one metric than another. For example, in two dimensions, under the Manhattan distance metric, the distance between the origin (0,0) and (0.5, 0.5) is the same as the distance between the origin and (0, 1), while under the Euclidean distance metric the latter is strictly greater. Some commonly used metrics for hierarchical clustering are:

Names	Formula
Euclidean distance	$\|a-b\|_2 = \sqrt{\sum_i (a_i - b_i)^2}$
Squared Euclidean distance	$\|a-b\|_2^2 = \sum_i (a_i - b_i)^2$
Manhattan distance	$\|a-b\|_1 = \sum_i \|a_i - b_i\|$
Maximum distance	$\|a-b\|_\infty = \max_i \|a_i - b_i\|$
Mahalanobis distance	$\sqrt{(a-b)^\top S^{-1}(a-b)}$ where S is the Covariance matrix

Table 3.1: Distance metric for HAC.

3.5.1 Python practice

Hierarchical clustering is a general family of clustering algorithms that build nested clusters by merging or splitting them successively. This hierarchy of clusters is represented as a tree (or dendrogram). The root of the tree is the unique cluster that gathers all the samples, the leaves being the clusters with only one sample. See the Wikipedia page for more details. The AgglomerativeClustering object performs a hierarchical clustering using a bottom up approach: each observation starts in its own cluster, and clusters are successively merged together. The linkage criteria determines the metric used for the merge strategy:

1. Ward minimizes the sum of squared differences within all clusters. It is a variance-minimizing approach and in this sense is similar to the k-means objective function but tackled with an agglomerative hierarchical approach.

2. Maximum or complete linkage minimizes the maximum distance between observations of pairs of clusters.

3. Average linkage minimizes the average of the distances between all observations of pairs of clusters.

4. Single linkage minimizes the distance between the closest observations of pairs of clusters.

AgglomerativeClustering can also scale to large number of sam-

ples when it is used jointly with a connectivity matrix, but is computationally expensive when no connectivity constraints are added between samples: it considers at each step all the possible merges.

```
>>> from sklearn.cluster import AgglomerativeClustering
>>> import numpy as np
>>> X = np.array([[1, 2], [1, 4], [1, 0],
...               [4, 2], [4, 4], [4, 0]])
>>> clustering = AgglomerativeClustering().fit(X)
>>> clustering
AgglomerativeClustering()
>>> clustering.labels_
array([1, 1, 1, 0, 0, 0])
```

3.5.2 Plotting Dendrogram

This example plots the corresponding dendrogram of a hierarchical clustering using AgglomerativeClustering and the dendrogram method available in scipy.

```
import numpy as np

from matplotlib import pyplot as plt
from scipy.cluster.hierarchy import dendrogram
from sklearn.datasets import load_iris
from sklearn.cluster import AgglomerativeClustering


def plot_dendrogram(model, **kwargs):
    # Create linkage matrix and then plot the dendrogram

    # create the counts of samples under each node
    counts = np.zeros(model.children_.shape[0])
    n_samples = len(model.labels_)
    for i, merge in enumerate(model.children_):
        current_count = 0
        for child_idx in merge:
            if child_idx < n_samples:
                current_count += 1  # leaf node
            else:
                current_count += counts[child_idx -
                                        ↪ n_samples]
```

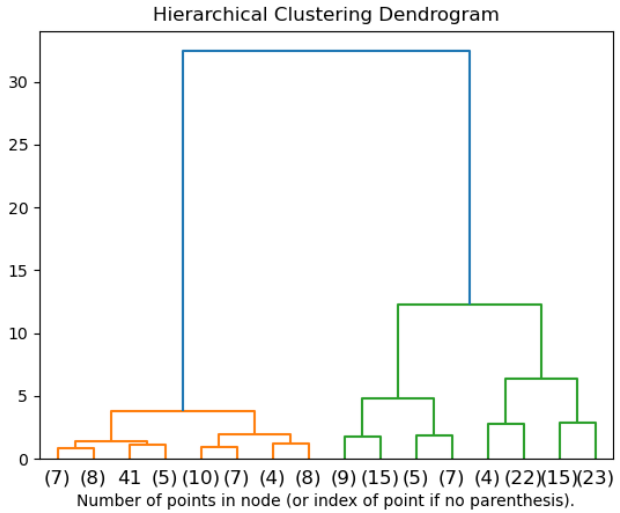

Figure 3.1: Agglomerative dendrogram

```
        counts[i] = current_count

    linkage_matrix = np.column_stack(
        [model.children_, model.distances_, counts]
    ).astype(float)

    # Plot the corresponding dendrogram
    dendrogram(linkage_matrix, **kwargs)


iris = load_iris()
X = iris.data

# setting distance_threshold=0 ensures we compute the
                              ↪ full tree.
model = AgglomerativeClustering(distance_threshold=0,
                              ↪ n_clusters=None)
```

```
model = model.fit(X)
plt.title("Hierarchical Clustering Dendrogram")
# plot the top three levels of the dendrogram
plot_dendrogram(model, truncate_mode="level", p=3)
plt.xlabel("Number of points in node (or index of point
                            ↪ if no parenthesis).")
plt.show()
```

3.6 Anomaly Detection

3.7 Expectation Maximization (EM) algorithm

3.8 Reinforcement Learning

Notes

[47]Hinton, Geoffrey; Sejnowski, Terrence (1999). Unsupervised Learning: Foundations of Neural Computation. MIT Press. ISBN 978-0262581684.

[48]Driver and Kroeber (1932). "Quantitative Expression of Cultural Relationships". University of California Publications in American Archaeology and Ethnology. Quantitative Expression of Cultural Relationships: 211 256 via http://dpg.lib.berkeley.edu.

[49]Zubin, Joseph (1938). "A technique for measuring like-mindedness". The Journal of Abnormal and Social Psychology. 33 (4): 508 516. doi:10.1037/h0055441. ISSN 0096-851X.

[50]Tryon, Robert C. (1939). Cluster Analysis: Correlation Profile and Orthometric (factor) Analysis for the Isolation of Unities in Mind and Personality. Edwards Brothers.

[51]Cattell, R. B. (1943). "The description of personality: Basic traits resolved into clusters". Journal of Abnormal and Social Psychology. 38 (4): 476 506. doi:10.1037/h0054116.

[52]Estivill-Castro, Vladimir (20 June 2002). "Why so many clustering algorithms A Position Paper". ACM SIGKDD Explorations Newsletter. 4 (1): 65 75. doi:10.1145/568574.568575. S2CID 7329935.

[53]See the KMeans documentation at https://scikit-learn.org/stable/modules/clustering.html#k-means

[54]Maimon, Oded; Rokach, Lior (2006). "Clustering methods". Data Mining and Knowledge Discovery Handbook. Springer. pp. 321 352. ISBN 978-0-387-25465-4.

[55]Nielsen, Frank (2016). "8. Hierarchical Clustering". Introduction to HPC with MPI for Data Science. Springer. pp. 195 211. ISBN 978-3-319-21903-5.

[56]R. Sibson (1973). "SLINK: an optimally efficient algorithm for the single-link cluster method" (PDF). The Computer Journal. British Computer Society. 16 (1): 30 34. doi:10.1093/comjnl/16.1.30.

[57]D. Defays (1977). "An efficient algorithm for a complete-link method". The Computer Journal. British Computer Society. 20 (4): 364 6. doi:10.1093/comjnl/20.4.364.

Chapter 4

Artificial Neural Networks

4.1 Neural Networks

A neural network is a network or circuit of neurons, or in a modern sense, an artificial neural network, composed of artificial neurons or nodes. [58] Thus a neural network is either a biological neural network, made up of biological neurons, or an artificial neural network, for solving artificial intelligence (AI) problems. The connections of the biological neuron are modeled as *weights*. A positive weight reflects an excited connection, while negative values mean inhibitory connections. All inputs are modified by a weight and summed. This activity is referred to as a *linear combination*. Finally, an activation function controls the amplitude of the output. For example, an acceptable range of output is usually between 0 and 1, or it could be ?1 and 1. These artificial networks may be used for predictive modeling, adaptive control and applications where they can be trained

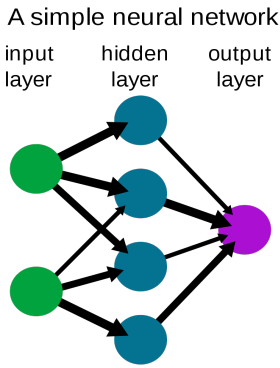

Figure 4.1: ANN model

via a dataset. Self-learning resulting from experience can occur within networks, which can derive conclusions from a complex and seemingly unrelated set of information.

A neural network (NN), in the case of artificial neurons called artificial neural network (ANN) or simulated neural network (SNN), is an interconnected group of natural or artificial neurons that uses a mathematical or computational model for information processing based on a connectionistic approach to computation. In most cases an ANN is an adaptive system that changes its structure based on external or internal information that flows through the network. In more practical terms neural networks are *non-linear* statistical data modeling or decision making tools. They can be used to model complex relationships between inputs and outputs or to find patterns in data.

An artificial neural network involves a network of simple processing elements (artificial neurons) which can exhibit complex global behavior, determined by the connections between the processing elements and element parameters. Artificial neurons were first proposed in 1943 by Warren McCulloch, a neurophys-

iologist, and Walter Pitts, a logician, who first collaborated at the University of Chicago. [59]

One classical type of artificial neural network is the recurrent *Hopfield network*. [60] The concept of a neural network appears to have first been proposed by Alan Turing in his 1948 paper Intelligent Machinery in which he called them "B-type unorganised machines". [61]

The utility of artificial neural network models lies in the fact that they can be used to infer a function from observations and also to use it. Unsupervised neural networks can also be used to learn representations of the input that capture the salient characteristics of the input distribution, e.g., see the Boltzmann machine (1983), and more recently, deep learning algorithms, which can implicitly learn the distribution function of the observed data. Learning in neural networks is particularly useful in applications where the complexity of the data or task makes the design of such functions by hand impractical.

4.2 Overview & concept

A biological neural network is composed of a groups of chemically connected or functionally associated neurons. A single neuron may be connected to many other neurons and the total number of neurons and connections in a network may be extensive. Connections, called *synapses*, are usually formed from *axons* to *dendrites*. Apart from the electrical signaling, there are other forms of signaling that arise from neurotransmitter diffusion.

Artificial intelligence, cognitive modeling, and neural networks are information processing paradigms inspired by the way biological neural systems process data. Artificial intelligence and cognitive modeling try to simulate some properties of biological neural networks. In the artificial intelligence field, artificial neural networks have been applied successfully to speech recog-

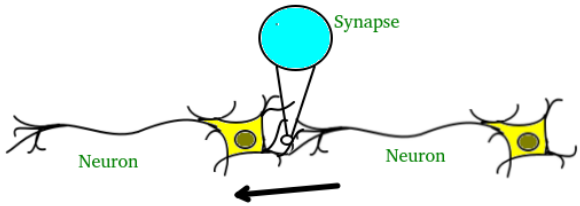

Figure 4.2: Neurons

nition, image analysis and adaptive control, in order to construct software agents (in computer and video games) or autonomous robots.

Historically, digital computers evolved from the *von Neumann model*, and operate via the execution of explicit instructions via access to memory by a number of processors. On the other hand, the origins of neural networks are based on efforts to model information processing in biological systems. *Unlike the von Neumann model, neural network computing does not separate memory and processing.* Neural network theory has served both to better identify how the neurons in the brain function and to provide the basis for efforts to create artificial intelligence.

4.2.1 History

The preliminary theoretical base for contemporary neural networks was independently proposed by Alexander Bain (1873) and William James (1890). [62] [63] In their work, both thoughts and body activity resulted from interactions among neurons within the brain. For *Bain*, every activity led to the firing of a certain set of neurons. When activities were repeated, the connections between those neurons strengthened. According to his theory, this repetition was what led to the formation of *memory*. The general scientific community at the time was skeptical

of Bain's theory because it required what appeared to be an inordinate number of neural connections within the brain. It is now apparent that the brain is exceedingly complex and that the same brain "wiring" can handle multiple problems and inputs. *James's* theory was similar to Bain's, however, he suggested that memories and actions resulted from electrical currents flowing among the neurons in the brain. His model, by focusing on the flow of electrical currents, did not require individual neural connections for each memory or action.

C. S. Sherrington (1898) conducted experiments to test James's theory. He ran electrical currents down the spinal cords of rats. However, instead of demonstrating an increase in electrical current as projected by James, Sherrington found that the electrical current strength decreased as the testing continued over time. Importantly, this work led to the discovery of the concept of habituation. [64]

McCulloch and *Pitts* (1943) created a computational model for neural networks based on mathematics and algorithms. They called this model threshold logic. The model paved the way for neural network research to split into two distinct approaches. One approach focused on biological processes in the brain and the other focused on the application of neural networks to artificial intelligence. [65]

In the late 1940s psychologist Donald Hebb created a hypothesis of learning based on the mechanism of neural plasticity that is now known as *Hebbian learning*. Hebbian learning is considered to be a typical unsupervised learning rule and its later variants were early models for long term potentiation. These ideas started being applied to computational models in 1948 with Turing's B-type machines. *Hebb, Donald (1949). The Organization of Behavior. New York: Wiley.* Farley and Clark (1954) first used computational machines, then called calculators, to simulate a Hebbian network at MIT. Other neural network computational machines were created by Rochester, Holland, Habit, and

Duda (1956). [66] [67]

Rosenblatt (1958) created the *perceptron*, an algorithm for pattern recognition based on a two-layer learning computer network using simple addition and subtraction. With mathematical notation, Rosenblatt also described circuitry not in the basic perceptron, such as the exclusive-or circuit, a circuit whose mathematical computation could not be processed until after the backpropagation algorithm was created by Werbos (1975). [68] [69]

Neural network research stagnated after the publication of machine learning research by *Marvin Minsky* and *Seymour Papert* (1969). They discovered two key issues with the computational machines that processed neural networks. The first issue was that single-layer neural networks were incapable of processing the exclusive-or circuit. The second significant issue was that computers were not sophisticated enough to effectively handle the long run time required by large neural networks. Neural network research slowed until computers achieved greater processing power. Also key in later advances was the backpropagation algorithm which effectively solved the exclusive-or problem (Werbos 1975). [70] The parallel distributed processing of the mid-1980s became popular under the name *connectionism*. The text by Rumelhart and McClelland (1986) provided a full exposition on the use of connectionism in computers to simulate neural processes. [71]

4.3 Applications

Neural networks can be used in different fields. The tasks to which artificial neural networks are applied tend to fall within the following broad categories:

- Function approximation, or regression analysis, including time series prediction and modeling.
- Classification, including pattern and sequence recognition,

novelty detection and sequential decision making.

- Data processing, including filtering, clustering, blind signal separation and compression.

Application areas of ANNs include nonlinear system identification and control (vehicle control, process control), game-playing and decision making (backgammon, chess, racing), pattern recognition (radar systems, face identification, object recognition), sequence recognition (gesture, speech, handwritten text recognition), medical diagnosis, financial applications, data mining (or knowledge discovery in databases, "KDD"), visualization and e-mail spam filtering. For example, it is possible to create a semantic profile of user's interests emerging from pictures trained for object recognition.

4.4 Hebbian Learning - A generic model for ANN

Also known as *Hebbian theory* is a neuroscientific theory claiming that an increase in synaptic efficacy arises from a presynaptic cell's repeated and persistent stimulation of a postsynaptic cell. It is an attempt to explain synaptic plasticity, the adaptation of brain neurons during the learning process. It was introduced by Donald Hebb in his 1949 book The Organization of Behavior. [72] The theory is also called Hebb's rule, Hebb's postulate, and cell assembly theory.

The theory is often summarized as "Cells that fire together wire together." However, Hebb emphasized that cell A needs to "take part in firing" cell B, and such causality can occur only if cell A fires just before, not at the same time as, cell B. This aspect of causation in Hebb's work foreshadowed what is now known about spike-timing-dependent plasticity, which requires temporal precedence.

The theory attempts to explain associative or Hebbian learning, in which simultaneous activation of cells leads to pronounced increases in synaptic strength between those cells. It also provides a biological basis for errorless learning methods for education and memory rehabilitation. In the study of neural networks in cognitive function, it is often regarded as the neuronal basis of unsupervised learning.

4.4.1 Weights

From the point of view of artificial neurons and artificial neural networks, Hebb's principle can be described as a method of determining how to alter the weights between model neurons. The weight between two neurons increases if the two neurons activate simultaneously, and reduces if they activate separately. Nodes that tend to be either both positive or both negative at the same time have strong positive weights, while those that tend to be opposite have strong negative weights.

The following is a formulaic description of Hebbian learning: (many other descriptions are possible)

$$w_{ij} = x_i x_j$$

where w_ij is the weight of the connection from neuron j to neuron i and x_i the input for neuron i. Note that this is pattern learning (weights updated after every training example). In a Hopfield network, connections w_ij are set to zero if $i = j$ (no reflexive connections allowed). With binary neurons (activations either 0 or 1), connections would be set to 1 if the connected neurons have the same activation for a pattern. When several training patterns are used the expression becomes an average of individual ones:

$$w_{ij} = \frac{1}{p} \sum_{k=1}^{p} x_i^k x_j^k = \langle x_i x_j \rangle,$$

where w_ij is the weight of the connection from neuron j to neuron i, p is the number of training patterns, x_i^k the k^{th}

input for neuron i and <> is the average over all training patterns. This is learning by epoch (weights updated after all the training examples are presented), being last term applicable to both discrete and continuous training sets. Again, in a Hopfield network, connections w_ij are set to zero if $i = j$ (no reflexive connections).

A variation of Hebbian learning that takes into account phenomena such as blocking and many other neural learning phenomena is the mathematical model of Harry Klopf. Klopf's model reproduces a great many biological phenomena, and is also simple to implement.

4.4.2 Relationship to unsupervised learning

Because of the simple nature of Hebbian learning, based only on the coincidence of pre- and post-synaptic activity, it may not be intuitively clear why this form of plasticity leads to meaningful learning. However, it can be shown that Hebbian plasticity does pick up the statistical properties of the input in a way that can be categorized as unsupervised learning.

This can be mathematically shown in a simplified example. Let us work under the simplifying assumption of a single rate-based neuron of rate $y(t)$, whose inputs have rates $x\_1(t)...x\_N(t)$. The response of the neuron $y(t)$ is usually described as a linear combination of its input, $\sum\_iw\_ix\_i$, followed by a response function f:

$$y = f\left(\sum\_i = 1^N w_i x_i\right).$$

As defined in the previous sections, Hebbian plasticity describes the evolution in time of the synaptic weight w:

$$\frac{dw_i}{dt} = \eta x_i y.$$

Assuming, for simplicity, an identity response function $f(a) = a$, we can write

$$\frac{dw_i}{dt} = \eta x_i \sum_{j=1}^{N} w_j x_j$$

or in matrix form:

$$\frac{d\mathbf{w}}{dt} = \eta \mathbf{x}\mathbf{x}^T \mathbf{w}$$

As in previous chapter, if training by epoch is done an average $< . >$ over discrete or continuous (time) training set of $\mathbf{x}$ can be done:

$$\frac{d\mathbf{w}}{dt} = \langle \eta \mathbf{x}\mathbf{x}^T \mathbf{w} \rangle = \eta \langle \mathbf{x}\mathbf{x}^T \rangle \mathbf{w} = \eta C \mathbf{w}.$$

where $C = \langle \mathbf{x}\mathbf{x}^T \rangle$ is the correlation matrix of the input under the additional assumption that $\langle \mathbf{x} \rangle = 0$ (i.e. the average of the inputs is zero). This is a system of N coupled linear differential equations. Since C is symmetric, it is also diagonalizable, and the solution can be found, by working in its eigenvectors basis, to be of the form

$$\mathbf{w}(t) = k_1 e^{\eta \alpha_1 t} \mathbf{c}_1 + k_2 e^{\eta \alpha_2 t} \mathbf{c}_2 + ... + k_N e^{\eta \alpha_N t} \mathbf{c}_N$$

where k_i are arbitrary constants, $\mathbf{c}_i$ are the eigenvectors of C and α_i their corresponding eigenvalues. Since a correlation matrix is always a positive-definite matrix, the eigenvalues are all positive, and one can easily see how the above solution is always exponentially divergent in time. This is an intrinsic problem due to this version of Hebb's rule being unstable, as in any network with a dominant signal the synaptic weights will increase or decrease exponentially. Intuitively, this is because whenever the presynaptic neuron excites the postsynaptic neuron, the weight between them is reinforced, causing an even stronger excitation in the future, and so forth, in a self-reinforcing way. One may think a solution is to limit the firing rate of the postsynaptic neuron by adding a non-linear, saturating response function f, but in fact, it can be shown that for any neuron model, Hebb's rule is unstable. Therefore, network models of neurons usually employ other learning theories such as BCM theory, Oja's rule, or the generalized Hebbian algorithm. Regardless, even for the unstable solution above, one can see that, when sufficient time

has passed, one of the terms dominates over the others, and

$$\mathbf{w}(t) \approx e^{\eta\alpha^* t}\mathbf{c}^*$$

where α^* is the largest eigenvalue of C. At this time, the postsynaptic neuron performs the following operation:

$$y \approx e^{\eta\alpha^* t}\mathbf{c}^*\mathbf{x}$$

Because, again, $\mathbf{c}^*$ is the eigenvector corresponding to the largest eigenvalue of the correlation matrix between the x_is, this corresponds exactly to computing the first principal component of the input.

This mechanism can be extended to performing a full PCA (principal component analysis) of the input by adding further postsynaptic neurons, provided the postsynaptic neurons are prevented from all picking up the same principal component, for example by adding lateral inhibition in the postsynaptic layer. We have thus connected Hebbian learning to PCA, which is an elementary form of unsupervised learning, in the sense that the network can pick up useful statistical aspects of the input, and "describe" them in a distilled way in its output.

4.5 Python Practice

4.5.1 Neural Networks

An Artificial Neural Network (ANN) is an information processing paradigm that is inspired the brain. ANNs, like people, learn by example. An ANN is configured for a specific application, such as pattern recognition or data classification, through a learning process. Learning largely involves adjustments to the synaptic connections that exist between the neurons.

The brain consists of hundreds of billion of cells called neurons. These neurons are connected together by synapses which are nothing but the connections across which a neuron can send an

Input 1	Input 2	Input 3	Output
0	1	1	1
1	0	0	0
1	0	1	1

Fig 2: Training Examples

Now we want to predict the output the following set of inputs:

1	0	1	?

Fig 3: Test Example

Figure 4.3: Input sets for NN problem

impulse to another neuron. When a neuron sends an excitatory signal to another neuron, then this signal will be added to all of the other inputs of that neuron. If it exceeds a given threshold then it will cause the target neuron to fire an action signal forward — this is how the thinking process works internally.

In Computer Science, we model this process by creating "networks" on a computer using matrices. These networks can be understood as abstraction of neurons without all the biological complexities taken into account. To keep things simple, we will just model a simple NN, with two layers capable of solving linear classification problem.

Let's say we have a problem where we want to predict output given a set of inputs and outputs as training example like so

Note that the output is directly related to third column i.e. the values of input 3 is what the output is in every training example in fig. 2. So for the test example output value should be 1. The

training process consists of the following steps:

1. Forward Propagation: Take the inputs, multiply by the weights (just use random numbers as weights) Let $Y = W_i I_i = W_1 I_1 + W_2 I_2 + W_3 I_3$. Pass the result through a sigmoid formula to calculate the neuron's output. The Sigmoid function is used to normalise the result between 0 and 1: $1/1 + e^{-y}$
2. Back Propagation: Calculate the error i.e the difference between the actual output and the expected output. Depending on the error, adjust the weights by multiplying the error with the input and again with the gradient of the Sigmoid curve: $Weight+ = (Error \times Input) \times [Output \times (1 - Output)]$,here $Output \times (1 - Output)$ is derivative of sigmoid curve.

Repeat the whole process for a few thousands iterations. Let's code up the whole process in Python. We'll be using Numpy library to help us with all the calculations on matrices easily. You'd need to install numpy library on your system to run the code

```
from joblib.numpy_pickle_utils import xrange
from numpy import *


class NeuralNet(object):
    def __init__(self):
        # Generate random numbers
        random.seed(1)

        # Assign random weights to a 3 x 1
            ↪ matrix,
        self.synaptic_weights = 2 * random.
            ↪ random((3, 1)) - 1

    # The Sigmoid function
```

```
    def __sigmoid(self, x):
        return 1 / (1 + exp(-x))

    # The derivative of the Sigmoid function.
    # This is the gradient of the Sigmoid
        ↪ curve.
    def __sigmoid_derivative(self, x):
        return x * (1 - x)

    # Train the neural network and adjust the
        ↪ weights each time.
    def train(self, inputs, outputs,
        ↪ training_iterations):
        for iteration in xrange(
            ↪ training_iterations):
            # Pass the training set through
                ↪ the network.
            output = self.learn(inputs)

            # Calculate the error
            error = outputs - output

            # Adjust the weights by a factor
            factor = dot(inputs.T, error *
                ↪ self.__sigmoid_derivative(
                ↪ output))
            self.synaptic_weights += factor

        # The neural network thinks.

    def learn(self, inputs):
        return self.__sigmoid(dot(inputs, self
            ↪ .synaptic_weights))


if __name__ == "__main__":
```

```
# Initialize
neural_network = NeuralNet()

# The training set.
inputs = array([[0, 1, 1], [1, 0, 0], [1,
    ↪ 0, 1]])
outputs = array([[1, 0, 1]]).T

# Train the neural network
neural_network.train(inputs, outputs,
    ↪ 10000)

# Test the neural network with a test
    ↪ example.
print(neural_network.learn(array([1, 0,
    ↪ 1])))
```

After 10 iterations our neural network predicts the value to be 0.65980921. It looks not good as the answer should really be 1. If we increase the number of iterations to 100, we get 0.87680541. Our network is getting smarter! Subsequently, for 10000 iterations we get 0.9897704 which is pretty close and indeed a satisfactory output.

4.5.2 Feedforward networks

A feedforward neural network is an artificial neural network wherein connections between the nodes do not form a cycle. [73] As such, it is different from its descendant: recurrent neural networks. The feedforward neural network was the first and simplest type of artificial neural network devised. [74] In this network, the information moves in only one direction—forward—from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.

Single-layer perceptron

The simplest kind of neural network is a single-layer perceptron network, which consists of a single layer of output nodes; the inputs are fed directly to the outputs via a series of weights. The sum of the products of the weights and the inputs is calculated in each node, and if the value is above some threshold (typically 0) the neuron fires and takes the activated value (typically 1); otherwise it takes the deactivated value (typically -1). Neurons with this kind of activation function are also called artificial neurons or linear threshold units. In the literature the term perceptron often refers to networks consisting of just one of these units. A similar neuron was described by Warren McCulloch and Walter Pitts in the 1940s.

A perceptron can be created using any values for the activated and deactivated states as long as the threshold value lies between the two. Perceptrons can be trained by a simple learning algorithm that is usually called the delta rule. It calculates the errors between calculated output and sample output data, and uses this to create an adjustment to the weights, thus implementing a form of gradient descent.

Single-layer perceptrons are only capable of learning linearly separable patterns; in 1969 in a famous monograph entitled Perceptrons, Marvin Minsky and Seymour Papert showed that it was impossible for a single-layer perceptron network to learn an XOR function (nonetheless, it was known that multi-layer perceptrons are capable of producing any possible boolean function).

Although a single threshold unit is quite limited in its computational power, it has been shown that networks of parallel threshold units can approximate any continuous function from a compact interval of the real numbers into the interval [-1,1]. This result can be found in Peter Auer, Harald Burgsteiner and Wolfgang Maass "A learning rule for very simple universal approximators consisting of a single layer of perceptrons". [75]

A single-layer neural network can compute a continuous output instead of a step function. A common choice is the so-called logistic function:

$$f(x) = \frac{1}{1+e^{-x}}$$

With this choice, the single-layer network is identical to the logistic regression model, widely used in statistical modeling. The logistic function is one of the family of functions called sigmoid functions because their S-shaped graphs resemble the final-letter lower case of the Greek letter Sigma. It has a continuous derivative, which allows it to be used in backpropagation. This function is also preferred because its derivative is easily calculated:

$$f'(x) = f(x)(1 - f(x))$$

(The fact that f satisfies the differential equation above can easily be shown by applying the chain rule.)

If single-layer neural network activation function is modulo 1, then this network can solve XOR problem with a single neuron.

$$f(x) = x \mod 1$$
$$f'(x) = 1$$

Multi-layer perceptron

This class of networks consists of multiple layers of computational units, usually interconnected in a feed-forward way. Each neuron in one layer has directed connections to the neurons of the subsequent layer. In many applications the units of these networks apply a sigmoid function as an activation function. However sigmoidal activation functions have very small derivative values outside a small range and do not work well in deep neural networks due to the vanishing gradient problem. Alternatives to sigmoidal activation functions that alleviate the vanishing gradient problems and allow deep networks to be trained have been proposed.

The universal approximation theorem for neural networks states that every continuous function that maps intervals of real numbers to some output interval of real numbers can be approximated arbitrarily closely by a multi-layer perceptron with just one hidden layer. This result holds for a wide range of activation functions, e.g. for the sigmoidal functions.

Multi-layer networks use a variety of learning techniques, the most popular being back-propagation. Here, the output values are compared with the correct answer to compute the value of some predefined error-function. By various techniques, the error is then fed back through the network. Using this information, the algorithm adjusts the weights of each connection in order to reduce the value of the error function by some small amount. After repeating this process for a sufficiently large number of training cycles, the network will usually converge to some state where the error of the calculations is small. In this case, one would say that the network has learned a certain target function. To adjust weights properly, one applies a general method for non-linear optimization that is called gradient descent. For this, the network calculates the derivative of the error function with respect to the network weights, and changes the weights such that the error decreases (thus going downhill on the surface of the error function). For this reason, back-propagation can only be applied on networks with differentiable activation functions.

In general, the problem of teaching a network to perform well, even on samples that were not used as training samples, is a quite subtle issue that requires additional techniques. This is especially important for cases where only very limited numbers of training samples are available. The danger is that the network overfits the training data and fails to capture the true statistical process generating the data. Computational learning theory is concerned with training classifiers on a limited amount of data. In the context of neural networks a simple heuristic, called early stopping, often ensures that the network will generalize well to examples not in the training set.

Other typical problems of the back-propagation algorithm are the speed of convergence and the possibility of ending up in a local minimum of the error function. Today, there are practical methods that make back-propagation in multi-layer perceptrons the tool of choice for many machine learning tasks. One also can use a series of independent neural networks moderated by some intermediary, a similar behavior that happens in brain. These neurons can perform separably and handle a large task, and the results can be finally combined.

Other feedforward networks

More generally, any directed acyclic graph may be used for a feedforward network, with some nodes (with no parents) designated as inputs, and some nodes (with no children) designated as outputs. These can be viewed as multilayer networks where some edges skip layers, either counting layers backwards from the outputs or forwards from the inputs. Various activation functions can be used, and there can be relations between weights, as in convolutional neural networks. Examples of other feedforward networks include radial basis function networks, which use a different activation function. Sometimes multi-layer perceptron is used loosely to refer to any feedforward neural network, while in other cases it is restricted to specific ones (e.g., with specific activation functions, or with fully connected layers, or trained by the perceptron algorithm).

4.5.3 Python practice

The goal of neural networks in general is to approximate some function f. The Universal Approximation Theorem says neural networks have the capacity to accomplish this for a large class of functions. In this case, we seek to approximate the function that generates a binary classification problem. We can model this with the binary random variable $Y|X : \omega \to \{0, 1\}$ with conditional distribution

$\mathrm{p(x)} = \mathrm{P(Y = y \mid X = x)} = \mathrm{P(Y = 1 \mid X = x)}^{y}(1 - P(Y = 1|X = x))^{1-y}$

which we seek to approximate. Instead of approximating this directly, we divide the function up and approximate each value y takes. Since this is binary classification, we only need to approximate

$\mathrm{P(Y = 1 \mid X = x)} : \mathrm{R}^2 \to [0, 1].$

Since this is a probability function, we will use relative entropy as a measure of distance, which is typically called a loss function. In order to approximate the relative entropy, we sample i.i.d.

$(\mathrm{y}_i, X_i)_{i=1}^{n}$

with

$$\begin{aligned} \mathrm{D(p \mid\mid)} &= \textstyle\int_{R^2} \log\left(\frac{p}{\hat{p}}\right) pdm \\ &\approx \textstyle\sum_{i=1}^{n} \log\left(\frac{p(X_i)}{\hat{p}(X_i)}\right) p(X_i) \\ &\approx \textstyle\sum_{i=1}^{n} \log\left(\frac{p(X_i)}{\hat{p}(X_i)}\right) \frac{1}{n} \end{aligned}$$

where m is Lebesgue measure. Since p is unknown, we minimize the estimated loss,

$$\mathrm{(p \mid\mid)} = -\textstyle\sum_{i=1}^{n} \log(\hat{p}(X_i))\frac{1}{n}$$
$$= -1\frac{}{n\sum_{i=1}^{n} y_i \log(\hat{p}(X_i)) + (1-y_i)\log(1-\hat{p}(X_i)).}$$

We define the single hidden layer neural network. Let W denote the weights and b be the intercept or bias term. We use sigmoid activations denoted by σ, but *Relu* is generally preferred. The hidden layer

$\mathrm{H(X)} = \sigma\left(XW_h + b_h\right)$

and output layer

$\mathrm{O(X)} = \sigma\left(h(X)W_o + b_o\right)$

The neural network is fit using gradient descent with the gradients computed using algorithmic differentiation. These are implemented naively for simplicity and are not using advanced algorithms as would be in practice. We generate data using Sklearn's *make_circles* function, split it into a training and test set, and plot with Matplotlib.

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import make_circles

n_samples = 1000
n_features = 2
n_ouputs = 1
X, y = make_circles(n_samples = n_samples, factor = .01,
                                ↪ noise = .2)

n_TRAIN = int(.75 * n_samples)
X_train = X[0:n_TRAIN, :]
y_train = y[0:n_TRAIN]
X_test = X[n_TRAIN:n_samples, :]
y_test = y[n_TRAIN:n_samples]

fig = plt.figure(figsize=(8, 8))
plt.scatter(X[:,0], X[:,1], c = y)
plt.xlabel("X1")
plt.ylabel("X2")
plt.savefig('nn_plot.pdf', bbox_inches='tight')
```

The neural network implementation

```
class NN():
  def __init__(self, n_samples, n_features, n_outputs,
                                  ↪ n_hidden = 1):
    self.n_samples = n_samples
    self.n_features = n_features
    self.n_hidden = n_hidden
    self.n_outputs = n_outputs

    self.W_h = np.random.randn(n_features, n_hidden)
    self.b_h = .01 + np.zeros((1, n_hidden))
    self.W_o = np.random.randn(n_hidden, n_outputs)
    self.b_o = .01 + np.zeros((1, n_outputs))
```

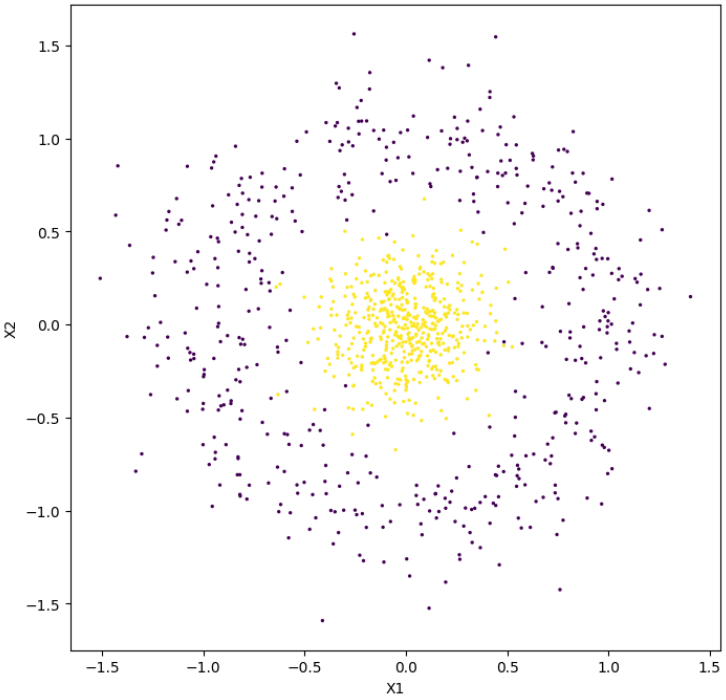

Figure 4.4: Circular data

```
def sigmoid(self, x):
  return 1/(1 + np.exp(-x))

def loss(self, y, p_pred):
  return -1/y.shape[0] * (np.sum(y * np.log(p_pred) + (
                                 ↪ 1 - y) * (np.log(1 -
                                 ↪ p_pred))))

def predict(self, X):
  return np.squeeze(np.round(self.forward_prop(X)["O"])
                                 ↪ )
```

```
def forward_prop(self, X):
  # Hidden layer
  A_h = X @ self.W_h + self.b_h
  H = self.sigmoid(A_h)

  # Output layer
  A_o = H @ self.W_o + self.b_o
  O = self.sigmoid(A_o)
  return {
    "A_h": A_h,
    "H": H,
    "A_o": A_o,
    "O": O
  }

# This is not a true implmentation of backprop
def backward_prop(self, X, y_, forward):
  one_n = np.ones(self.n_samples)
  y = (y_[np.newaxis]).T # convert to column vector

  dA_o = (y - forward["O"])
  dL_dW_o = 1/self.n_samples * forward["H"].T @ dA_o
  dL_db_o = 1/self.n_samples * one_n.T @ dA_o

  dA_h = (dA_o @ self.W_o.T) * (self.sigmoid(forward["
                                ↪ A_h"]) * (1 - self.
                                ↪ sigmoid(forward["A_h"]
                                ↪ )))
  dL_dW_h = 1/self.n_samples * X.T @ dA_h
  dL_db_h = 1/self.n_samples * one_n.T @ dA_h

  return {
    "dL_dW_h": dL_dW_h,
    "dL_db_h": dL_db_h,
    "dL_dW_o": dL_dW_o,
    "dL_db_o": dL_db_o
  }

def train(self, X, y, learning_rate = .5, max_iter =
                              ↪ 1001):
  for i in range(0, max_iter):
    forward_prop_dict = self.forward_prop(X)
    G = self.backward_prop(X, y, forward_prop_dict)
```

```
        # Gradient step
        self.W_h = self.W_h + learning_rate * G["dL_dW_h"]
        self.b_h = self.b_h + learning_rate * G["dL_db_h"]

        self.W_o = self.W_o + learning_rate * G["dL_dW_o"]
        self.b_o = self.b_o + learning_rate * G["dL_db_o"]

        if i % 100 == 0:
          print(f"Iteration: {i}, Training Loss: {self.loss
                                        ↪ (y, np.squeeze(
                                        ↪ forward_prop_dict
                                        ↪ ['O']))}")
```

We use 10 hidden units in the hidden layer and report the 0-1 accuracy on both the training and test sets.

```
nn = NN(n_samples = n_TRAIN, n_features = n_features,
                                ↪ n_outputs = n_ouputs,
                                ↪ n_hidden = 10)
nn.train(X_train, y_train)

print("Train accuracy:", 1/X_train.shape[0] * np.sum(nn.
                                ↪ predict(X_train) ==
                                ↪ y_train))
print("Test accuracy:", 1/X_test.shape[0] * np.sum(nn.
                                ↪ predict(X_test) == y_test)
                                ↪ )
```

Output

```
Iteration: 0, Training Loss: 0.7119608071592811
Iteration: 100, Training Loss: 0.639833196437629
Iteration: 200, Training Loss: 0.5215796424999427
Iteration: 300, Training Loss: 0.368528633036507
Iteration: 400, Training Loss: 0.25561713311943096
Iteration: 500, Training Loss: 0.1887819942694729
Iteration: 600, Training Loss: 0.14915555078792483
Iteration: 700, Training Loss: 0.12416122322345967
Iteration: 800, Training Loss: 0.10734573047261009
Iteration: 900, Training Loss: 0.0953975198921105
Iteration: 1000, Training Loss: 0.08652396074499336
```

```
Train accuracy: 0.9906666666666666
Test accuracy: 0.992
```

4.5.4 Backpropagation network

In machine learning, backpropagation is a widely used algorithm for training feedforward neural networks. Generalizations of backpropagation exist for other artificial neural networks (ANNs), and for functions generally. These classes of algorithms are all referred to generically as "backpropagation". In fitting a neural network, backpropagation computes the gradient of the loss function with respect to the weights of the network for a single input–output example, and does so efficiently, unlike a naive direct computation of the gradient with respect to each weight individually. This efficiency makes it feasible to use gradient methods for training multilayer networks, updating weights to minimize loss; gradient descent, or variants such as stochastic gradient descent, are commonly used. The backpropagation algorithm works by computing the gradient of the loss function with respect to each weight by the chain rule, computing the gradient one layer at a time, iterating backward from the last layer to avoid redundant calculations of intermediate terms in the chain rule; this is an example of dynamic programming.

The term backpropagation strictly refers only to the algorithm for computing the gradient, not how the gradient is used; however, the term is often used loosely to refer to the entire learning algorithm, including how the gradient is used, such as by stochastic gradient descent. Backpropagation generalizes the gradient computation in the delta rule, which is the single-layer version of backpropagation, and is in turn generalized by automatic differentiation, where backpropagation is a special case of reverse accumulation (or "reverse mode"). The term backpropagation and its general use in neural networks was announced in Rumelhart, Hinton & Williams (1986a), then elaborated and popularized in Rumelhart, Hinton & Williams (1986b), but the technique was

independently rediscovered many times, and had many predecessors dating to the 1960s; see § History. A modern overview is given in the deep learning textbook by Goodfellow, Bengio Courville (2016).

Overview

Backpropagation computes the gradient in weight space of a feedforward neural network, with respect to a loss function. Denote:

1. x: input (vector of features)
2. y: target output For classification, output will be a vector of class probabilities (e.g., $(0.1, 0.7, 0.2)$, and target output is a specific class, encoded by the one-hot/dummy variable (e.g., $(0, 1, 0)$.
3. C: loss function or "cost function" For classification, this is usually cross entropy (XC, log loss), while for regression it is usually squared error loss (SEL).
4. L: the number of layers
5. $W^l = (w^l_{jk})$: the weights between layer $l - 1$ and l, where w^l_{jk} is the weight between the $k - th$ node in layer $l - 1$ and the $j - th$ node in layer l
6. f^l: activation functions at layer l For classification the last layer is usually the logistic function for binary classification, and *softmax* (*softargmax*) for multi-class classification, while for the hidden layers this was traditionally a *sigmoid function* (logistic function or others) on each node (coordinate), but today is more varied, with rectifier (*ramp, ReLU*) being common.

In the derivation of backpropagation, other intermediate quantities are used; they are introduced as needed below. Bias terms are not treated specially, as they correspond to a weight with

a fixed input of 1. For the purpose of backpropagation, the specific loss function and activation functions do not matter, as long as they and their derivatives can be evaluated efficiently. Traditional activation functions include but are not limited to sigmoid, tanh, and ReLU. Since, swish, mish, and other activation functions were proposed as well. The overall network is a combination of function composition and matrix multiplication:

$$g(x) := f^L(W^L f^{L-1}(W^{L-1} \cdots f^1(W^1 x) \cdots))$$

For a training set there will be a set of input–output pairs, $\{(x_i, y_i)\}$. For each input–output pair (x_i, y_i) in the training set, the loss of the model on that pair is the cost of the difference between the predicted output $g(x_i)$ and the target output y_i:

$$C(y_i, g(x_i))$$

During model evaluation, the weights are fixed, while the inputs vary (and the target output may be unknown), and the network ends with the output layer (it does not include the loss function). During model training, the input–output pair is fixed, while the weights vary, and the network ends with the loss function.

Backpropagation computes the gradient for a fixed input–output pair (x_i, y_i), where the weights w^l_{jk} can vary. Each individual component of the gradient, $\partial C/\partial w^l_{jk}$, can be computed by the chain rule; however, doing this separately for each weight is inefficient. Backpropagation efficiently computes the gradient by avoiding duplicate calculations and not computing unnecessary intermediate values, by computing the gradient of each layer – specifically, the gradient of the weighted input of each layer, denoted by δ^l – from back to front.

Informally, the key point is that since the only way a weight in W^l affects the loss is through its effect on the next layer, and it does so linearly, δ^l are the only data you need to compute the gradients of the weights at layer l, and then you can com-

pute the previous layer δ^{l-1} and repeat recursively. This avoids inefficiency in two ways. Firstly, it avoids duplication because when computing the gradient at layer l, you do not need to recompute all the derivatives on later layers $l+1, l+2, \ldots$ each time. Secondly, it avoids unnecessary intermediate calculations because at each stage it directly computes the gradient of the weights with respect to the ultimate output (the loss), rather than unnecessarily computing the derivatives of the values of hidden layers with respect to changes in weights $a_{j'}^{l'}/\partial w_{jk}^{l}$.

Backpropagation can be expressed for simple feedforward networks in terms of matrix multiplication, or more generally in terms of the adjoint graph.

Learning as an optimization problem

To understand the mathematical derivation of the backpropagation algorithm, it helps to first develop some intuition about the relationship between the actual output of a neuron and the correct output for a particular training example. Consider a simple neural network with two input units, one output unit and no hidden units, and in which each neuron uses a linear output (unlike most work on neural networks, in which mapping from inputs to outputs is non-linear)[g] that is the weighted sum of its input.

Initially, before training, the weights will be set randomly. Then the neuron learns from training examples, which in this case consist of a set of tuples (x_1, x_2, t) where x_1 and x_2 are the inputs to the network and t is the correct output (the output the network should produce given those inputs, when it has been trained). The initial network, given x_1 and x_2, will compute an output y that likely differs from t (given random weights). A loss function $L(t, y)$ is used for measuring the discrepancy between the target output t and the computed output y. For regression analysis problems the squared error can be used as a loss function, for classification the categorical crossentropy can be used.

As an example consider a regression problem using the square error as a loss:

$$L(t, y) = (t - y)^2 = E,$$

where E is the discrepancy or error.

Consider the network on a single training case: $(1, 1, 0)$. Thus, the input x_1 and x_2 are 1 and 1 respectively and the correct output, t is 0. Now if the relation is plotted between the network's output y on the horizontal axis and the error E on the vertical axis, the result is a parabola. The minimum of the parabola corresponds to the output y which minimizes the error E. For a single training case, the minimum also touches the horizontal axis, which means the error will be zero and the network can produce an output y that exactly matches the target output t. Therefore, the problem of mapping inputs to outputs can be reduced to an optimization problem of finding a function that will produce the minimal error.

However, the output of a neuron depends on the weighted sum of all its inputs:

$$y = x_1 w_1 + x_2 w_2,$$

where w_1 and w_2 are the weights on the connection from the input units to the output unit. Therefore, the error also depends on the incoming weights to the neuron, which is ultimately what needs to be changed in the network to enable learning.

In this example, upon injecting the training data the loss function becomes

$$E = (t - y)^2 = y^2 = (x_1 w_1 + x_2 w_2)^2 = (w_1 + w_2)^2.$$

Then, the loss function E takes the form of a parabolic cylinder with its base directed along $w_1 = -w_2$. Since all sets of weights that satisfy $w_1 = -w_2$ minimize the loss function, in this case additional constraints are required to converge to a

unique solution. Additional constraints could either be generated by setting specific conditions to the weights, or by injecting additional training data.

One commonly used algorithm to find the set of weights that minimizes the error is gradient descent. By backpropagation, the steepest descent direction is calculated of the loss function versus the present synaptic weights. Then, the weights can be modified along the steepest descent direction, and the error is minimized in an efficient way.

Loss function

The loss function is a function that maps values of one or more variables onto a real number intuitively representing some "cost" associated with those values. For backpropagation, the loss function calculates the difference between the network output and its expected output, after a training example has propagated through the network.

Assumptions

The mathematical expression of the loss function must fulfill two conditions in order for it to be possibly used in backpropagation. The first is that it can be written as an average $E = \frac{1}{n}\sum_x E_x$ over error functions E_x, for n individual training examples, x. The reason for this assumption is that the backpropagation algorithm calculates the gradient of the error function for a single training example, which needs to be generalized to the overall error function. The second assumption is that it can be written as a function of the outputs from the neural network.

Example loss function

Let y, y' be vectors in R^n.

Select an error function $E(y, y')$ measuring the difference between two outputs. The standard choice is the square of the

Euclidean distance between the vectors y and y':

$$E(y, y') = \frac{1}{2}\|y - y'\|^2$$

The error function over n training examples can then be written as an average of losses over individual examples:

$$E = \frac{1}{2n}\sum_x \|(y(x) - y'(x))\|^2$$

4.5.5 Python practice

BPN was discovered by Rumelhart, Williams Honton in 1986. The core concept of BPN is to backpropagate or spread the error from units of output layer to internal hidden layers in order to tune the weights to ensure lower error rates. It is considered a practice of fine-tuning the weights of neural networks in each iteration. Proper tuning of the weights will make a sure minimum loss and this will make a more robust, and generalizable trained neural network.

BPN learns in an iterative manner. In each iteration, it compares training examples with the actual target label. target label can be a class label or continuous value. The backpropagation algorithm works in the following steps:

1. *Initialize Network*: BPN randomly initializes the weights.
2. *Forward Propagate*: After initialization, we will propagate into the forward direction. In this phase, we will compute the output and calculate the error from the target output.
3. *Back Propagate Error*: For each observation, weights are modified in order to reduce the error in a technique called the delta rule or gradient descent. It modifies weights in a "backward" direction to all the hidden layers.

Importing libraries

Lets import the required modules and libraries such as numpy, pandas, scikit-learn, and matplotlib.

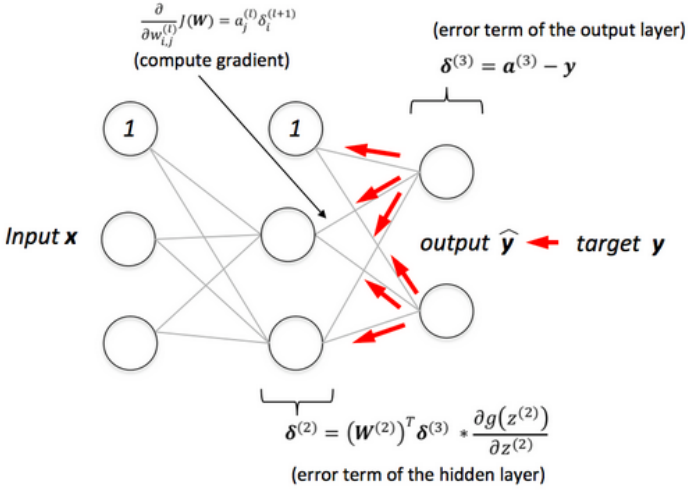

Figure 4.5: Backpropagation procedure

```
# Import Libraries
import numpy as np
import pandas as pd
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

Load Dataset

Let's first load the Iris dataset using *load_iris*() function of scikit-learn library and seprate them in features and target labels. This data set has three classes Iris-setosa, Iris-versicolor, and Iris-virginica.

```
# Load dataset
data = load_iris()

# Get features and target
```

```
X=data.data
y=data.target
```

Prepare Dataset

Create dummy variables for class labels using *get_dummies*() function

```
# Get dummy variable
y = pd.get_dummies(y).values

print(y[:3])
```

Output:

```
array([[1, 0, 0],
       [1, 0, 0],
       [1, 0, 0]], dtype=uint8)
```

Split train and test set

To understand model performance, dividing the dataset into a training set and a test set is a good strategy.

Let's split dataset by using function *train_test_split*(). you need to pass basically 3 parameters features, target, and *test_set* size. Additionally, you can use *random_state* in order to get the same kind of train and test set.

```
#Split data into train and test data
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                ↪  test_size=20,
                                ↪ random_state=4)
```

Initialize Hyperparameters and Weights

Lets initialize the hyperparameters such as learning rate, iterations, input size, number of hidden layers, and number of output layers.

```
# Initialize variables
learning_rate = 0.1
iterations = 5000
N = y_train.size

# number of input features
input_size = 4

# number of hidden layers neurons
hidden_size = 2

# number of neurons at the output layer
output_size = 3

results = pd.DataFrame(columns=["mse", "accuracy"])
```

Lets initialize the weights for hidden and output layers with random values.

```
# Initialize weights
np.random.seed(10)

# initializing weight for the hidden layer
W1 = np.random.normal(scale=0.5, size=(input_size,
                              ↪ hidden_size))

# initializing weight for the output layer
W2 = np.random.normal(scale=0.5, size=(hidden_size ,
                              ↪ output_size))
```

Helper Functions

Lets create helper functions such as sigmoid, *mean_square_error*, and accuracy.

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def mean_squared_error(y_pred, y_true):
    return ((y_pred - y_true)**2).sum() / (2*y_pred.size)

def accuracy(y_pred, y_true):
```

```
    acc = y_pred.argmax(axis=1) == y_true.argmax(axis=1)
    return acc.mean()
```

Backpropagation Neural Network

In this phase, we will create backpropagation neural network in three steps feedforward propagation, error calculation and backpropagation phase. Here , we will create a for loop for given number of iterations that execute the three steps(feedforward propagation, error calculation and backpropagation phase) and update the weights in each iteration.

```
for itr in range(iterations):

    # feedforward propagation
    # on hidden layer
    Z1 = np.dot(x_train, W1)
    A1 = sigmoid(Z1)

    # on output layer
    Z2 = np.dot(A1, W2)
    A2 = sigmoid(Z2)


    # Calculating error
    mse = mean_squared_error(A2, y_train)
    acc = accuracy(A2, y_train)
    results=results.append({"mse":mse, "accuracy":acc},
                           ↪ ignore_index=True )

    # backpropagation
    E1 = A2 - y_train
    dW1 = E1 * A2 * (1 - A2)

    E2 = np.dot(dW1, W2.T)
    dW2 = E2 * A1 * (1 - A1)

    # weight updates
    W2_update = np.dot(A1.T, dW1) / N
    W1_update = np.dot(x_train.T, dW2) / N

    W2 = W2 - learning_rate * W2_update
```

```
    W1 = W1 - learning_rate * W1_update
```

Plot MSE and Accuracy

Lets plot mean squared error in each iteration using pandas *plot()* funciton.

```
results.mse.plot(title="Mean Squared Error")
```

Output:

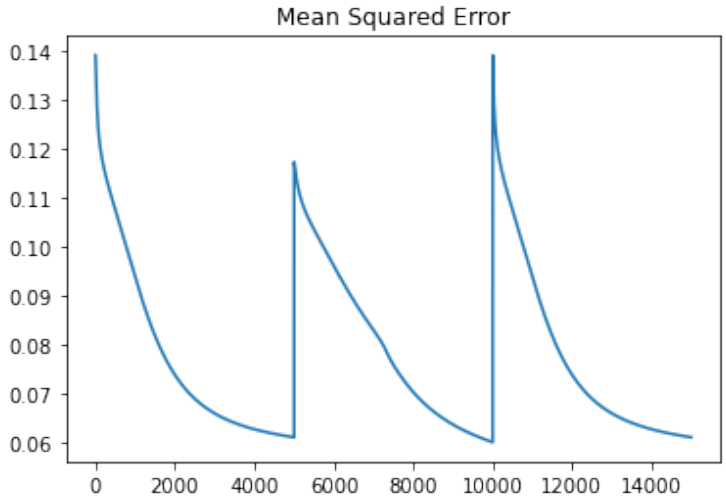

Figure 4.6: Mean squared error

Lets plot accuracy in each iteration using pandas plot() funciton.

```
results.accuracy.plot(title="Accuracy")
```

Output:

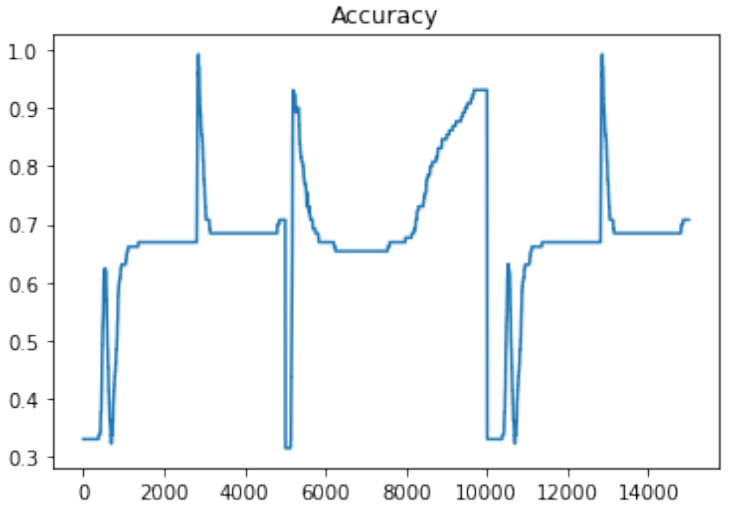

Figure 4.7: Accuracy

Predict for Test Data and Evaluate the Performance

Lets make prediction for the test data and assess the performance of Backpropagation neural network.

```
# feedforward
Z1 = np.dot(x_test, W1)
A1 = sigmoid(Z1)

Z2 = np.dot(A1, W2)
A2 = sigmoid(Z2)

acc = accuracy(A2, y_test)
print("Accuracy: {}".format(acc))
```

Output:

```
Accuracy: 0.8
```

you can see in the above output, we are getting 80% accuracy on test dataset.

Notes

[58]Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities". Proc. Natl. Acad. Sci. U.S.A. 79 (8): 2554–2558. doi:10.1073/pnas.79.8.2554. PMC 346238. PMID 6953413.

[59]McCulloch, Warren; Pitts, Walter (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133. doi:10.1007/BF02478259.

[60]A Hopfield network (or Ising model of a neural network or Ising–Lenz–Little model) is a form of recurrent artificial neural network and a type of spin glass system popularised by John Hopfield in 1982 as described earlier by Little in 1974 based on Ernst Ising's work with Wilhelm Lenz on the Ising model. Hopfield networks serve as content-addressable ("associative") memory systems with binary threshold nodes. Hopfield networks also provide a model for understanding human memory.

[61]Copeland, B. Jack, ed. (2004). The Essential Turing. Oxford University Press. p. 403. ISBN 978-0-19-825080-7.

[62]Bain (1873). Mind and Body: The Theories of Their Relation. New York: D. Appleton and Company.

[63]James (1890). The Principles of Psychology. New York: H. Holt and Company.

[64]Sherrington, C.S. (1898). "Experiments in Examination of the Peripheral Distribution of the Fibers of the Posterior Roots of Some Spinal Nerves". Proceedings of the Royal Society of London. 190: 45–186. doi:10.1098/rstb.1898.0002.

[65]McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133. doi:10.1007/BF02478259.

[66]Farley, B.; W.A. Clark (1954). "Simulation of Self-Organizing Systems by Digital Computer". IRE Transactions on Information Theory. 4 (4): 76–84. doi:10.1109/TIT.1954.1057468.

[67]Rochester, N.; J.H. Holland, L.H. Habit and W.L. Duda (1956). "Tests on a cell assembly theory of the action of the brain, using a large digital computer". IRE Transactions on Information Theory. 2 (3): 80–93. doi:10.1109/TIT.1956.1056810.

[68]Rosenblatt, F. (1958). "The Perceptron: A Probalistic Model For Information Storage And Organization In The Brain". Psychological Review. 65 (6): 386–408. CiteSeerX 10.1.1.588.3775. doi:10.1037/h0042519. PMID 13602029.

[69]Werbos, P.J. (1975). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.

[70]Minsky, M.; S. Papert (1969). An Introduction to Computational

Geometry. MIT Press. ISBN 978-0-262-63022-1.

[71]Rumelhart, D.E.; James McClelland (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Cambridge: MIT Press.

[72]Hebb, D.O. (1949). The Organization of Behavior. New York: Wiley & Sons.

[73]Zell, Andreas (1994). Simulation Neuronaler Netze [Simulation of Neural Networks] (in German) (1st ed.). Addison-Wesley. p. 73. ISBN 3-89319-554-8.

[74]Schmidhuber, Jürgen (2015-01-01). "Deep learning in neural networks: An overview". Neural Networks. 61: 85–117. arXiv:1404.7828.

[75]Auer, Peter; Harald Burgsteiner; Wolfgang Maass (2008). "A learning rule for very simple universal approximators consisting of a single layer of perceptrons" (PDF). Neural Networks. 21 (5): 786–795. doi:10.1016/j.neunet.2007.12.036.

Chapter 5

Applications of Machine Learning

Notes

[58]Hopfield, J. J. (1982). "Neural networks and physical systems with emergent collective computational abilities". Proc. Natl. Acad. Sci. U.S.A. 79 (8): 2554–2558. doi:10.1073/pnas.79.8.2554. PMC 346238. PMID 6953413.

[59]McCulloch, Warren; Pitts, Walter (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133. doi:10.1007/BF02478259.

[60]A Hopfield network (or Ising model of a neural network or Ising–Lenz–Little model) is a form of recurrent artificial neural network and a type of spin glass system popularised by John Hopfield in 1982 as described earlier by Little in 1974 based on Ernst Ising's work with Wilhelm Lenz on the Ising model. Hopfield networks serve as content-addressable ("associative") memory systems with binary threshold nodes. Hopfield networks also provide a model for understanding human memory.

[61]Copeland, B. Jack, ed. (2004). The Essential Turing. Oxford University Press. p. 403. ISBN 978-0-19-825080-7.

[62]Bain (1873). Mind and Body: The Theories of Their Relation. New York: D. Appleton and Company.

[63]James (1890). The Principles of Psychology. New York: H. Holt and Company.

[64]Sherrington, C.S. (1898). "Experiments in Examination of the Peripheral Distribution of the Fibers of the Posterior Roots of Some Spinal Nerves". Proceedings of the Royal Society of London. 190: 45–186. doi:10.1098/rstb.1898.0002.

[65]McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics. 5 (4): 115–133. doi:10.1007/BF02478259.

[66]Farley, B.; W.A. Clark (1954). "Simulation of Self-Organizing Systems by Digital Computer". IRE Transactions on Information Theory. 4 (4): 76–84. doi:10.1109/TIT.1954.1057468.

[67]Rochester, N.; J.H. Holland, L.H. Habit and W.L. Duda (1956). "Tests on a cell assembly theory of the action of the brain, using a large digital computer". IRE Transactions on Information Theory. 2 (3): 80–93. doi:10.1109/TIT.1956.1056810.

[68]Rosenblatt, F. (1958). "The Perceptron: A Probalistic Model For Information Storage And Organization In The Brain". Psychological Review. 65 (6): 386–408. CiteSeerX 10.1.1.588.3775. doi:10.1037/h0042519. PMID 13602029.

[69]Werbos, P.J. (1975). Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.

[70]Minsky, M.; S. Papert (1969). An Introduction to Computational

Geometry. MIT Press. ISBN 978-0-262-63022-1.

[71]Rumelhart, D.E.; James McClelland (1986). Parallel Distributed Processing: Explorations in the Microstructure of Cognition. Cambridge: MIT Press.

[72]Hebb, D.O. (1949). The Organization of Behavior. New York: Wiley & Sons.

[73]Zell, Andreas (1994). Simulation Neuronaler Netze [Simulation of Neural Networks] (in German) (1st ed.). Addison-Wesley. p. 73. ISBN 3-89319-554-8.

[74]Schmidhuber, Jürgen (2015-01-01). "Deep learning in neural networks: An overview". Neural Networks. 61: 85–117. arXiv:1404.7828.

[75]Auer, Peter; Harald Burgsteiner; Wolfgang Maass (2008). "A learning rule for very simple universal approximators consisting of a single layer of perceptrons" (PDF). Neural Networks. 21 (5): 786–795. doi:10.1016/j.neunet.2007.12.036.

Chapter 6

Appendix 1

Python is a high-level, interpreted, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation. Python is dynamically-typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language and first released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000 and introduced new features such as *list comprehensions, cycle-detecting garbage collection, reference counting*, and *Unicode support*. Python 3.0, released in 2008, was a major revision that is not completely backward-compatible with earlier versions. Python 2 was discontinued with version 2.7.18 in 2020. Python consistently ranks as one of the most popular programming languages in the world.

6.1 Installation

Installing Python is very easy doesn't matter if it is Windows, Linux or MacOS. In this section I am going to explain you installation methods for both Windows and Linux. Though, it is not a big deal as such but helps in working with both *OSes*. Most of the world is still revolving around MS Windows. In desktop computing, Windows is the largest OS that rules the world with roughly 87.07% market share. [76]. The second being *Mac* (9.54%), followed by Linux (2.35%). However, this is only related to desktop computing. Android is, of course, leading in mobile phones and other hand-held gadgets. So, most of the folk who use computers seem to use Windows mostly. So, I thought of including a section for Windows folk. However, most of the tasks that I have shown here or the rest of the book are implemented in Linux platform.

6.1.1 Installing in Linux

Working with Python in Linux is very easy for being one simple reason that you don't need to install at all, if you are using a *debian* system like Ubuntu. It is default programming language which comes automatically through OS installation. Execute the following command in the Terminal, to know the current version. Open the terminal by executing keyboard shortcut key such as *Alt+Ctl+T*, if you are in Linux like OS.

```
python --version
```

There will be two Python distributions, in any Linux, at least in Debian based systems. Same statement if executed in Ubuntu (18.04) will return 2.7. Python by default is 2.7. There is other command for version 3 i.e. `python3 --version`. If you skip `--version`, the terminal gets you . Oh! I didn't tell you what is REPL right! REPL is known as *Read-Eval-Print Loop* in short we refer to REPL. It is *an interactive top level or language shell,*

is a simple, interactive computer programming environment that takes single user inputs (i.e. single expressions), evaluates them, and returns the result to the user. [77] If your system is Debian based system:

```
sudo apt-get install python3
```

You can also use `python2.7` in stead of python3. There are addeed advantages of having Python 2.7 if not both. This may be owing to the fact that python grew exponentially during 2.7.

6.1.2 Installing in Windows

Installation in Windows is not that difficult. Python is available as executable binaries. Visit https://www.python.org/ which is an official portal for all Python development. One important section that needs your attention is *PyPI* besides *Doc*. I mean *menus* at https://www.python.org/ website.

6.2 Execution

The program is a collection of statements called a module or simply a script. Execution is the way a script is being processed. Usually languages are expected to be operated thorugh terminals or consoles of respective OSes. However, for user ease there are plenty of Editors. Editors helps users while performing user level tasks such as cut, paste, copy etc. Python executes code in two modes; (1) interactive, (2) batch. Observe the following chunk of code.

6.2.1 Interactive

Interactive mode needs either a shell like utility such as Bash in Linux and CMD in Windows. Python has a shell called REPL. REPL gives a Python prompt for user where the statements or commands can be executed.

```
>>> 1+2
3
>>> (1+2)/2
1.5
>>> print("Hello World!")
Hello World!
```

6.2.2 Batch

Now that you know how to start Python, you should also know how to close the same. To quit Python you need to execute `exit()` not `exit`, please be cautious. The other type of execution is called *batch mode*. The primary purpose of batch mode, of course, is to develop modules and applications. Once the module is finished, it may be called as *source code*. However, developing source code and applications is beyond the scope of this book. However, it is better to know various modes of dealing with Python. We need to follow certain discipline to execute code in batch mode. The following are the few steps that might help understand the batch mode execution in Python.

1. Open text editor (such as *gedit* or *notepad*): It is always better to keep a separate directory for all Python related work. You may use `sudo mkdir` *your_directory_name* if it is *debian* OS. In case of Windows; File Explorer -> Home + New Folder or simply execute *Ctl + Shift + N* inside the window. [78]
2. Write Python statements: We will try a program that returns a sum of two values.
3. Close the file and go back to the Terminal.
4. Execute the file by using `python` *`file`*.

The following shows the execution of the code in Linux Terminal.

```
sudo mkdir python_work
sudo gedit myFirstProg.py
```

1. The first statement creates the directory with a name `python_work`.
2. The second statement opens a text editor. [79]
3. You may write your program as shown below

   ```
   a=2
   b=3
   c=a+b
   print("The Sum of %d and %d is %d" %(a, b, c))
   ```

4. Go back to Terminal and execute the following code.

   ```
   python myFirstProg.py
   ```

   The result of the above statment might be *The Sum of 2, 3 is 5.*

Of couse this is done in certain working directory which also can be work space for the tool.

6.3 I/O and file management

Python has nice ways of managing inputs and outputs. The *os* module is useful to know about file system.

```
>>> import os
>>> os.getcwd()
'C:\\Program Files\\Python310'
>>> os.chdir('D:\\Miscel')
>>> os.getcwd()
'D:\\Miscel'
>>> os.listdir()
['django', 'Document.docx', 'markdown-demo', 'Rplots.pdf'
                            ↪ , 'screen shots', 'test.R'
                            ↪ ]
>>> os.mkdir('pythonwork')
>>> os.listdir()
['django', 'Document.docx', 'markdown-demo', 'pythonwork'
                            ↪ , 'Rplots.pdf', 'screen
                            ↪ shots', 'test.R']
>>> os.rmdir('pythonwork')
```

```
>>> os.listdir()
['django', 'Document.docx', 'markdown-demo', 'Rplots.pdf'
                              ↪ , 'screen shots', 'test.R'
                              ↪ ]
```

Python use \\ as path separator. The above code snippet shows few tasks such as creating, changing removing directories. Following code snippet shows writing data to a file.

```
>>> file_path = os.path.join('pythonwork', 'test.txt')
>>> file_path
'pythonwork\\test.txt'
```

These statements will create a file with a name *test.txt* in current directory.

```
with open(file_path, 'w') as fw:
    for i in range(5):
        fw.write(str(i))
    print('Done')


1
1
1
1
1
Done
```

Above code writes a linear number series from 1 to 5 to the newly created file.

```
>>> with open(file_path, 'r') as fr:
...     x = fr.read()
...     print('Done')
...
...
Done
>>> x
'01234'
>>> for i in x:
...     int(i)
```

```
...
...
0
1
2
3
4
```

Above code reads the data to Python shell from the same file that was created earlier. It is also possible to save such data as in the form of data structures such as vectors, arrays, dictionaries etc. There is a in-built module called *csv* which helps reading data from CSV files. Suppose if I have some gender based data in certain CSV file called gender.csv in my file system. The data can be read to Python shell using the following code.

```
>>> with open('pythonwork\\gender.csv', 'r') as gf:
...     gender = gf.readlines()
...
...
>>> gender
['male\n', 'female\n', 'male\n', 'female\n', 'female\n',
                            ↪ 'male\n', 'female\n', '
                            ↪ male\n', 'female\n', 'male
                            ↪ \n']
```

The resultant object *gender* is a list. We can print the data nicely ordered in the shell

```
>>> for i in gender:
...     print(i.replace('\n', ''))
...
...
male
female
male
female
female
male
female
male
female
```

```
male
```

CSV files can be structured in rows and columns. At times we may be having more than one column in CSV files. Assume that I have two columns in a file called *data.csv* which represents *gender* and *age*. In such case the file can processed as shown below.

```
>>> with open('pythonwork\\gender.csv', 'r') as gf:
...     data = gf.readlines()
...
...
>>> data
['male,21\n', 'female,74\n', 'male,64\n', 'female,62\n',
                             ↪ 'female,66\n', 'male,23\n'
                             ↪ , 'female,30\n', 'male,73\
                             ↪ n', 'female,62\n', 'male,
                             ↪ 80\n']
```

The data in the object *data* is not usable. It needs to be separated into two separate lists in order to use the variables.

```
>>> for i in data:
...                 a, b = i.split(',')
...                 j.append(a)
...                 k.append(b)
...                 print('Done')
...
...
Done
Done
Done
Done
Done
Done
Done
Done
Done
Done
>>> j
...
```

```
['male', 'female', 'male', 'female', 'female', 'male', '
                              ↪ female', 'male', 'female',
                              ↪  'male']
>>> k
...
['21\n', '74\n', '64\n', '62\n', '66\n', '23\n', '30\n',
                              ↪ '73\n', '62\n', '80\n']
>>> k_ = []
>>> for i in k:
...                 k_.append(i.replace('\n', ''))
>>> k_
...
['21', '74', '64', '62', '66', '23', '30', '73', '62', '
                              ↪ 80']
```

Let us computes summaries for these two variables.

```
>>> c = 0
>>> d = 0
>>> for i in j:
...                 if i == 'male':
...                     c = c + 1
...                 else:
...                     d = d + 1
...
...
>>> (c, d)
(6, 5)
>>> print('gender: ' + repr(c) + ', ' + 'age: ' + repr(d)
                              ↪ )
gender: 6, age: 5
```

6.4 Data types & data structures

Data type refers to primitives of programming. For instance, string, integer, float, double, complex are known as primitives. Data structures are combinations of these types. For instance,

```
>>> type(1+2)
<class 'int'>
>>> x = 1; y=2; z=x+y; type(z)
```

```
<class 'int'>
>>> x = 1; y=2; z=x/y; type(z)
<class 'float'>
>>> x = 1000000; y=2000000; z=x/y; type(z)
<class 'float'>
>>> x = 1.5; y=2.5; z=x*y; type(z)
<class 'float'>
>>> my_name = 'kamakshaiah musunuru'
>>> type(my_name)
<class 'str'>
```

Python has sufficiently large number of data structures for various needs of data processing. Following is the list of data structures that are supported in

1. Tuple
2. List
3. Dictionary
4. Set

6.4.1 Tuple

```
>>> x, y = (1, 2)
>>> x; y
1
2
>>> x, y = (1, 2)
>>> x; y
1
2
>>> x = (1, 2, 3, 4, 5)
>>> x[0]
1
>>> x[1]
2
```

Tuples are highly dynamic and easy to handle through index numbers. Tuples are created using *parentheses* to contain values.

6.4.2 List

Lists are created using *square brackets* to contain values. Lists has quite a few methods associated with it. Use help('list') to know more about lists.

```
>>> x = list()
...
>>> type(x)
...
<class 'list'>
>>> list_methods = [func for func in dir(list) if
                            ↪ callable(getattr(list,
                            ↪ func)) and not func.
                            ↪ startswith('__')]
...
>>> len(list_methods)
...
11
>>> list_methods
...
['append', 'clear', 'copy', 'count', 'extend', 'index', '
                            ↪ insert', 'pop', 'remove',
                            ↪ 'reverse', 'sort']
```

List is a class not a function. There are 11 methods associated with *list* class. How lists are contained?

```
>>> import random
>>> random.randint(1, 2)
2
>>> list(range(5))
[0, 1, 2, 3, 4]
>>> x = list(range(5))
>>> x
[0, 1, 2, 3, 4]
```

List comprehensions

One of the most distinctive features of Python is the *list comprehension*, which can be used to create powerful functionality within a single line of code. However, many developers struggle

to fully leverage the more advanced features of a list comprehension in Python. Some programmers even use them too much, which can lead to code that s less efficient and harder to read. List comprehensions provides easy way to manipulate lists. For instance,

```
>>> [i for i in range(10)]
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [random.randint(1, 10) for i in range(10)]
...
[7, 5, 3, 4, 4, 4, 9, 7, 7, 5]
>>> [round(random.random()*100, 2) for i in range(10)]
...
[55.18, 71.09, 82.87, 32.67, 3.51, 66.0, 36.21, 26.81, 22
                              ↪ .56, 50.77]
>>>
```

We can even create named lists and code, transform them as shown below.

```
>>> x = [random.randint(1, 2) for i in range(10)]
>>> x
[1, 1, 2, 2, 1, 1, 2, 2, 1, 1]

>>> y = ['male' if x[i] == 1 else 'female' for i in range
                              ↪ (len(x))]
>>> y
['male', 'male', 'female', 'female', 'male', 'male', '
                              ↪ female', 'female', 'male',
                              ↪  'male']
```

6.4.3 Dictionary

The other important data structure is *dictionary*. Dictionary is a special type of data structure that has two important attributes called *keys* and *values*.

```
>>> data_dict = dict(a=1, b=[round(random.random()*100, 2
                              ↪ ) for i in range(5)])
>>> data_dict
{'a': 1, 'b': [92.03, 48.61, 15.63, 91.54, 49.51]}
```

```
>>> data_dict.keys()
dict_keys(['a', 'b'])
>>> data_dict.items()
dict_items([('a', 1), ('b', [92.03, 48.61, 15.63, 91.54,
                             ↪ 49.51])])
>>> data_dict.keys()
dict_keys(['a', 'b'])
>>> data_dict.values()
dict_values([1, [92.03, 48.61, 15.63, 91.54, 49.51]])
```

It is possible to make summaries of contents of dictionaries, but they need to be changed to lists or other structures suitable for computations.

```
>>> import statistics as sts

>>> list(data_dict.values())
[1, [92.03, 48.61, 15.63, 91.54, 49.51]]
>>> list(data_dict.values())[1]
[92.03, 48.61, 15.63, 91.54, 49.51]
>>> sts.mean(list(data_dict.values())[1])
59.464
```

6.4.4 Set

A set can be thought of simply as a well-defined collection of distinct objects, typically called elements or members. Sets are useful to arrange unique elements of any data structure into another data structure. For instance,

```
>>> education = [random.randint(1, 3) for i in range(10)]
...
>>> education
...
[2, 1, 3, 3, 1, 3, 1, 1, 1, 1]
>>> educ_factor = ['primary' if education[i] == 1 else '
                             ↪ secondary' if education[i]
                             ↪  == 2  else 'higher' for i
                             ↪  in range(len(education))]
...
>>> educ_dict = dict(data=education, levels=set(
                             ↪ educ_factor))
```

```
...
>>> educ_dict['data']
...
[2, 1, 3, 3, 1, 3, 1, 1, 1, 1]
>>> educ_dict['levels']
...
{'primary', 'secondary', 'higher'}

>>> list(educ_dict['levels'])[0]
...
'primary'
```

In the above code, *set()* is used to find out unique instances or elements in the variable called *education*. There are so many methods associated with this class set.

```
>>> x = set()
...
>>> type(x)
...
<class 'set'>
```

So *set()* is not a function but a class.

```
>>> method_list = [func for func in dir(set) if callable(
                            ↪ getattr(set, func)) and
                            ↪ not func.startswith("__")]
...
>>> method_list
...
['add', 'clear', 'copy', 'difference', 'difference_update
                            ↪ ', 'discard', '
                            ↪ intersection', '
                            ↪ intersection_update', '
                            ↪ isdisjoint', 'issubset', '
                            ↪ issuperset', 'pop', '
                            ↪ remove', '
                            ↪ symmetric_difference', '
                            ↪ symmetric_difference_update
                            ↪ ', 'union', 'update']
>>> len(method_list)
...
17
```

There are approximately 17 different methods available for *set()* class.

6.5 Functions

Python is multi-paradigm language. It is possible to write functions and routines related to object oriented programming. Functions are defined by a keyword called *def* followed by name of the function and then arguments within parentheses.

```
def mean(x):
    return sum(x)/len(x)

def variance(x):
    n = len(x)
    am = mean(x)
    return sum((x - am) ** 2 for x in x) / (n - 1)

def stdev(x):
    res = variance(x)
    return math.sqrt(res)
```

Above code has three functions and they calculates *arithmetic mean, variance* and *standard deviation* respectively. Now let us test these functions with data simulated using an in-built module called *random*.

```
import random
import math
import statistics as sts

data_x  = [random.randint(10, 100) for i in range(10)]
print(data_x)
print(mean(data_x))
print(sts.mean(data_x))
print(variance(data_x))
print(sts.variance(data_x))
print(stdev(data_x))
print(sts.stdev(data_x))
```

The output for above code could be

```
[54, 49, 59, 93, 26, 40, 40, 78, 31, 63]
53.3
53.3
436.4555555555555
436.4555555555556
20.89151874698332
20.891518746983323
```

The values are checked using a package *statistics*. The values of *user defined functions* perfectly match with *built-in functions* of this package.

6.6 Classes

Classes provide a means of bundling data and functionality together. Creating a new class creates a new type of object, allowing new instances of that type to be made. Each class instance can have attributes attached to it for maintaining its state. Class instances can also have methods (defined by its class) for modifying its state.

Compared with other programming languages, Python s class mechanism adds classes with a minimum of new syntax and semantics. It is a mixture of the class mechanisms found in C++ and Modula-3. Python classes provide all the standard features of Object Oriented Programming: the class inheritance mechanism allows multiple base classes, a derived class can override any methods of its base class or classes, and a method can call the method of a base class with the same name. Objects can contain arbitrary amounts and kinds of data.

Following code shows one of the OOP feature known as encapsulation handled through few class based functions.

```
class summaries():

    data = list()
```

```
    def __init__(self, data):
        self.data = data

    def mean(self):
        return sum(self.data)/len(self.data)

    def variance(self):
        n = len(self.data)
        am = self.mean()
        return sum((i - am) ** 2 for i in self.data) / (n
                                        ↪  - 1)

    def stdev(self):
        res = self.variance()
        return math.sqrt(res)
```

We will try to instantiate this class and create object. We can use the class methods together with objects but using *dot* notation, just any other function.

```
import random
import math
import statistics as sts

x  = [random.randint(10, 100) for i in range(10)] # data
                                ↪ simulation

summaries = summaries(x) # instantiation

print(summaries.mean())
print(summaries.variance())
print(summaries.stdev())
```

Output for the above code could be

```
58.9
892.5444444444444
29.87548232990464
```

Chapter 7

Appendix 2

7.1 Data simulations

Programmers never care for data, they just use data only for testing code. In fact, programmers can create or simulate data sets required for their use. This may sound a bit odd for data analysts for whom data is a collection of rows and columns which is obtained through meticulously defined methods with sacrosanct efforts. For analyst, every thing starts from data. Analysts expects data to be precise and also accurate but it is not true for programmers. For programmers, the data is just bits and bytes. Analysts need to care context or scenarios to which data is linked to. For instance, the variable *age* has certain sense to analysts. However, for a programmer it is just a collection of numbers may be an array, a vector, or at most a data matrix. Programmers try to see these variables as collection of integers or decimals or anything else from the number system. The aim of the programmer is to see the data fits to a model defined through code or *vice versa*. That kind of effort is referred to data science.

A simulation is the imitation of the operation of a real-world process or system over time. Data simulations are highly useful for testing models. Usually simulations required to use models. Statisticians refer them to probability distributions. To start with, a number series can be modeled as linear or non-linear depending on the context. In the same fashion every probability distribution an underlying model in it. For instance, a normally distributed data arise or originate from a process known as "Gaussian". The model for normal distribution is a mathematical representation of that process. As such these models or processes determines the the very occurrence of data points. In this section there are few yet very potential methods that are necessary to deal with few practice exercises given in this text or book.

Following code is available from a companion Github repository at ml-book. The code available from a module called *datasimulations*.

```
import random

def createFactor(cats=['male', 'female'], n=10):
    cats = cats*n
    out = random.sample(cats, n)
    return out

def createVector(start=1, end=10, n=10, m= 0, Type=None):
    if Type=='linear':
        out = list(range(n))
    elif Type=='random':
        out = [round(random.uniform(1, n)*m, 0) for i in
                                        ↪ range(n)]
    else:
        out = [random.randint(1, n)*m for i in range(n)]
    return out
```

There are two methods in the above modules and these methods are plain Python functions (defs). These functions can be used to simulate multivariate data sets using following procedure.

```
>>> from datasimulations import createFactor,
                              ↪ createVector
>>> import random
>>> createFactor(['yes', 'no'], 10)
['no', 'yes', 'yes', 'yes', 'no', 'yes', 'no', 'yes', 'no
                              ↪ ', 'no']
>>> createVector(1, 10, m=100, Type='random')
[220.0, 621.0, 373.0, 344.0, 638.0, 361.0, 292.0, 499.0,
                              ↪ 308.0, 951.0]
```

Finally, creating data sets is very easy using *pandas* package. Following code snippet shows the procedure to create multivariate data set with three different data variables known as *age, education, gender* and a dichotomous target variable called *target*. All these variables are defined as columns in a data set called *df*.

```
age = createVector(1, 10, m=10, Type='random')
age = [round(i, 0) for i in age]
education = gender = createFactor(['primary', 'secondary'
                              ↪ , 'higher'], 10)
gender = createFactor(['yes', 'no'], 10)
target = createVector(1, 2)
df = pd.DataFrame({'age': age, 'gender': gender, '
                              ↪ education':education, '
                              ↪ target': target})
>>> df
    age gender  education  target
0  73.0     no    primary       2
1  64.0     no    primary       2
2  98.0    yes     higher       1
3  58.0    yes  secondary       2
4  14.0     no    primary       1
5  13.0     no    primary       1
6  51.0     no     higher       2
7  30.0    yes    primary       2
8  30.0     no  secondary       1
9  86.0    yes  secondary       1

>>> df['age'].mean()
51.7
```