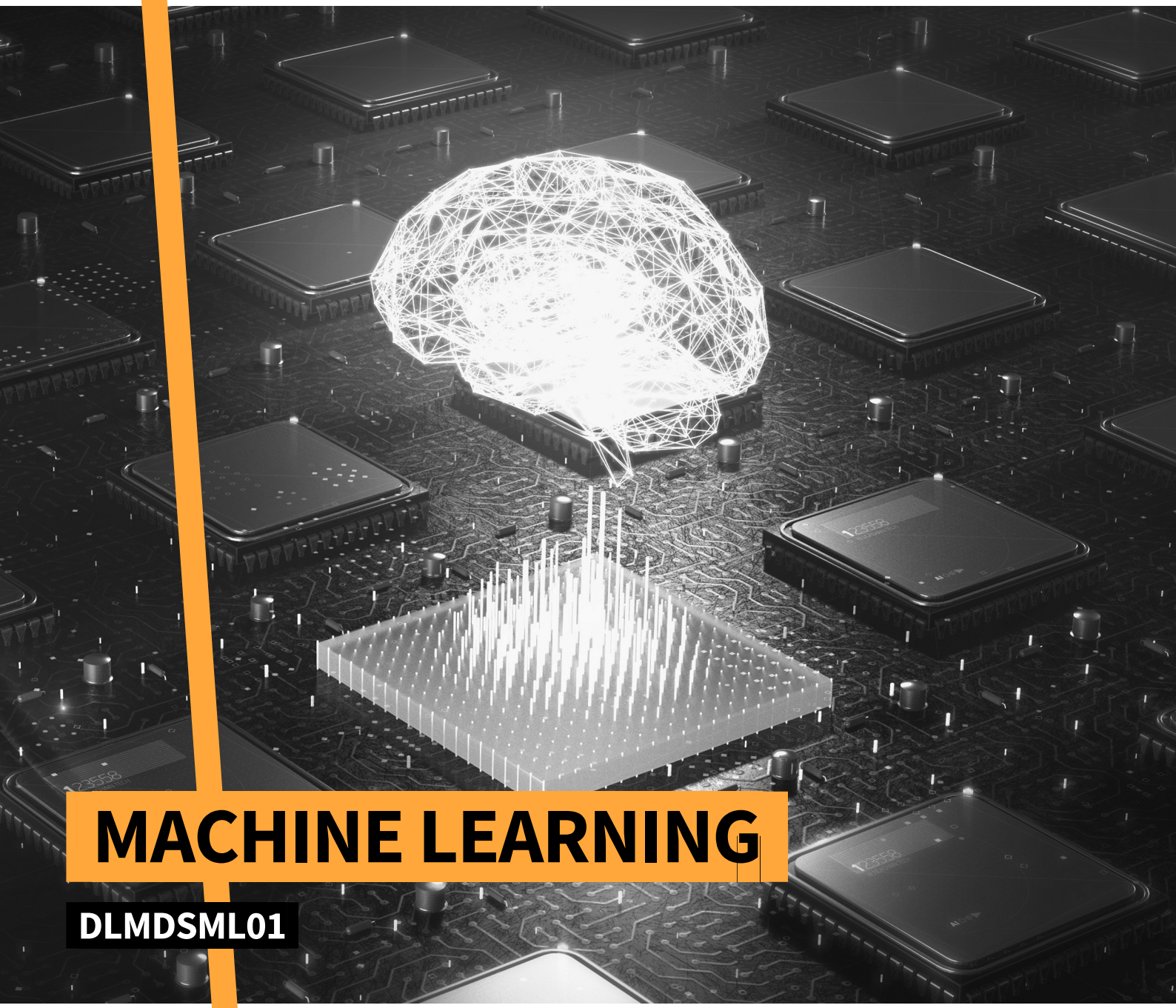
Course Book



MACHINE LEARNING

DLMDSML01

iu
INTERNATIONAL
UNIVERSITY OF
APPLIED SCIENCES

MACHINE LEARNING

MASTHEAD

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

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

DLMDSML01
Version No.: 001-2024-0328

N. N.

TABLE OF CONTENTS

MACHINE LEARNING

INTRODUCTION

WELCOME

SIGNPOSTS THROUGHOUT THE COURSE BOOK

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

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

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

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

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

Good luck!

BASIC READING

Akerkar, R., & Sajja, P. S. (2016). *Intelligent techniques for data science*. Springer International Publishing.

Hodeghatta, U. R., & Nayak, U. (2017). *Business analytics using R- A practical approach*. Apress Publishing. Database: EBSCOhost

Lahoz-Beltra, R. (2016). *SGA: Simple Genetic Algorithm (SGA) in Python*. Database: BASE

Runkler, T. A. (2012). *Data analytics: Models and algorithms for intelligent data analysis.* Springer Vieweg Press. Database: Springer eBook Package English Computer Science

Skiena, S. S (2017). *The data science design manual*. Springer International Publishing. Database: Springer eBook Package English Computer Science

REQUIRED READING

UNIT 1

Igual, L., & Seguí, S. (2017). *Introduction to data science*. Springer International Publishing. Chapter 5 Database: Springer eBook Package English Computer Science

Skiena, S. S (2017). *The data science design manual*. Springer International Publishing. Chapters 11.1 and 11.4. Database: Springer eBook Package English Computer Science

UNIT 2

Mailund, T. (2017). *Beginning data science in R*. Apress Publishing. Database: EBSCOhost Chapters 6 and 7

Runkler, T. A. (2012). *Data Analytics: Models and algorithms for intelligent data analysis.* Springer Vieweg Press. Database: BASE Chapter 9

UNIT 3

Igual, L., & Seguí, S. (2017). *Introduction to data science*. Springer International Publishing. Chapters 2 and 7 Database: Springer eBook Package English Computer Science

Runkler, T. A. (2012). *Data analytics: Models and algorithms for intelligent data analysis.* Springer Vieweg Press. Database: BASE Chapter 6

UNIT 4

Abe, S. (2010). *Support vector machines for pattern classification.* Springer. Database: ProQuest Ebook Central Chapters 1—3

Awad, M., & Khanna, R. (2015). *Efficient learning machines*. Apress. Database: eBook Index Chapter: Support Vector Regression

Christmann, A., & Steinwart, I. (2008). *Support vector machines*. Springer. Database: Internationale Hochschule Bad Honnef Bonn

UNIT 5

Barros, R., Carvalho, A., & Freitas, A. (2015). *Automatic design of decision-tree induction algorithms*. Springer. Database: Internationale Hochschule Bad Honnef Bonn

Han, J. (2005). *Data mining: Concepts and techniques.* Morgan Kaufmann. Database: Internationale Hochschule Bad Honnef Bonn

UNIT 6

Michalewicz, Z. (1994). *Genetic algorithms + data structures = evolution programs*. Springer.

Lienland, B., & Zeng, L. (2015). A review and comparison of genetic algorithms for the 0-1 multidimensional knapsack problem. *International Journal of Operations Research and Information Systems (IJORIS), 6*(2), 21—31. Database: BASE

Luque, G., & Alba, E. (2011). *Parallel genetic algorithms: Theory and real world applications*. Springer-Verlag. Database: BASE

FURTHER READING

UNIT 1

Pathak, M. (2014). *Beginning data science with R*. Springer International Publishing. Database: EBSCOhost Chapter 7.2

Runkler, T. A. (2012). *Data analytics: Models and algorithms for intelligent data analysis.* Springer Vieweg Press. Database: Springer eBook Package English Computer Science. Chapters 6.4 and 8.6

UNIT 2

Hodeghatta, U. R., & Nayak, U. (2017). *Business analytics using R- A practical approach*. Apress Publishing. Database: EBSCOhost Chapter 8.4

Igual, L., & Seguí, S. (2017). *Introduction to data science*. Springer International Publishing. Database: Springer eBook Package English Computer Science Chapters 5.9 and 7.3

UNIT 3

Skiena, S. S (2017). *The data science design manual.* Springer International Publishing. Database: Springer eBook Package English Computer Science Chapter 9

Unpingco, J. (2016). *Python for probability, statistics, and machine learning*. Springer. Database: Internationale Hochschule Bad Honnef Bonn Chapter 4

UNIT 4

Shigeo, A. (2005). *Support vector machines for pattern classification*. Springer. Database: Internationale Hochschule Bad Honnef Bonn

Thorsten, J. (2001). *Learning to classify text using support vector machines*. Springer. Database: Internationale Hochschule Bad Honnef Bonn

Yunqian, M., & Guodong, G. (2014). *Support vector machines applications.* Springer. Database: Internationale Hochschule Bad Honnef Bonn

UNIT 5

Dahan, H., Cohen, S., Rokach, L., & Maimon, O. (2014). *Proactive data mining with decision trees*. Springer. Database: Internationale Hochschule Bad Honnef Bonn

Singh, S., & Gupta, P. (2014). Comparative study ID3, cart and C4.5 decision tree algorithm: a survey. *International Journal of Advanced Information Science and Technology (IJAIST)*, *27*, 97—103. Available online.

LEARNING OBJECTIVES

The aim of **"Machine Learning"** is to equip the students with the necessary knowledge about machine learning techniques and mathematical tools to take a leading role in a data science or artificial intelligence team.

While neural networks and deep learning have become cornerstones of modern machine learning, this book focuses on the other widely implemented machine learning techniques such as decision trees, clustering, and SVM.

The book starts with a general overview to introduce the field of machine learning and its nomenclature. A complete explanation is presented for machine learning, what is meant by classification and regression, and supervised versus unsupervised versus semi-supervised learning. An introduction to reinforcement learning is provided, as well as its important aspects. The discussion is extended to cover what a policy and the value functions are, how model-based and model-free reinforcement learning work, etc.

Afterwards, several unsupervised learning approaches (i.e., clustering algorithms) are explained to show how input data can be distributed into clusters according to different metrics that measure similarity, proximity, and density of data points.

Regression analysis is discussed in detail to present how a supervised learning model for the relationship between dependent variables (targets) and independent variables (features) is established, either in linear or nonlinear form. Moreover, estimators such as maximum likelihood and least squares are introduced to explain model behavior. Model accuracy measures are described that can be used to decide about the quality of the model's performance. Furthermore, since most of the regression models are un-regularized, and, for example, prone to outliers, regularization tools such as lasso and ridge are often used to add an appropriate penalty term to the process to make the models more robust.

The support vector machine (SVM) technique for classification and regression is explained to understand how they work in detail from a mathematical perspective as well. SVMs are essentially linear classifiers that use a line/plane/hyperplane in n dimensions to separate classes from each other. The main parameters such as the margin, support vectors, hard margin, margin violations, soft margin, and slack variables are introduced. In nonlinear problems, SVM maps to higher dimensions to find a hyperplane that separates the classes by employing the kernel trick.

Additionally, the different types of decision trees (DT) technique are described, which show the corresponding feature ranking methods used by each type (such as CART, ID3, and C4.5). This is followed by an explanation on how the decision tree can be used as a single-tree classifier, ensemble classifier, or as a clustering technique.

Finally, the idea behind the evolutionary algorithms and the different types of these algorithms are presented. In particular, the genetic algorithm is explained as a particular type of evolutionary algorithm that implements the Darwinian theory of survival of the fittest.

The different phases of the algorithm are discussed including the crossover and mutation phases. A detailed optimization problem is discussed and solved by applying the genetic algorithm.

The topics discussed in this course book use the Python eco-system as the technical programming tool for selected real life examples.

UNIT 1

INTRODUCTION TO MACHINE LEARNING

STUDY GOALS

On completion of this unit, you will have learned …

- what is meant by machine learning.
- common terms and definitions in machine learning.
- the different applications of machine learning.
- the concepts of classification and regression.
- the difference between each of the machine learning paradigms.
- how reinforcement learning is performed.

1. INTRODUCTION TO MACHINE LEARNING

Introduction

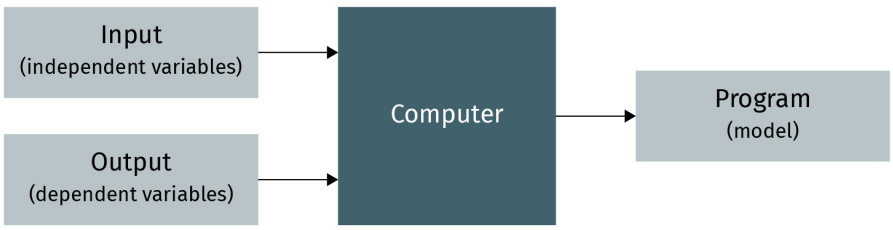
A home appliance company plans to offer an online purchasing service for its products. The marketing team proposed different advertisement campaigns to attract customers that perform the correct clicks. After few weeks, the data analytics team became able to conclude which campaign(s) brought the highest revenue as well as the top customers. The data analytics team had to dive into the purchasing data to find the "patterns" which define the campaign with the highest revenue, and to"predict" the type of customers who bought the most products or spent the most money. For the vast amount of purchasing data, which increases over time, the task becomes intractable by human or manual implementations. Therefore, machine learning tools are introduced to employ the high performance capabilities of a computer (i.e., machine) in handling this large amount of purchasing data (i.e., inputs), and develop a model which learns from this data to automatically and efficiently achieve both the pattern finding and prediction tasks (i.e., outputs).

The challenge is to find an algorithmic description of the task that can be expressed in the form of a classical computer program based on the knowledge that we have concerning the domain. Instead, large amount of inputs (purchasing data) are collected as well as their associated outputs (underlying patterns and predictions). Then, we fit a mathematical model to define the relations between the inputs and the outputs, and utilize it to predict the outputs for new scenarios. The developed model is called a **machine learning** model.

Machine Learning
This field is concerned with the question of how to construct computer programs that automatically improve their performance (P) from experience (E) with respect to some provided tasks (T).

Machine learning uses a computing device to process data, for example, the input products campaigns (which are called the independent variables) and their associated output revenue (which is called the dependent, or target, variable). The output will be a model (i.e., mapping of inputs to output that is typically represented as a program) that performs the pattern finding or prediction tasks, as schematically described in the following figure.

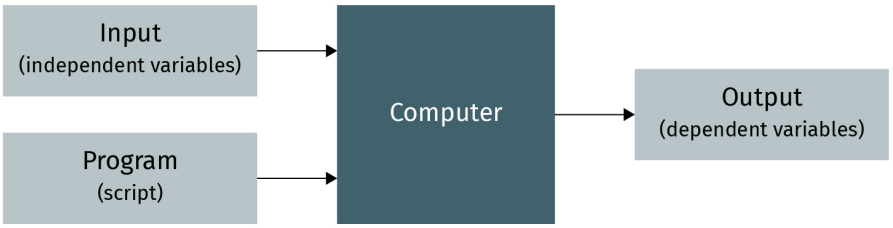
Figure 1: Machine Learning



Source: Walid Hussein, 2020.

Hence, machine learning is different from traditional programming, which constructs an explicit processing of input variables into desired outputs via a set of code instructions, as explained in the following figure.

Figure 2: Traditional Programming



Source: Walid Hussein, 2020.

Machine learning is a mathematical and algorithmic approach that builds a model from the given data to perform predictions or generate insights about new data scenarios Samuel, 1959.

In practice, the data analytics task starts by understanding the data domain and applying the necessary data integration, pre-processing, etc. Afterwards, the machine learning models are developed, and the results are interpreted. Finally, the discovered patterns and the predicted outputs are presented.

Data are given as input/output pairs with no explicit relations between them, hence, the need to develop a model that discovers the important hidden relationships and correlations between the data elements. Machine learning is then an important approach to transforming data, which is rather useless in its raw form, into meaningful information. To achieve this goal, machine learning integrates many distinct mathematical tools such as probability and statistics, optimization, and control theory.

The developed machine learning models find application in many disciplines, ranging from vision/language processing, forecasting, pattern recognition, games, data mining, expert systems, and robotics. Machine learning use cases cover many fields such as

- recognizing patterns for facial identification and emotion detection.
- recognizing handwritten characters and speech.
- recognizing objects of interest in medical images.
- detection of motion in consequent images.
- recognizing unusual credit card transactions, sensor readings, engine sounds, etc.
- prediction of future stock prices or currency exchange rates.
- detection of fraud and spam filtering.
- establishing a recommendation system.
- grouping documents or images with similar contents.

- dimensionality reduction, i.e., to make a set of high-dimensional data points amenable to visual inspection.

It is all about data! For a given dataset of inputs (independent variables) and outputs (labels or dependent variables), machine learning methods are implemented to discover the patterns within the inputs and/or to predict the relationships between the inputs and outputs. These relationships can be utilized later to predict the outputs for new inputs. In general, the possible outputs in any dataset are either continuous or discrete values. For example, the outputs in a students' grading dataset are continuous values in the range from 0 to 100 (mark), while the outputs in a weather datasets may be discrete values, such as "0" for windy, "1" for sunny, and "2" for cloudy.

For continuous outputs, machine learning builds a prediction model called a regression model. For discrete outputs, the prediction model is called a classification model, where the outputs indicate a class for each of the possible inputs. There are many approaches to performing regression and classification, as will be discussed in detail.

On the other hand, when the objective is to discover the hidden patterns within the data (i.e., there are no target outputs provided), machine learning techniques can be used to perform a clustering analysis, which groups the inputs into homogenous clusters according to their degree of similarity based on the values of the given independent variables. Moreover, other types of analyses can be performed on the inputs to detect outliers (e.g., data records that are highly atypical in comparison to the remainder of the dataset) or to remove irrelevant variables and duplicated data (i.e., dimensionality reduction).

In general, there are four major machine learning paradigms, each of them corresponding to a particular abstract learning task. These are supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning. Supervised learning denotes the learning tasks when data inputs and corresponding target outputs are provided, and includes classification and regression approaches. Unsupervised learning is concerned with discovering the hidden patterns in the data inputs and includes clustering as an important sub-domain. Unsupervised learning is considered an important frontier of machine learning because most big datasets do not come with labels (i.e., outputs are not known). Semi-supervised learning covers problems where only partial label information exists. Finally, reinforcement learning denotes the learning setup where the goal is to find an action policy that achieves a given goal. This model learns how to achieve that goal by trial-and-error interactions with its environment.

1.1 Classification & Regression

Given a dataset that includes hundreds of emails and their attributes, where each email is labeled by either spam or not spam, can a newly received email be classified as spam?

For a dataset related to red wine samples with their physicochemical variables as inputs, and their associated quality (poor, average, or excellent) as outputs: If a new wine sample is provided with its physicochemical variables, can its quality be predicted?

The above two problems are typical classification problems, where the dataset is a collection of labeled-data records in the form: {independent variables as inputs, and the associated classes (i.e., labels) as outputs}. The objective is to develop a machine learning model to relate the inputs to the outputs, and to predict the class of new inputs.

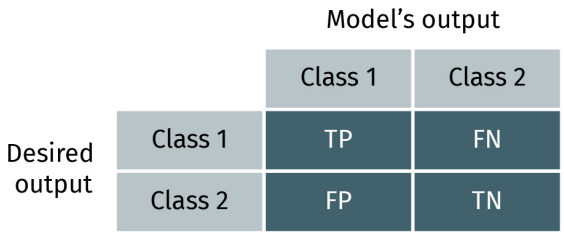
In classification, the outputs are finite and categorical, and the developed model must assign a single class to new inputs. Therefore, the classes should be disjointed.

In practice, the dataset is divided into two sets, which are the training and the testing sets. The **training set** is employed to develop the classification model, while the **testing set** is utilized afterwards to evaluate the accuracy of the developed model. For binary classification models, the outputs of the model can be presented in a confusion matrix form. The confusion matrix shows how many instances are correctly classified by the developed model, as seen in the following figure for a two-class problem.

Training set
This dataset is used by the machine learning model to build its predictive mathematical function.

Testing set
This dataset is used to measure the performance of the developed machine learning model.

Figure 3: The Confusion Matrix



Source: Walid Hussein, 2020.

Here, TP (True Positives) stands for the number of data instances that belong to Class 1 and are correctly classified as such by the trained model. TN (True Negatives) stands for those instances belonging to Class 2 that are correctly classified by the model. FP (False Positives) stands for the number of instances that belong to Class 2, but are wrongly classified by the model to be in Class 1. Finally, FN (False Negatives) stands for those instances that belong to Class 1, and the model wrongly classified them to Class 2.

The evaluation of a classification model is achieved by one or both of the following metrics:

- precision. The number of true positives (i.e., the number of correctly labeled instances) to a specific class, divided by the total number of instances assigned to this class.

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

- recall. The number of true positives to a specific class, divided by the total number of instances actually in this class.

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

The precision is a measure of the model's exactness, whereas recall is a measure of the model's completeness. Another metric is called F-score, and it measures the weighted average of precision and recall, as shown below.

$$\text{F-score} = \frac{2(\text{Precision} \cdot \text{Recall})}{\text{Precision} + \text{Recall}}$$

There are many techniques to develop classification models. In many classification problems, several techniques may be tested in parallel, and the technique that returns the highest evaluation performance is selected. Typically, it is very difficult to decide at the beginning of the analysis which classification technique will perform best, especially if there are no prior assumptions to the nature of the classification task. Therefore, the experience of the data scientists together with proper testing and evaluation of different approaches is the best way to solve a given problem.

Alternatively, in regression problems, the task is to develop a machine learning model that predicts a value, not a class. In general, the developed model is a mathematical function that relates the outputs to the inputs.

One example is the historical dataset of real estate values. If the characteristics and corresponding prices for many houses within a certain city are provided, can the price of a different house in this area be predicted by its characteristics?

As in the case of classification, there are many techniques to develop a regression model for the given problem. The evaluation metric that is routinely calculated to judge the model's performance is the mean squared error (MSE), as given by the following equation:

$$\text{MSE} = \frac{1}{\text{n}} \sum_{\text{i}=1}^{\text{n}} (\text{y} - \text{d})^2$$

where y is the output of the developed regression model, d is the desired output, and n is the total number of the instances.

Overall, in machine learning, there is a trade-off between the goodness of fit and the complexity of the model. In particular, it is a challenge to develop a machine learning model with many variables and free parameters. In such a case, the model becomes complex because it can use these degrees of freedom to always produce a good fit on its training data. On the other hand, if the model has insufficient degrees of freedom for the given task, there will be a degrading influence on its produced fit.

In general, when exposed to more observations, the model improves its predictive performance. However, too much adaptability will force the model to learn the noise within the training data rather than the underlying input/output relations. Therefore, the resultant model overfits the training set, and will not perform equally well on the testing set. Overfitting happens when the model highly adapts to the training set but by doing so fails on the testing set. As Croteau et al. (2017) explain, "overfitting will impact negatively on

the degree of generalization to new data and thus must be avoided in order for solutions to be useful for practical application" (p. 306). An efficient machine learning model attempts to decrease generalization errors and thus have good predictions on data that the model was not trained for. On the other hand, underfitting happens when the model does not capture enough of the inherent structure in the training data, which results in poor performance with both training and testing sets.

1.2 Machine Learning Paradigms

Supervised Learning

Supervised learning is a paradigm of machine learning where the given dataset contains both inputs (independent variables, $\mathrm{x_i}$) and desired outputs (dependent variable, y). The objective of supervised learning is to develop an association model (f) that relates the inputs to the outputs, and can predict the output for future inputs, as clarified in the following equation.

$$\mathrm{y} = \mathrm{f}(\mathrm{x_i})$$
$$\mathrm{i} = 1, ..., \mathrm{n}$$

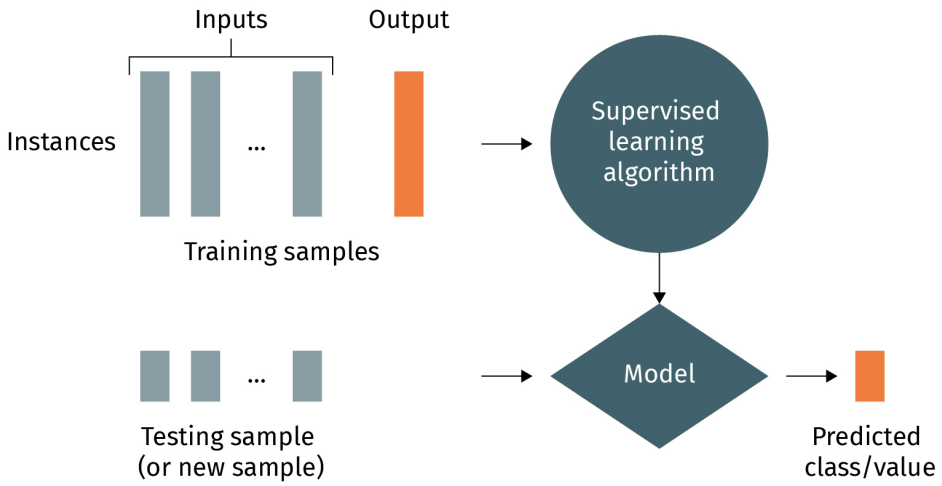
where n is the total number of the variables (i.e., characteristics) in the data samples.

The function (f) is an estimation of the possible output for the underlying $(\mathrm{x_i})$ variables. In classification, the output (y) belongs to a set of finite and discrete values that define the predicted classes. In regression, the output (y) belongs to a range of infinite continuous values that define the numerical outcome(s). The model utilizes the training set to compute the best fit.

During the learning of the model, its parameters are continuously updated until the optimum setting is achieved. This parameter updating process is governed by a specific loss function, and the objective is to adjust the parameters such that this loss function is minimized. For regression problems, this loss function can be the MSE, and for classification problems, the loss function can be the number of wrongly classified instances.

The structure of the supervised learning procedure is shown in the following figure. Here, the instances of the training set are used to learn the model through one of the available classification/regression algorithms. Afterwards, the model is applied to predict the output for the testing samples, which were not presented during this model's training. If the computed output is within an acceptable threshold from the desired output, then the model is employed to do the prediction task for future instances. Otherwise, the learning process must be repeated.

Figure 4: Supervised Learning Structure



Source: Walid Hussein, 2020.

There are many applications where supervised learning is implemented, as seen in the following table.

Table 1: Supervised Learning Examples

Example Dataset	Prediction	Type
Previous home sales	How much is a specific home worth?	Regression
Previous loans that were paid	Will this client default on a loan?	Classification
Previous weeks' visa applications	How many businesspersons will apply for visa next week?	Regression
Previous statistics of benign/ malignant cancers	Is this cancer malignant?	Classification

Source: Walid Hussein, 2020.

Some of the most common supervised learning techniques are listed below.

Table 2: Supervised Learning Techniques

Technique	Obtained Function
Linear classifier, linear regression, multi-linear regression.	Numerical functions

Technique	Obtained Function
Support Vector Machine (SVM), Naïve Bayes, Gaussian discriminant analysis (GDA), Hidden Markov models (HMM).	Parametric Probabilistic functions
K-nearest neighbors, Kernel regression, Kernel density estimation	Non-parametric instance based functions
Decision tree	Non-metric symbolic functions

Source: Walid Hussein, 2020.

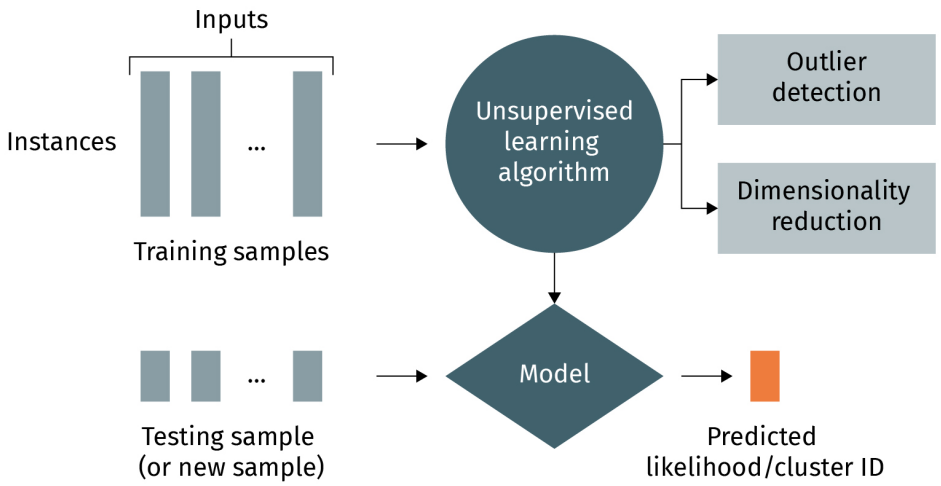
Unsupervised Learning

Suppose you are given a basket with some unlabeled objects, and you plan to arrange the objects by their similarity. You might approach the task by picking a random object and select any interesting physical characteristic of it, such as its surface shape. Afterwards, you will pick all other objects that look similar in shape to the initially picked object, and group them together. You will then repeat the process until all objects are clustered into groups. This process belongs to the field called unsupervised learning because you do not know the desired categories for any of the given objects beforehand.

Unsupervised machine learning is implemented for the problems with unlabeled datasets. Thus, the provided dataset consists of inputs (independent variables, x_i) while the output (dependent variable, y) is not known. This setting is common in practice since the acquisition of labels is typically expensive in big data applications. The aim of unsupervised learning is to discover the salient patterns within the given inputs, which results in **dimensionality reduction**, and/or clustering the data instances into groups according to their relative similarity. The structure of unsupervised machine learning is shown in the following figure.

Dimensionality reduction
This is the process of removing the irrelevant variables from the dataset.

Figure 5: Unsupervised Learning Structure



Source: Walid Hussein, 2020.

Therefore, unsupervised learning models learn to represent the inputs by extracting statistical structure that is intrinsic to the overall collection of data instances. Unsupervised learning can also be used to find the best independent variables that can be useful for either visualizing or clustering the dataset. Furthermore, unsupervised learning also encompasses the identification of data points with high difference to the typical data points in a given dataset (i.e., **outliers**).

Outlier
This term refers to those data records which are far from the bulk of the input dataset. There are true outliers and fake outliers. The former are actual results of the data generating process while the latter stem from measurement, recording, or data storage errors.

The cost function in an unsupervised learning model can be the minimum quantization error, minimum distance between similar data instances, or the maximum likelihood estimation of the correct cluster.

Some examples of unsupervised learning applications can be seen in the following table.

Table 3: Unsupervised Learning Examples

Example dataset	Discovered patterns	Type
Customers profiles	Are these customers similar?	Clusters
Previous transactions	Is a specific transaction odd?	Anomaly detection
Previous purchasing	Are these products purchased together?	Association discovery

Source: Walid Hussein, 2020.

Some common unsupervised learning techniques that are routinely implemented are given in the following table.

Table 4: Unsupervised Learning Techniques

Technique	Description
K-Means, hierarchical clustering	Clustering analysis
Gaussian mixture model (GMM), graphical models	Density estimation
DBSCAN	Outlier detection
Principal component analysis, factor analysis	Dimensionality reduction

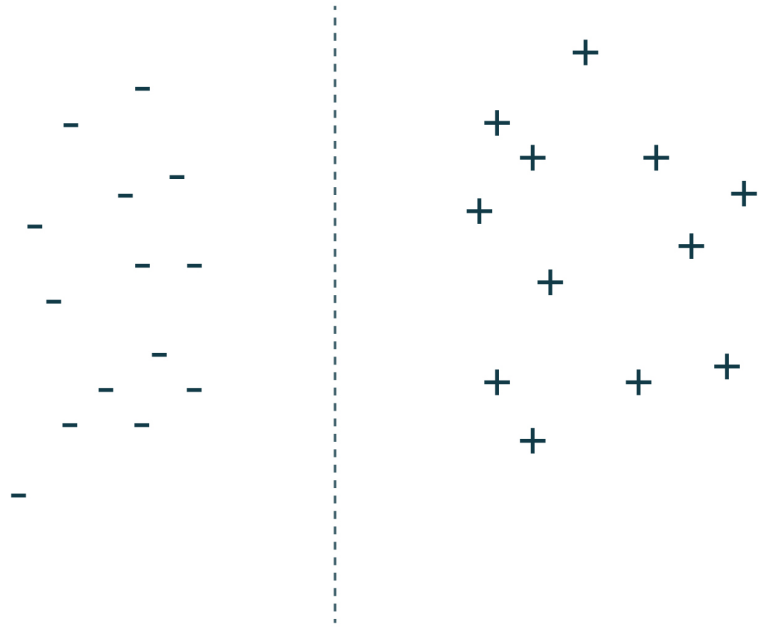
Source: Walid Hussein, 2020.

Semi-Supervised Learning

Semi-supervised machine learning is implemented for datasets where the output (dependent variable, y) is given for only a few instances of the inputs (independent variables, x_i). Therefore, semi-supervised learning is a mix of the unsupervised learning and supervised learning paradigms, and combines the properties of both.

In semi-supervised learning, a basic classification model is designed on the few labeled data instances, which is called the semi-supervised classification step, as shown in the following figure.

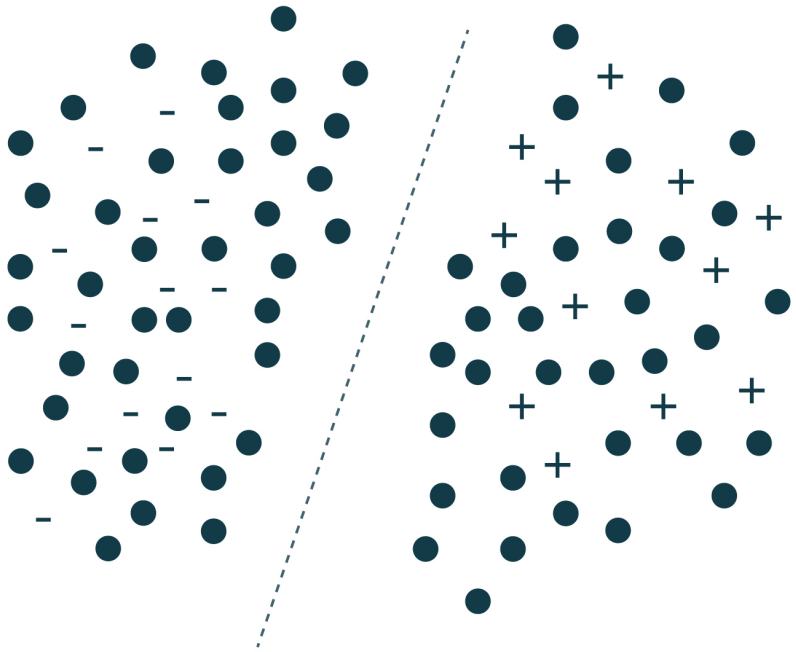
Figure 6: Semi-Supervised Learning (Classification Step)



Source: Walid Hussein, 2020.

A semi-supervised clustering step is then performed, where the model is tuned up to operate without supervision on the remaining large unlabeled data instances, and assigns them to the classes from the first step. Thus, the second step of semi-supervised learning is to transfer labels to all samples through their similarity to originally labeled data points, as shown in the following figure.

Figure 7: Semi-Supervised Learning (Clustering Step)



Source: Walid Hussein, 2020.

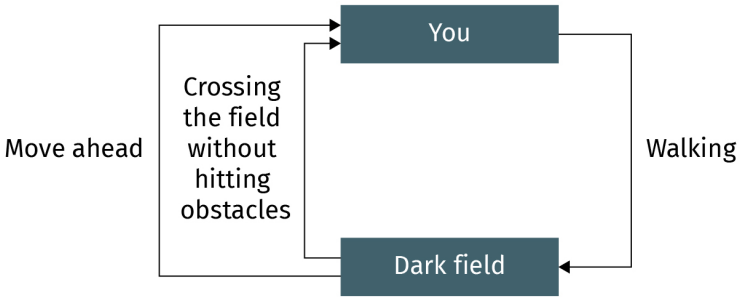
The advantage is that a lot of effort and computational cost are saved because collecting and labeling large datasets can be very expensive. Furthermore, the patterns and the similarities among the data instances are discovered, which brings more insight into the dataset structure.

A popular application for semi-supervised learning is in the field of medical imaging (Chebli et al., 2018). An expert can label a small number of radiography scans of a specific disease and forego the time-intensive task to label all the scans. The developed semi-supervised learning model will automatically group the rest of the scans.

1.3 Reinforcement Learning

If you need to cross a dark field without hitting any obstacles, a naïve approach would be to start walking from your starting position until you hit an obstacle, then you return to your initial place and restart walking while trying to detour just before the obstacle you had previously hit, as described in the following figure.

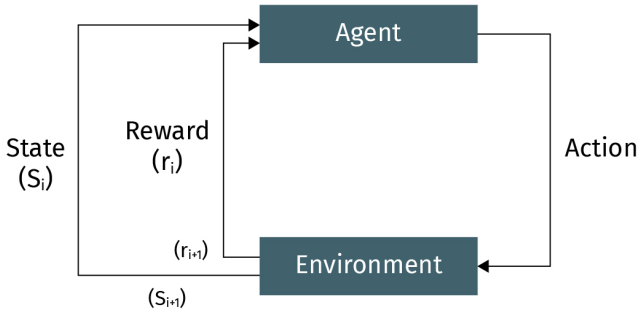
Figure 8: Reinforcement Learning Example



Source: Walid Hussein, 2020.

In this task, you are the agent, the field is the environment, and your walking is the action. Crossing the field without hitting an obstacle is the reward (r) you gain. At each iteration (i), you start at the initial place with a new state (S) and step ahead until you hit a new obstacle. You are closer to crossing the field than the previous state, which means a higher reward is achieved, as shown in the following figure.

Figure 9: Reinforcement Learning Terms



Source: Walid Hussein, 2020.

The process will be repeated until you learn how to cross the field to the other side without hitting any obstacles, hence, achieving the maximum reward. The above learning operation is called reinforcement learning, which is a learning approach for sequential decision making. The model is simply given a goal to achieve, and it learns how to achieve that goal by trial-and-error interactions with its environment.

The ingredients (i.e., building blocks) of a reinforcement learning setting are concluded in the following figure.

Figure 10: Reinforcement Learning Ingredients

Agent	hypothetical entity that performs actions in an environment to gain some reward
Action (A)	all the possible moves that the agent can take
Environment (E)	scenario that the agent has to face
State (S)	current situation returned by the environment
Reward (R)	an immediate return sent back from the environment to evaluate the last action by the agent
Policy (π)	strategy that the agent employs to determine the next action based on the current state
Value (V)	the expected long-term return of the current state under policy

Source: Walid Hussein, 2020.

Reinforcement learning accomplishes machine learning in a different way from both the supervised and unsupervised learning paradigms. It follows the“cause and effect” method because the developed reinforcement model learns by performing an action in the environment that result in a maximum reward over a time horizon. The actions are adjusted until a maximum reward is achieved.

Both supervised and reinforcement learning map from the inputs to the outputs in their respective models, but in reinforcement learning there is a reward function that acts as a feedback to the agent as opposed to the set-up in supervised learning. On the other hand, in unsupervised learning, there is no mapping from the inputs to outputs, but the learning algorithm is to find the underlying patterns in the dataset.

The widely applied approach to perform a reinforcement learning is the model-based approach. In the model-based approach, the agent learns a model using few interactions with the environment. In particular, if the agent starts at state (S_1) and will take action (a_1), they will reach state (S_2) and gain a reward (r_2). By moving further, taking an action (a_2), the output will be (S_3) and (r_3). Based on this collection of experiences, a generalized supervised model can be deduced to relate the current inputs (S_i,a_i) to their consequent outputs (S_{i+1},r_{i+1}).

The advantage of the model-based approach is the speed-up of the learning phase because there is no need to wait for the environment to respond, nor to reset the learning process to its initial state.

Well-known applications of reinforcement learning are game playing and robotic movement control. In game playing, the objective is to succeed in a certain number of moves, and to achieve a score. However, if a wrong action is performed, loss of score is incurred or the game is over. In robotic movement control, a series of actions may be taken for a robot to travel between two points. At every step, the robot moves closer to the other point, which constitutes the reward.

SUMMARY

This unit introduces machine learning and its relevant definitions and concepts. Machine learning is an inductive process that automatically builds a prediction model or extracts relevant patterns by learning the natural structure of a given dataset. The output of the resulting model is a discrete value, in classification problems, and a continuous value, in regression problems. Meanwhile, if the dataset is not labeled with an output variable, the machine learning objective in such cases is to recognize the salient patterns, e.g., by applying clustering analysis.

There are four paradigms of machine learning, which are supervised, unsupervised, semi-supervised, and reinforcement learning. Each paradigm of machine learning is characterized by the kinds of data they require, the kinds of output they generate, and the different types of algorithms used to arrive at the learned model.

UNIT 2

CLUSTERING

STUDY GOALS

On completion of this unit, you will have learned …

- what is meant by clustering.
- common applications of clustering analysis.
- the different methods for clustering analysis.
- the implementation of clustering methods in Python.
- the advantages and limitations of each clustering method.

2. CLUSTERING

Introduction

Unsupervised learning
This term denotes the machine learning task of pattern or structure discovery in unlabeled datasets.

As discussed in the previous unit, **unsupervised learning** addresses use cases where no labels or desired outcomes are given. Thus, we can either cluster the data to reveal meaningful partitions and/or hierarchies, or to find association rules that interrelate the involved data's variables (e.g., how frequently two products (x and y) are bought together).

Clustering means to gather data records into natural groups (i.e., clusters) of similar samples according to predefined similarity/dissimilarity metrics, which results in extracting a set of useful information about the given dataset. The contents of any cluster should be similar to each other, which is called high intra-cluster similarity. Conversely, the contents of any cluster should be very different (i.e., dissimilar) from the contents of other clusters, which is called high inter-cluster separation. The similarity/dissimilarity metric that is routinely utilized in clustering analysis is a form of distance function between each pair of data records (e.g., A and B). Therefore, the distance measures how close A and B are to each other, and a decision is made whether to combine A and B in one cluster.

There are two commonly implemented simple forms of distance function, which are the Euclidean distance and the Manhattan distance. The Euclidean distance for two dimensional datasets (i.e., having two features) is calculated as

$$\mathrm{d}_{\mathrm{A,B}} = \sqrt{(\mathrm{x_A} - \mathrm{x_B})^2 + (\mathrm{y_A} - \mathrm{y_B})^2}$$

while the Manhattan distance is calculated as

$$\mathrm{d}_{\mathrm{A,B}} = |\mathrm{x_A} - \mathrm{x_B}| + |\mathrm{y_A} - \mathrm{y_B}|$$

Feature
This is a measurable and observable quantity about the data records.

where $(\mathrm{x_A}, \mathrm{y_A})$ and $(\mathrm{x_B}, \mathrm{y_B})$ are the coordinates (i.e., **features**) of data records A and B, respectively. d is the value representing the distance between the two data records.

It is worth mentioning that the features of the datasets that have scales with widely differing ranges should be standardized to the same scale before progressing in any form of clustering analysis.

The clustering evaluation is usually completed by manual inspection of the results, benchmarking on existing labels, and/or by distance measures to denote the similarity level within a cluster and the dissimilarity level across the clusters.

Clustering analysis
This type of unsupervised learning sorts a set of data records into clusters or groups.

Clustering analysis is applied in many fields including pattern recognition, image processing, spatial data analysis, bioinformatics, crime analysis, medical imaging, climatology, and robotics. One noteworthy area of clustering applications is market segmentation, which focuses on grouping customers into clusters of different characteristics (e.g., pay-

ment history, customers' interests), and using them for targeted marketing. Another common application is to implement clustering analysis to develop a recommendation system, for example, to cluster similar documents together or to recommend similar songs/ movies (Das et al., 2014).

Clustering analysis is an active area of research, and many individual models have been developed. Nevertheless, there are some main classes of methods and techniques that can be distinguished based on the broad approach to how the grouping of the data records is performed, as listed below:

1. Centroid-based clustering methods
2. Gaussian mixtures models clustering methods
3. Hierarchical clustering methods
4. Density based clustering methods

In the following sections, a detailed explanation of each of these methods will be presented.

2.1 Centroid-Based Clustering

Centroid-based clustering searches for a pre-determined number of clusters within an unlabeled and possibly multidimensional dataset. Each data record is assigned to one, and only one, cluster (i.e., there are no intersections between clusters). The rule is that the distance between a data record and each of the cluster's **centroids** is calculated, and this data record is assigned to the cluster achieving the minimum distance.

Centroid
The arithmetic average position of all the points is called a centroid. It may be located at a given point (real centroid) or at a new placement (imaginary centroid).

K-Means Clustering

K-means clustering is a centroid-based clustering approach commonly used in practice. The method partitions (n) data records from the given dataset into (K) clusters. The approach consists of three main steps: initialization, assignment, and update step.

In the initialization step, the number of clusters are assumed (i.e., K is predetermined), and the centroid of each cluster is randomly defined. The simple procedure to define the initial placement of a cluster's centroid is to locate it at one of the given data records. Note that it is prudent to locate the centroids far from each other to span the whole domain of the data records.

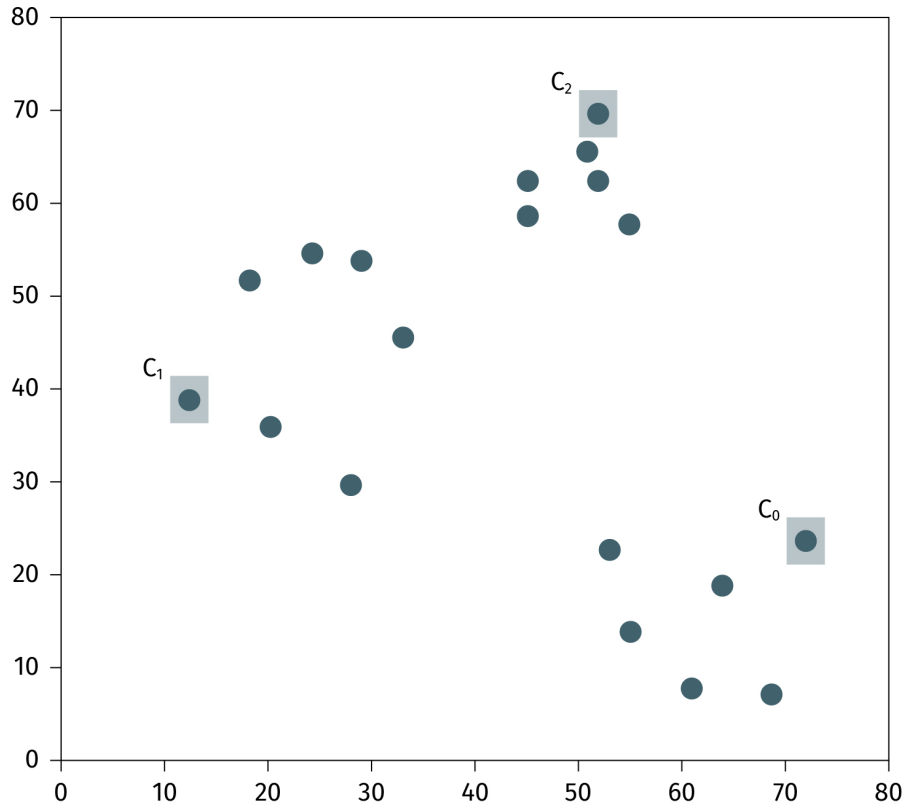
In the assignment step, the clusters are formed by connecting each data record with its nearest centroid. Afterwards, in the update step, a more accurate centroid of each cluster is calculated as the mean point of its included data records. Then, the assignment and update steps are repeated until convergence, when there are no changes in the calculated clusters' centroids.

To further explain the k-means algorithm, we will apply it to a simple dataset of 19 data records and two features, shown below.

$$x: [12, 20, 28, 18, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53, 55, 61, 64, 69, 72]$$
$$y: [39, 36, 30, 52, 54, 46, 55, 59, 63, 70, 66, 63, 58, 23, 14, 8, 19, 7, 24]$$

This dataset can be displayed in a Cartesian plane:

Figure 11: Cartesian Plane



Source: Walid Hussein, 2020.

In the initialization step, we will assume three clusters for this dataset (i.e., $K = 3$), and select their initial centroids to be $C_0 = \{72,24\}$, $C_1 = \{12,39\}$, and $C_2 = \{52,70\}$.

Hence, the distances between a data record and these three centroids are calculated using the Euclidean distance, as shown in the following table.

Figure 12: Euclidean Distance Calculation

#	Data record		Distance to C_0	Distance to C_1	Distance to C_2
	x	y			
1	12	39	61.85	0.00	50.61
2	20	36	53.37	8.54	46.69
3	28	30	44.41	18.36	46.65
4	18	52	60.83	14.32	38.47
5	29	54	52.43	22.67	28.02
6	33	46	44.78	22.14	30.61
7	24	55	57.14	20.00	31.76
8	45	59	44.20	38.59	13.04
9	45	63	47.43	40.80	9.90
10	52	70	50.16	50.61	0.00
11	51	66	46.96	47.43	4.12
12	52	63	43.83	46.65	7.00
13	55	58	38.01	47.01	12.37
14	53	23	19.03	44.01	47.01
15	55	14	19.72	49.74	56.08
16	61	8	19.42	57.98	62.65
17	64	19	9.43	55.71	52.39
18	69	7	17.26	65.37	65.25
19	72	24	0.00	61.85	50.16

Source: Walid Hussein, 2020.

From the calculated distances, each data record is assigned to its closest centroid as highlighted in orange. Therefore, there are sixn data records grouped in the first cluster, seven data records grouped in the second cluster, and six data records grouped in the third cluster.

Now, we reach the update step, where the new centroid of each cluster is calculated as the mean of the included data records, as shown below:

$$C_0' = \left\{\frac{\sum 53+55+61+64+69+72}{6}, \frac{\sum 23+14+8+19+7+24}{6}\right\}$$
$$= \{62.33,\ 15{,}83\}$$
$$C_1' = \left\{\frac{\sum 12+20+28+18+29+33+24}{7}, \frac{\sum 39+36+30+52+54+46+55}{7}\right\}$$
$$= \{23.42,\ 44.57\}$$
$$C_2' = \left\{\frac{\sum 45+45+52+51+52+55}{6}, \frac{\sum 59+63+70+66+63+58}{6}\right\}$$
$$= \{50,\ 63.16\}$$

Since the new centroids differ from the old ones, a second iteration must be performed through the assignment step. Therefore, the distance of each data record is calculated to the new centroid of each cluster to investigate whether a data record needs to move to a different cluster. These distance calculations and the new assignment of clusters (if any) are given in the following table.

Figure 13: New Euclidian Distance Calculations

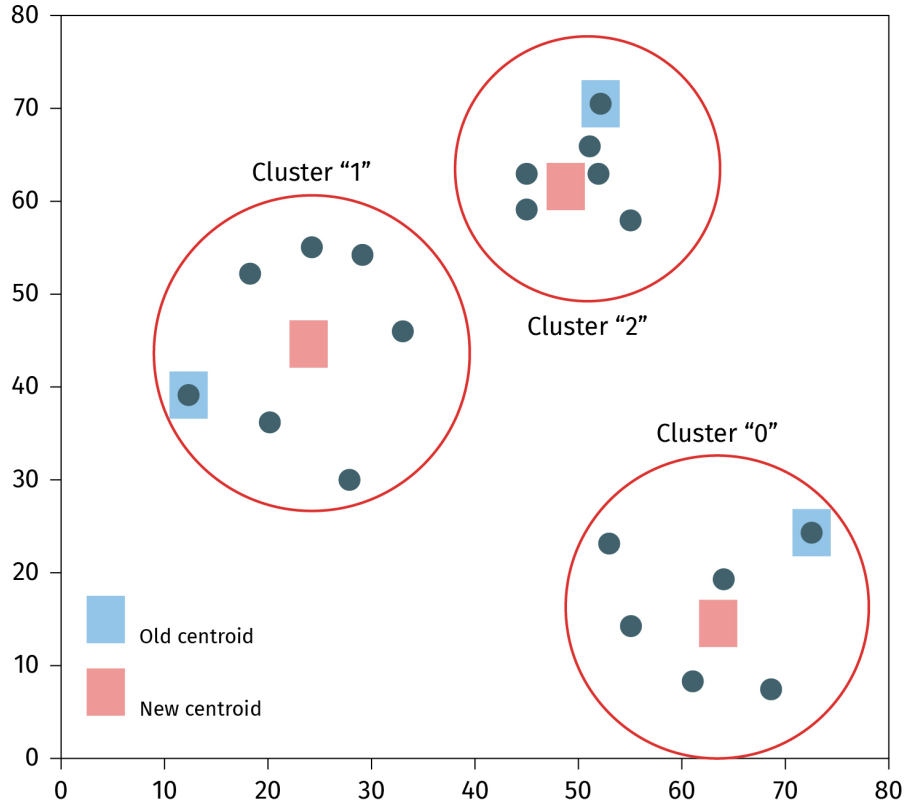
#	Data record		Distance to C_0'	Distance to C_1'	Distance to C_2'
	x	y			
1	12	39	55.41	12.71	45.03
2	20	36	46.89	9.23	40.47
3	28	30	37.14	15.27	39.79
4	18	52	57.21	9.20	33.89
5	29	54	50.67	10.96	22.91
6	33	46	42.08	9.69	24.16
7	24	55	54.80	10.45	27.25
8	45	59	46.52	25.96	6.50
9	45	63	50.25	28.38	5.00
10	52	70	55.15	38.26	7.13
11	51	66	51.43	34.93	3.01
12	52	63	48.29	34.01	2.01
13	55	58	42.80	34.32	7.19
14	53	23	11.77	36.61	40.27
15	55	14	7.55	43.95	49.41
16	61	8	7.94	52.44	56.25
17	64	19	3.58	47.96	46.33
18	69	7	11.07	59.07	59.29
19	72	24	12.66	52.76	44.92

Source: Walid Hussein, 2020.

As seen in the above table, although the centroids are changed, the contents of each cluster remain as they are. This implies that there is no need for an update step nor any further iterations because there will be no modification to the recently calculated centroids.

Hence, we decide that convergence is successfully obtained and the k-means model is determined with its three clusters centered at $C_0' = \{62.33, 15.83\}$, $C_1' = \{23.42, 44.57\}$, and $C_2' = \{50, 63.16\}$, respectively, as described in the following figure.

Figure 14: K-Means Example (Centroids Updated)



Source: Walid Hussein, 2020.

In the above figure, the centroid represents the mean of all the data records that belong to that cluster. In addition, the observations that belong to a given cluster are closer to its centroid, in comparison to the centroids of the other clusters.

The resulting k-means model can be applied to predict the associated cluster(s) of some test data records. For example, if you need to know the cluster of a data record (p) with $(\mathrm{x} = 20$, and $\mathrm{y} = 20)$, you first calculate its Euclidean distance to each cluster's centroid as follows:

$$\mathrm{d}_{\mathrm{p},\,\mathrm{C}'_0} = \sqrt{(20 - 62.33)^2 + (20 - 15.83)^2} = 42.53$$
$$\mathrm{d}_{\mathrm{p},\,\mathrm{C}'_1} = \sqrt{(20 - 23.42)^2 + (20 - 44.57)^2} = 24.81$$
$$\mathrm{d}_{\mathrm{p},\,\mathrm{C}'_2} = \sqrt{(20 - 50)^2 + (20 - 63.16)^2} = 52.56$$

Therefore, the data record (p) is assigned to cluster "1."

For another data record (q) with ($\mathrm{x} = 60$, and $\mathrm{y} = 40$), the distances are:

$$\mathrm{d}_{\mathrm{q},\,\mathrm{C}_0'} = \sqrt{(60 - 62.33)^2 + (40 - 15.83)^2} = 24.28$$
$$\mathrm{d}_{\mathrm{q},\,\mathrm{C}_1'} = \sqrt{(60 - 23.42)^2 + (40 - 44.57)^2} = 36.86$$
$$\mathrm{d}_{\mathrm{q},\,\mathrm{C}_2'} = \sqrt{(60 - 50)^2 + (40 - 63.16)^2} = 25.23$$

Therefore, the data record (q) is assigned to cluster"0."

K-Means in Python

In this section, we will implement the k-means algorithm to build a clustering model and fit it using a given dataset. Next, the model will be employed to predict the associated cluster(s) of new data records.

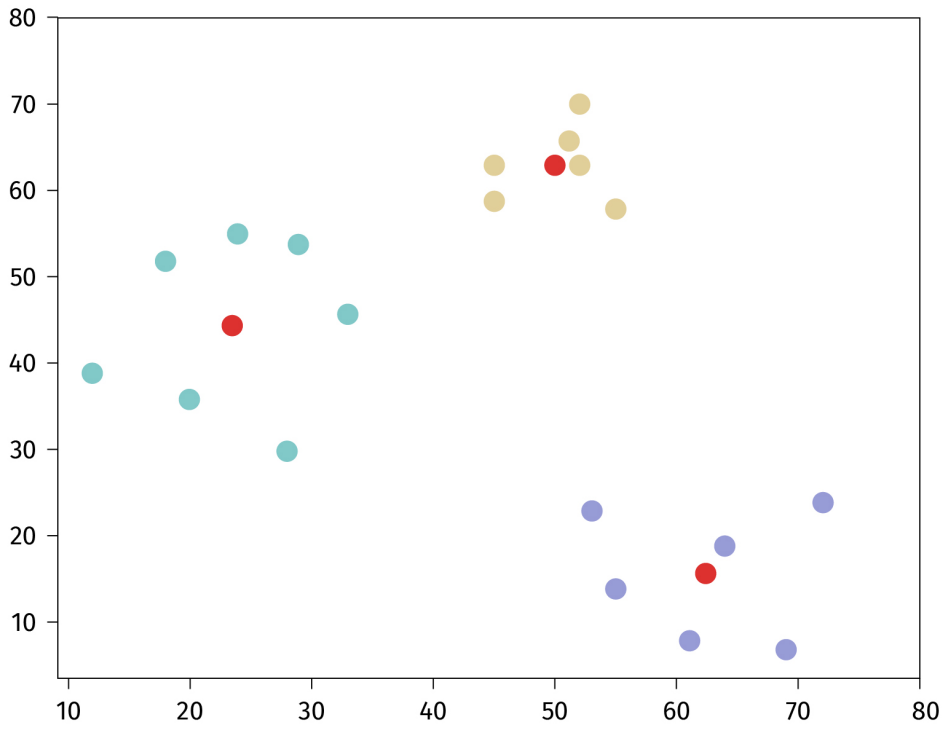
The working example will be the same dataset, which was utilized to explain the k-means algorithm.

Code

```
>>> import pandas as pd        # For reading datasets
>>> import numpy as np        # For computations
>>> import matplotlib.pyplot as plt  # For visualization
>>> from pandas import DataFrame   # For creating data frame
>>> from sklearn.cluster import KMeans
>>> Data={'x': [12, 20, 28, 18, 29, 33, 24, 45, 45, 52, 51, 52, 55, 53,
55, 61, 64, 69, 72],'y': [39, 36, 30, 52, 54, 46, 55, 59, 63, 70, 66, 63,
58, 23, 14, 8, 19, 7, 24]}
>>> df = DataFrame(Data,columns=['x','y'])

# Create and fit the KMeans model
>>> kmeans = KMeans(n_clusters=3).fit(df)

# Find the centroids of the clusters
>>> centroids = kmeans.cluster_centers_
>>> print(centroids)
[[62.33333333 15.83333333]
 [23.42857143 44.57142857]
 [50.    63.16666667]]
# Get the associated cluster for each data record
>>> kmeans.labels_
array([1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0])

# Display the clusters contents and their centroids
>>> plt.scatter(df['x'], df['y'], c= kmeans.labels_.astype(float),
s=50, alpha=0.5)
<matplotlib.collections.PathCollection object at 0x0000005D3E70F940>
```

```
>>> plt.scatter(centroids[:, 0], centroids[:, 1], c='red', s=50)
<matplotlib.collections.PathCollection object at 0x0000005D3E70FDD8>
>>> plt.show()
```

Figure 15: K-Means in Python



Source: Walid Hussein, 2020.

Code

```
# Note that the centroid of each cluster is displayed in red

# Use the model to predict the cluster(s) of another input(s)
>>> kmeans.predict([[20, 20], [60, 40]])
array([1, 0])
```

The default function of KMeans() in Python is written with its main parameters as follows:

Code

```
KMeans(n_clusters=8,init='kmeans++',n_init=10, max_iter=300,
tol=0.0001, precompute_distances='auto')
```

The included parameters above are explained in the following table.

Table 5: K-Means Parameters in Python

Parameter	Value	Description
n_clusters	int, optional, default: 8	The number of clusters and centroids.
init	init : {'k-means++', 'random' or an ndarray}	Method for initialization, defaults to 'k-means++', which selects initial centroids in a smart way to speed up convergence. 'random': choose k rows at random from data as initial centroids. If an ndarray is passed, it should be of the form (n_clusters, n_features)
n_init	int, default: 10	Number of time the k-means algorithm will be run with different centroid seeds.
max_iter	int, default: 300	Maximum number of iterations of the k-means algorithm for a single run.
tol	float, default: 1e-4	Relative tolerance with regards to declaring convergence.
precompute_distances	precompute_distances : {'auto', True, False}	'auto' : do not precompute distances if n_samples * n_clusters > 12 million. This corresponds to about 100MB overhead per job using double precision. True : always pre-compute distances. False : never pre-compute distances.

Source: Walid Hussein, 2020.

Limitations of K-Means

The k-means clustering technique is relatively simple to employ, suitably efficient for large datasets (many data points), assures convergence, can easily assume the initial centroids' placements, and smoothly adjusts the clusters' outlines when new data records are introduced.

However, the user must first determine the number of the clusters, although it is significantly difficult to predict. Different initial partitions can result in different final clusters. Moreover, k-means assumes a spherical shape of clusters; therefore, the technique does not work effectively in clustering datasets having regions with differently shaped densities. K-means handles only numerical datasets, and the initial assumption of the centroids may result in poor convergence rate. The normalization of the dataset may also produce completely different results. Moreover, using k-means with the standard Euclidian distance measure is not well-suited for high-dimensional data.

The dataset may be dynamic over time, which adds the burden of checking the optimum number of clusters periodically. Furthermore, fake outliers may not be detected and influence the structures of the clusters.

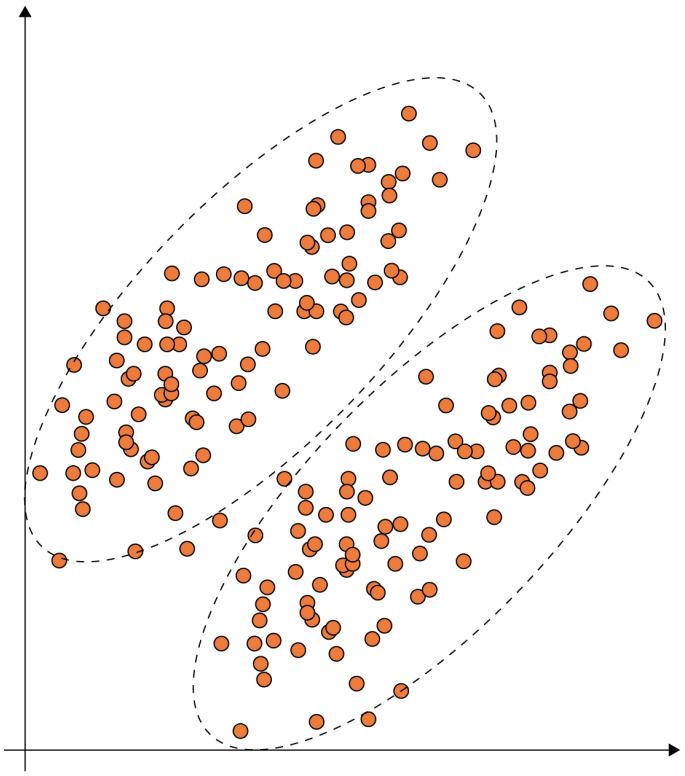
At each iteration of (M) iterations, the computations of the distances among the data records (n) and the centroids (K), will result in a computation complexity of $\mathrm{O}(\mathrm{M} \cdot \mathrm{n} \cdot \mathrm{K})$.

2.2 Gaussian Mixture Models Clustering

Probabilistic generative models
These models explain how the input data are generated in terms of their joint probability distribution.

As described in the previous section, the k-means clustering is performing a“hard” clustering to generate circular clusters centered at the found centroids. Whereas, in Gaussian mixture models (GMM) clustering, each cluster is considered as a **probabilistic generative model**. A data record has a probability for belonging to each cluster, and it is assigned to the cluster returning the highest probability. Thus, it is a way of performing a “soft” clustering, as shown in the following figure.

Figure 16: GMM Clustering



Source: Walid Hussein, 2020.

As in the k-means method, GMM also initially assumes the number of clusters for the input dataset. GMM tries to fit mixtures of Gaussian distributions to the dataset, where each distribution defines one cluster.

An important note, as the above figure implies, is that each cluster in GMM clustering follows a Gaussian distribution with two parameters, which are the cluster’s mean (μ) and standard deviation (σ). However, in k-means clustering, each cluster is circular and deter-

mined by its centroid and radius (to include exactly the assigned data records). In the particular case when the inferred GMM clusters have equal and diagonal covariance matrix (i.e., circular shapes), the result is equivalent to that obtained by k-means clustering.

In general, the task in GMM clustering is to find the optimum values of both μ and σ which maximize the overall probability or (likelihood) of the data records to be in their assigned cluster(s). To achieve this outcome, the expectation maximization algorithm is implemented.

Expectation Maximization Algorithm

The expectation maximization (EM) algorithm is implemented when there is an analytic model for the dataset (i.e., the probabilistic Gaussian mixtures model), the model's shape (i.e., Gaussian distribution) is known, but the parameters μ and σ of this model are unknown. In this setting, the EM algorithm is an iterative method to find the maximum likelihood estimates of these parameters to get the "best fit" model for the input dataset. The iteration process stops when a desired convergence criterion is achieved.

The Gaussian mixtures model (GMM) is defined as a set of K probability distributions, and each distribution corresponds to one cluster (Sammut, 2011). A data record is assigned a membership probability for each cluster (i.e., each data record belongs to each cluster with a certain probability). Afterwards, we can assign a data record to a specific cluster based on the largest probability value.

The EM algorithm alternates consequently between two steps, which are the expectation step (E), and the maximization step (M), as explained below:

1. Assume the number of clusters (K), and guess initial parameter values for each cluster μ_k and σ_k, $k = 1, ..., K$. Additionally, assume a weight for each cluster, i.e., the probability that any given data record is a member of that specific cluster, q_k. One example of a natural initial assumption for the q_k is that they are equal for all clusters. Note that

$$\sum_{1}^{K} q_k = 1$$

2. E step:
   a) Compute the membership probability for each data record $(x_i, i = 1, ..., n)$ in each cluster (k).

$$P(x_i|k) = \frac{1}{\sqrt{(2\pi)^n \sigma_k^2}} \cdot e^{-\frac{(x_i - \mu_k)^T \cdot (x_i - \mu_k)}{2\sigma_k^2}}$$

   b) Construct an expression for the likelihood of all data records (i.e., responsibilities of each distribution for each data point).

$$r_{ik} = \frac{\text{probability that } x_i \text{ belongs to cluster } k}{\text{probability of } x_i \text{ over all clusters}} = \frac{q_k \cdot P(x_i k)}{\sum_{1}^{K} q_k \cdot P(x_i k)}$$

where $k = 1, ..., K$. Hence if x_i is very close to one cluster k, it will get a high r_{ik} value for this cluster and relatively low values otherwise.

3. M step:
   a) For each cluster k, calculate the total likelihood m_k (loosely speaking the fraction of points allocated to cluster k).

$$m_k = \sum_{i=1}^{n} r_{ik}$$
$$k = 1, ..., K$$

   b) Update the initially assumed parameters.

$$q_k{}' = \frac{m_k}{\sum_{1}^{K} m_k}$$

$$\mu_k{}' = \frac{1}{m_k} \sum_{i=1}^{n} r_{ik} \cdot x_i$$

$$\sigma_k{}'^2 = \frac{1}{m_k} \sum_{i=1}^{n} r_{ik} \cdot (x_i - \mu_k{}')^T \cdot (x_i - \mu_k{}')$$

4. Iterate through the two steps until no further changes in the parameters are noted, or when the log-likelihood function, given below, converges.

$$\sum_{i=1}^{N} \ln\left(\sum_{k=1}^{K} q_k{}' \cdot P(x_i k) \right)$$

5. Assign each data record to the cluster with which it has the highest membership probability.

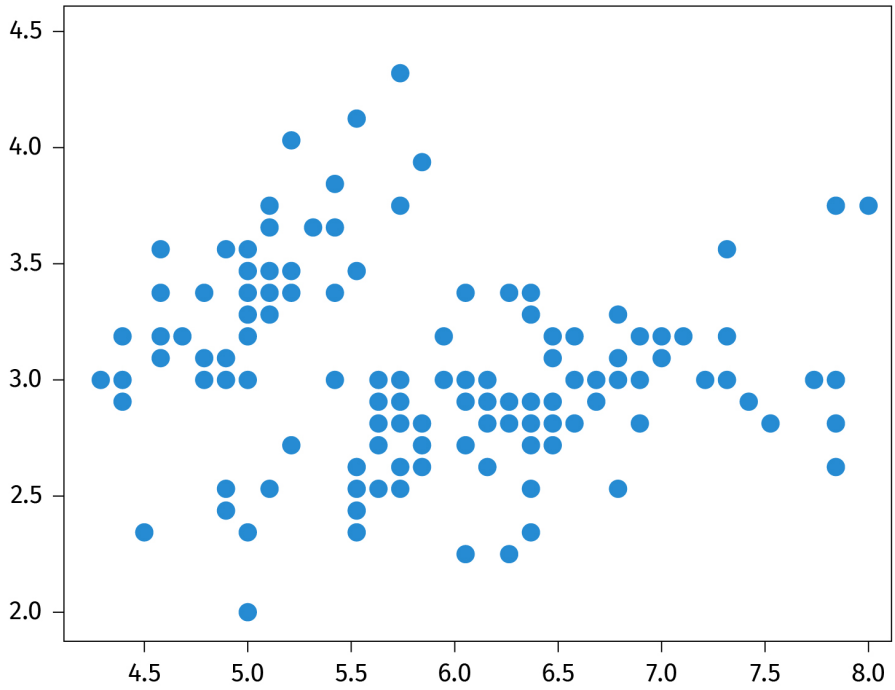
Since the algorithm contains heavy mathematical calculations, we will utilize the power of Python programming to show how EM is applied to solve the GMM clustering problem on the well-known Iris dataset.

Code

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> from pandas import DataFrame
>>> from sklearn import datasets
```

```
>>> from sklearn.mixture import GaussianMixture

>>> # load the iris dataset
>>> iris = datasets.load_iris()

>>> # select first two columns
>>> X = iris.data[:, :2]

>>> # turn it into a dataframe
>>> d = pd.DataFrame(X)

>>> # plot the data
>>> plt.scatter(d[0], d[1])
<matplotlib.collections.PathCollection object at 0x000000211AD03710>
>>> plt.show()
```

Figure 17: EM Applied to GMM Clustering I



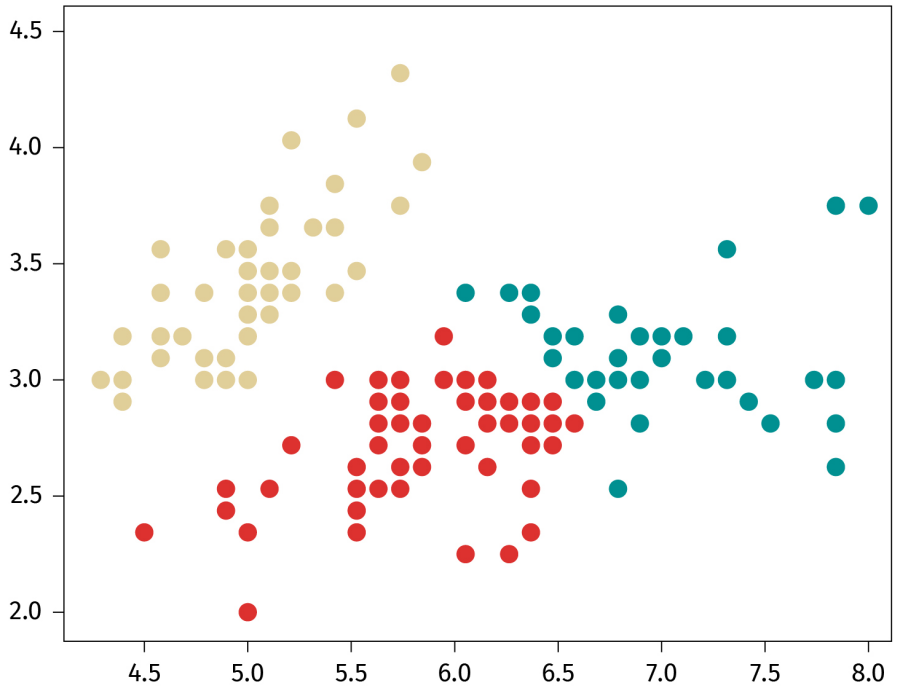
Source: Walid Hussein, 2020.

Code

```
>>> # fit the data as a mixture of 3 Gaussians
>>> gmm = GaussianMixture(n_components = 3)
>>> gmm.fit(d)
```

```
>>> # predict the cluster of each data record
>>> labels = gmm.predict(d)

>>> # Check if the model has converged
>>> print('Converged:',gmm.converged_)
Converged: True

>>> # print the number of iterations needed
>>> print(gmm.n_iter_)
8

>>> # get the final “means” for each cluster
>>> means = gmm.means_
>>> means
array([[5.9009976 , 2.74387546],
   [5.01507898, 3.4514463 ],
   [6.68055626, 3.02849627]])

>>> # get the final “standard deviations” (i.e., covariance matrix)
for each cluster
>>> covariances = gmm.covariances_
>>> covariances
array([[[0.27671149, 0.08897036],
   [0.08897036, 0.09389206]],

[[0.11944714, 0.08835648],
   [0.08835648, 0.11893388]],

[[0.36153508, 0.05159664],
   [0.05159664, 0.08927917]]])

>>> # plot the data records in each clusters in different color
>>> d['labels']= labels
>>> d0 = d[d['labels']== 0]
>>> d1 = d[d['labels']== 1]
>>> d2 = d[d['labels']== 2]
>>> plt.scatter(d0[0], d0[1], c ='r')
<matplotlib.collections.PathCollection object at 0x000000212B208908>
>>> plt.scatter(d1[0], d1[1], c ='yellow')
<matplotlib.collections.PathCollection object at 0x000000212B208EB8>
>>> plt.scatter(d2[0], d2[1], c ='g')
<matplotlib.collections.PathCollection object at 0x000000212B64C278>
>>> plt.show()
```

Figure 18: EM Applied to GMM Clustering II



Source: Walid Hussein, 2020.

Limitations of GMM Clustering

GMM clustering is preferred for datasets consisting of regions that do not look like simple circles, but are in shapes that are more complex (i.e., ellipsoids). It is also the method of choice if it is desirable to know the probability of a data record to belong to each of the clusters. The technique is able to simultaneously optimize a large number of variables, to create both “hard”clusters with crisp 0—1 membership encoding, and the “soft” clusters with soft borders according to the probability level (i.e., membership) of each data record.

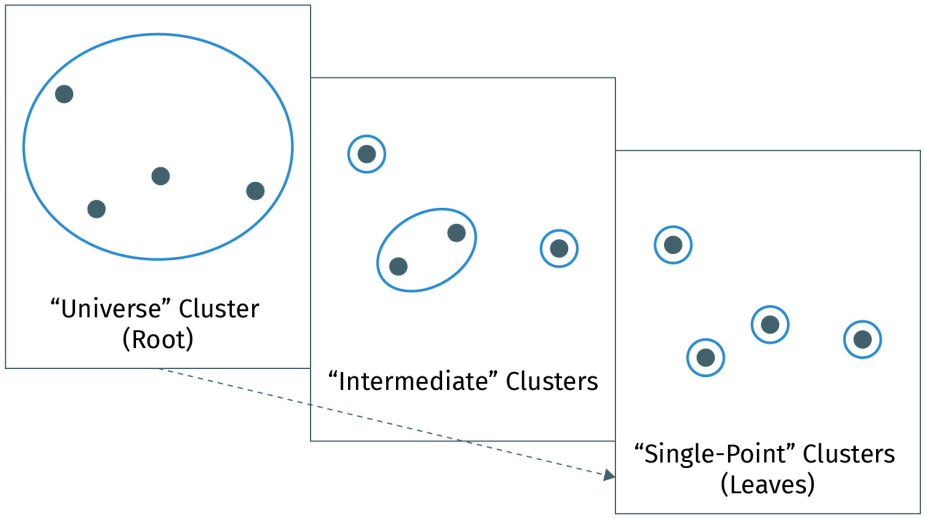
On the other hand, the EM algorithm can be slow because it is time consuming to find the probability distribution for each data record. It can also get stuck in a local maximum of the log-likelihood function. The steps, in general, suffer from heavy computations of the conditional probabilities.

2.3 Hierarchical Clustering

The k-means algorithm must decide the number of clusters at its first step. This is a challenge which hierarchical clustering solves.

As the name implies, in hierarchical clustering, the clusters are built in a hierarchy. This hierarchy of clusters is represented as a tree (or dendrogram). The root of this tree is the "universe" cluster that includes all the data records, while the leaves form "single-point" clusters, where they include an individual data record for each leaf. From the leaves to the root, the most similar "single-point" clusters are combined to form larger clusters, and the process is repeated until reaching the "universe" cluster, as described in the following figure.

Figure 19: Hierarchical Clustering



Source: Walid Hussein, 2020.

There are two types of hierarchical clustering: agglomerative and divisive. Agglomerative clustering is a bottom-up approach that starts at the "single-point" clusters and moves up by merging similar clusters until it reaches the "universe" cluster. On the other hand, divisive clustering works the other way around, as a top-down approach.

In this section, we will explain in detail the agglomerative clustering algorithm, as given in the following steps:

1. Consider each data record as a cluster (i.e., "single-point" cluster). The number of clusters is equal to n, which is the number of data records within the input dataset.
2. Merge the two closest clusters into one bigger cluster. The number of clusters will become $(\mathrm{n}-1)$.
3. Repeat step two until a single cluster is formed: the "universe" cluster.
4. Construct a tree (i.e., dendrogram) to visualize the progression of the formed clusters at each step.

This raises the question of how to decide whether two “single-point” clusters are close to each other. The answer is the concept of the proximity matrix, which includes all the distances from one data record to the other data records. Therefore, the proximity matrix is $\mathrm{n} \times \mathrm{n}$ matrix of distances, where the minimum off-diagonal value in this matrix defines the closest data records. The distance itself can be Euclidean, Manhattan, or any other suitable distance measure.

For clusters that contain more than one data record, there are few options that are routinely implemented to define the distance between two clusters of this type. These are

- Determine the distance between the closest points of these two clusters (this is called single linkage).
- Determine the distance based on the arithmetic means of the distances between the objects of these two clusters (this is called average linkage).
- Determine the distance between the farthest points of these two clusters (this is called complete linkage).

Let us apply the bottom-up agglomerative clustering algorithm to a simple example to cluster five students according to their grades math and physics. Therefore, the dataset contains two features which are the students' grades in these subjects, x and y, respectively, as given in the following table.

Table 6: Grades-Based Clustering

Student ID	x	y
1	10	12
2	7	10
3	28	27
4	20	22
5	35	33

Source: Walid Hussein, 2020.

For instance, the Euclidean distance between the student with “ID = 4” and the student with “ID = 5” is calculated by

$$\mathrm{d}_{4,5} = \sqrt{(20-35)^2 + (22-33)^2} = 18.6$$

Hence, the proximity matrix for the Euclidean distances between all the data records is created as follows:

Table 7: Proximity Matrix

ID	1	2	3	4	5
1	0.00	3.61	23.43	14.14	32.65
2	3.61	0.00	27.02	17.69	36.24
3	23.43	27.02	0.00	9.43	9.22
4	14.14	17.69	9.43	0.00	18.60
5	32.65	36.24	9.22	18.60	0.00

Source: Walid Hussein, 2020.

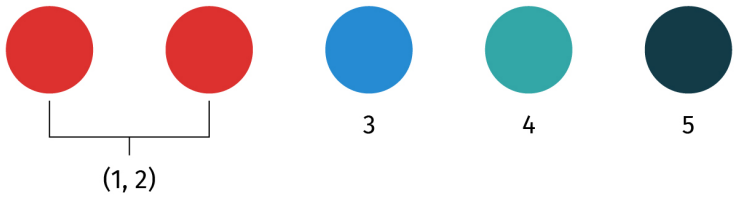
In the first step of the algorithm, we assign each data record to a “single-point” cluster, as shown below.

Figure 20: Single-Point Cluster

Source: Walid Hussein, 2020.

The second step is to combine the closest two data records into one cluster (Cluster 1), which are data record number 1 and data record number 2, because they have the minimum value (3.61) as seen in the proximity matrix.

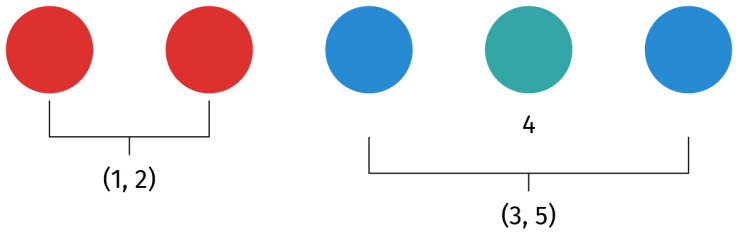
Figure 21: Cluster 1



Source: Walid Hussein, 2020.

The third step is to combine the next closest data records (or clusters). The single linkage approach, explained above, is used to determine the distance between a cluster and all the other data records. Hence, the next minimum distance is 9.22, which is between data record number 3 and data record number 5; therefore, they are combined in one cluster (Cluster 2).

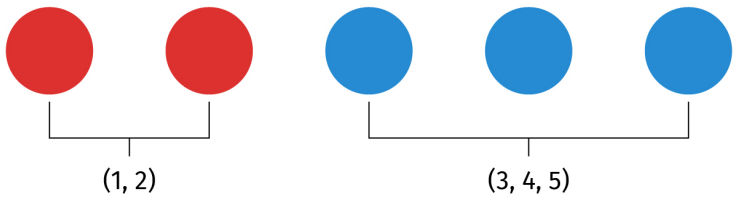
Figure 22: Cluster 2



Source: Walid Hussein, 2020.

The fourth step is to continue calculating the next minimum distance between the data records (or clusters), which is the distance with a value of 9.43, found between data record number 3 and data record number 4. Note that the data record number 3 is already an element in Cluster 2; therefore, the data record number 4 is combined with Cluster 2 forming a larger cluster (Cluster 3).

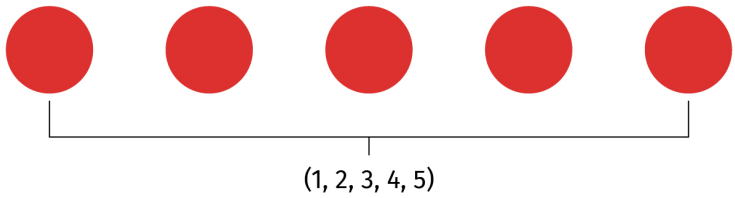
Figure 23: Cluster 3



Source: Walid Hussein, 2020.

The next minimum distance is 14.14 between data record number 1 (element of Cluster 1) and data record number 4 (element of Cluster 3). A larger cluster (Cluster 4) is then formed by combining clusters 1 and 3. Since all the data records are included in this new cluster, this cluster is the “universe” cluster.

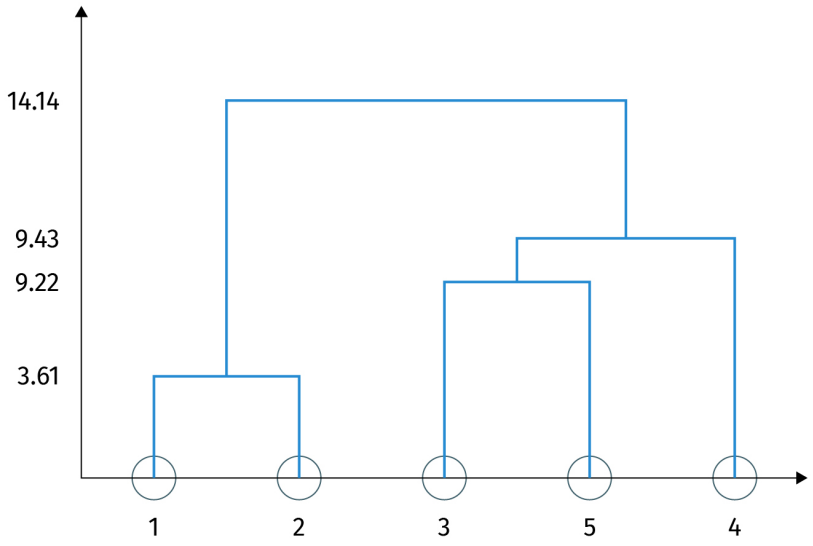
Figure 24: Universe Cluster



Source: Walid Hussein, 2020.

Finally, the tree (i.e., dendrogram) is constructed as a two dimensional graph with x-axis for the data records of the input dataset, and the y-axis for the recorded distance between the combined data records (or clusters) at each step, as shown in the following figure.

Figure 25: Dendrogram

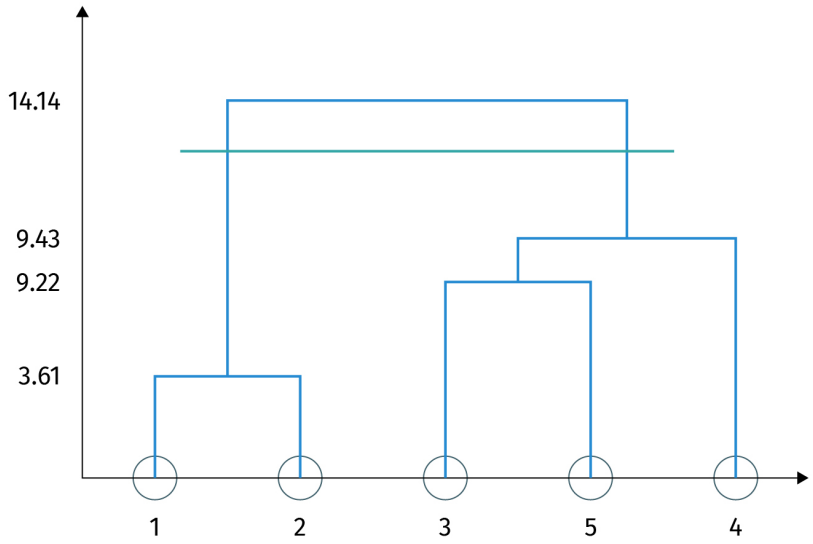


Source: Walid Hussein, 2020.

The dendrogram is an important visualization of the hierarchical clustering algorithm. It is not only a tool to show how the clusters progressed from the "single-point" clusters to the "universe" cluster, but it also aids the decision for the optimum number of clusters in the given dataset. The greater the length of the vertical lines in the dendrogram, the higher the distance between those clusters. The optimum number of clusters, which ensures the largest intra-distances, can be determined heuristically through the following steps:

1. Determine the largest vertical line in the dendrogram that does not intersect any of the other clusters. In our example, it is the vertical line from 9.43 to 14.14 with a length of 4.71.
2. Draw a horizontal line along this line.
3. The optimal number of clusters is equal to the number of intersections this horizontal line has. In our example, there are two intersections, as shown below.

Figure 26: Optimal Clusters



Source: Walid Hussein, 2020.

Hence, the optimum number of clusters for the input dataset is two, where the first cluster contains the students with IDs 1 and 2, while the second cluster contains the students with IDs 3, 4, and 5.

Hierarchical Clustering in Python

The hierarchical clustering algorithm has been implemented in Python as a module in the "sklearn" package. We will run the algorithm on the dataset for 440 customers of a wholesale distributor that contains their annual spending on varied products with the following features:

Table 8: Wholesale Distributor Dataset

Feature	Description
Fresh	Annual spending on fresh products
Milk	Annual spending on milk products

Feature	Description
Grocery	Annual spending on grocery products
Frozen	Annual spending on frozen products
Detergents	Annual spending on detergents and paper products
Delicatessen	Annual spending on delicatessen products
Channel	Customers channel (hotel, restaurant, café, or retail) in nominal representation
Region	Customers region in nominal representation

Source: Walid Hussein, 2020.

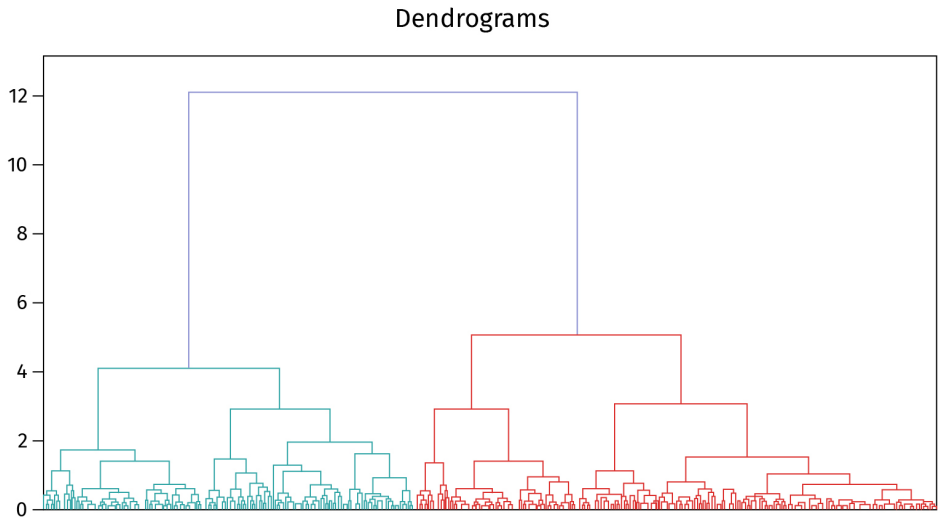
The Python code to cluster the customers with respect to these features is explored below.

Code

```
>>> import pandas as pd
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> data = pd.read_csv('Wholesale customers data.csv')
>>> data.head()
 Channel Region Fresh Milk Grocery Frozen Detergents_Paper Delicassen
0   2   3 12669 9656  7561  214      2674   1338
1   2   3 7057 9810  9568 1762      3293   1776
2   2   3 6353 8808  7684 2405      3516   7844
3   1   3 13265 1196  4221 6404       507   1788
4   2   3 22615 5410  7198 3915      1777   5185
>>>
#Normalize the dataset to get all the features at the same scale
>>> from sklearn.preprocessing import normalize
>>> data_scaled = normalize(data)
>>> data_scaled = pd.DataFrame(data_scaled, columns=data.columns)
>>> data_scaled.head()
 Channel Region  Fresh ... Frozen Detergents_Paper Delicassen
0 0.000112 0.000168 0.708333 ... 0.011965    0.149505 0.074809
1 0.000125 0.000188 0.442198 ... 0.110409    0.206342 0.111286
2 0.000125 0.000187 0.396552 ... 0.150119    0.219467 0.489619
3 0.000065 0.000194 0.856837 ... 0.413659    0.032749 0.115494
4 0.000079 0.000119 0.895416 ... 0.155010    0.070358 0.205294

[5 rows x 8 columns]
>>>
# Draw the dendrogram to find the optimum number of clusters
>>> import scipy.cluster.hierarchy as shc
>>> plt.figure(figsize=(10, 7))
<Figure size 1000x700 with 0 Axes>
```

```
>>> plt.title("Dendrograms")
Text(0.5, 1.0, 'Dendrograms')
>>> dend = shc.dendrogram(shc.linkage(data_scaled, method='ward'))
>>> plt.show()
```

Figure 27: Wholesale Distributor Python Code I



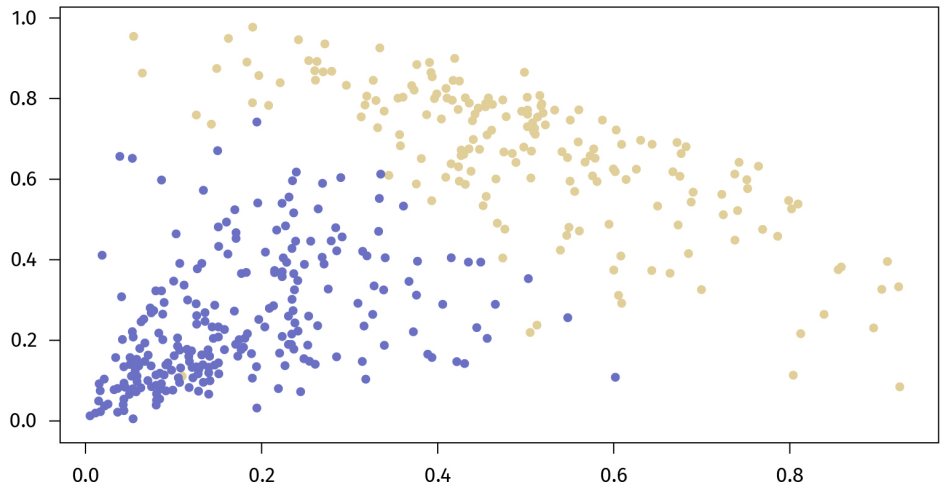
Source: Walid Hussein, 2020.

Code

```
# hence we can decide an optimum number of clusters equals 2
# apply hierarchical clustering foe two clusters only

>>> from sklearn.cluster import AgglomerativeClustering
>>> cluster = AgglomerativeClustering(n_clusters=2,
affinity='euclidean', linkage='ward')
>>> cluster.fit_predict(data_scaled)
array([1, 1, 1, 0, 0, 1, 0, 1, 1, 1, 1, 0, 0, 0, 0, 0, 1, 1, 0, 0,
   0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0,
   1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1,
   0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 1,
   1, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 1, 1,
   1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0,
   0, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1,
   0, 1, 0, 1, 1, 0, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1,
   1, 1, 0, 0, 0, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 1, 0, 0,
   0, 1, 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 1, 0,
   1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0,
   0, 0, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
```

```
   0, 1, 1, 1, 1, 0, 1, 0, 0, 1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0,
   0, 0, 0, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1, 1, 1, 1,
   1, 0, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1,
   0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 1, 1,
   0, 1, 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0,
   1, 0, 0, 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 1, 0,
   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1,
   1, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 1,
   0, 1],
  dtype=int64)
# where "0" implies a customer in the first cluster, and "1" implies
a customer in the second cluster.

# to visualize the two clusters
>>> plt.figure(figsize=(10, 7))
<Figure size 1000x700 with 0 Axes>
>>> plt.scatter(data_scaled['Milk'], data_scaled['Grocery'],
c=cluster.labels_)
<matplotlib.collections.PathCollection object at 0x000000F866217860>
>>> plt.show()
```

Figure 28: Wholesale Distributor Python Code II



Source: Walid Hussein, 2020.

Limitations of Hierarchical Clustering

Hierarchical clustering is easy to implement and results in a hierarchy that is more informative than the unstructured set of flat clusters returned by most of the other clustering approaches. Therefore, it is easier to decide on the number of clusters by looking at the

formed tree (i.e., dendrogram). The most important value of the hierarchical clustering is the lack of assumptions concerning a particular number of clusters in the beginning of the algorithm.

Once a decision is made to combine two clusters, it cannot be undone. Hence, it is not possible to undo the previous step. Hierarchical clustering is not suitable for large datasets and is very sensitive to outliers within the data. As an alternative for large datasets, k-means is typically faster to compute and produces tighter clusters than hierarchical clustering (Pragathi et al., 2018).

2.4 Density-Based Clustering

The centroid-based clustering algorithms do not address the fake outliers, so these data records are usually assigned to clusters even if they do not belong in any. On the other hand, density-based clustering approaches work to identify clusters by grouping "dense" data records together, which permits the representation of arbitrarily shaped clusters, as well as the learning of outliers within the data.

The estimation of "dense" clusters is based on the ε-neighborhood concept. The ε-neighborhood of a data record p is the set of points contained in a shape of extension 2ε, centered at p. Assuming the Euclidean distance measure, this surface is a circle for 2D space, a sphere for 3D space, and N-sphere for general multi-dimensional space. In general, the ε-neighborhood of a point p in the dataset d is given by:

$$\mathrm{N(p)} = \{\mathrm{q} \in \mathrm{d} \,|\, \mathrm{dist(p,q)} \leq \epsilon\}$$

The number of data points in $\mathrm{N(P)}$ indicates how dense the ε-neighborhood at p is. For the subsequent clustering, we define a minimum density value of MinPts for a neighborhood to be considered part of a cluster. With these definitions, there are three possible descriptions for the data record p, which are core point, border point, or noise (i.e., outlier) point.

A data point is a core point if its ε-neighborhood contains more than the specified MinPts. It is a border point if the ε-neighborhood has fewer than the specified (MinPts), but the point is in the ε-neighborhood of a core point. It is a noise point if it is neither a core point, nor a border point.

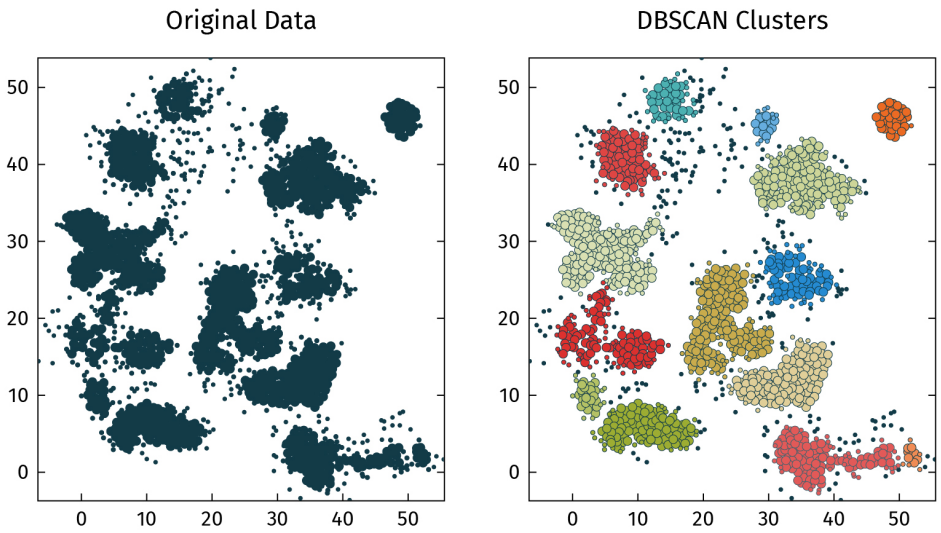
DBSCAN Algorithm

Ester et al. (1996) proposed a technique to perform the general density-based spatial clustering for applications with noise (DBSCAN). This technique is efficient for large datasets of thousands of data records yet requires some domain knowledge of the underlying datasets.

The steps of DBSCAN algorithm are explained as follows:

1. Assume (MinPts) and the surface radius (ε).
2. Start at an arbitrary data record (p).
3. If p contains (MinPts) within its εneighborhood, cluster formation starts. Otherwise, p is labeled as noise. N.B. This may be modified later if p found within the ε-neighborhood of a core point data record.
4. Density-connected cluster step: if p is a core point, then the data records within its ε-neighborhood are part of the formed cluster. All of these data records are added, along with their own ε-neighborhood, if they are also core points.
5. Start at new data record (not previously processed), and repeat steps 3 to 4 until there are no unhandled data records, as schematically described in the following figure.

Figure 29: DBSCAN Example



Ernst (2017)

Source: Ernst, 2017.

DBSCAN in Python

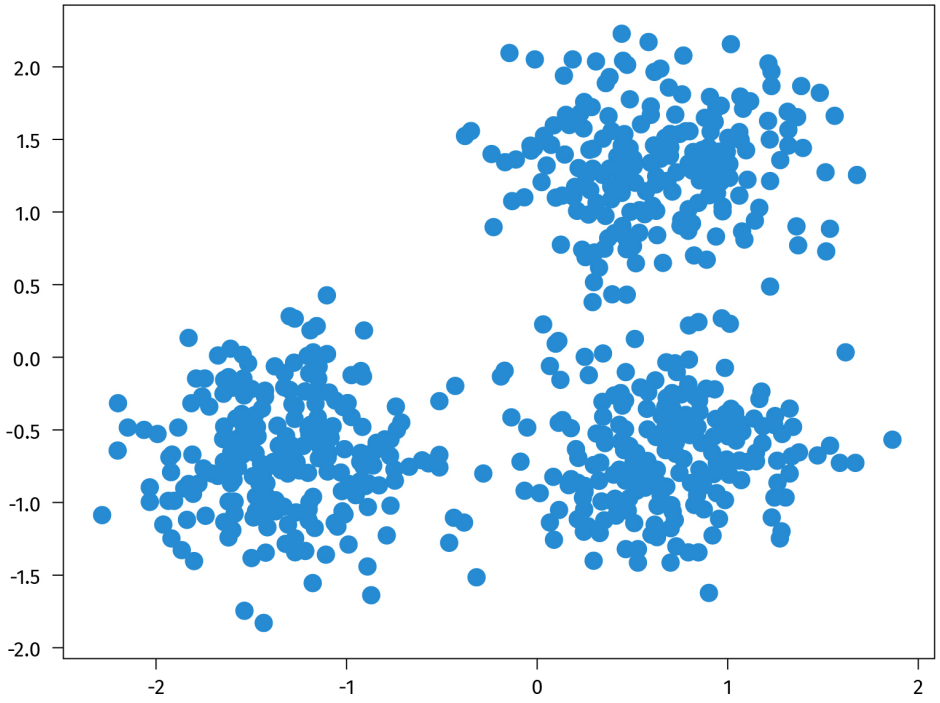
To implement the DBSCAN algorithm in Python, we will use the sklearn package and its supported module DBSCAN, as elaborated below for the clustering of a dataset of 750 randomly generated samples.

Code

```
>>> import numpy as np
>>> from sklearn.cluster import DBSCAN
>>> from sklearn import metrics
>>> from sklearn.datasets.samples_generator import make_blobs
>>> from sklearn.preprocessing import StandardScaler
>>> from pylab import *
```

```
>>> import matplotlib.pyplot as plt

>>> # Generate sample data
>>> centers = [[1, 1], [-1, -1], [1, -1]]
>>> X, labels_true = make_blobs(n_samples=750, centers=centers,
cluster_std=0.4, random_state=0)
>>> # Scale and standardize the dataset
>>> X = StandardScaler().fit_transform(X)
>>> xx, yy = zip(*X)
>>> scatter(xx,yy)
<matplotlib.collections.PathCollection object at 0x000000CB1DB24D30>
>>> show()
```

Figure 30: DBSCAN in Python 1a



Source: Walid Hussein, 2020.

Code

```
>>> # Set up DBSCAN parameters
>>> db = DBSCAN(eps=0.3, min_samples=10).fit(X)
>>> core_samples = db.core_sample_indices_
>>> core_samples

Squeezed text (53 lines).
```

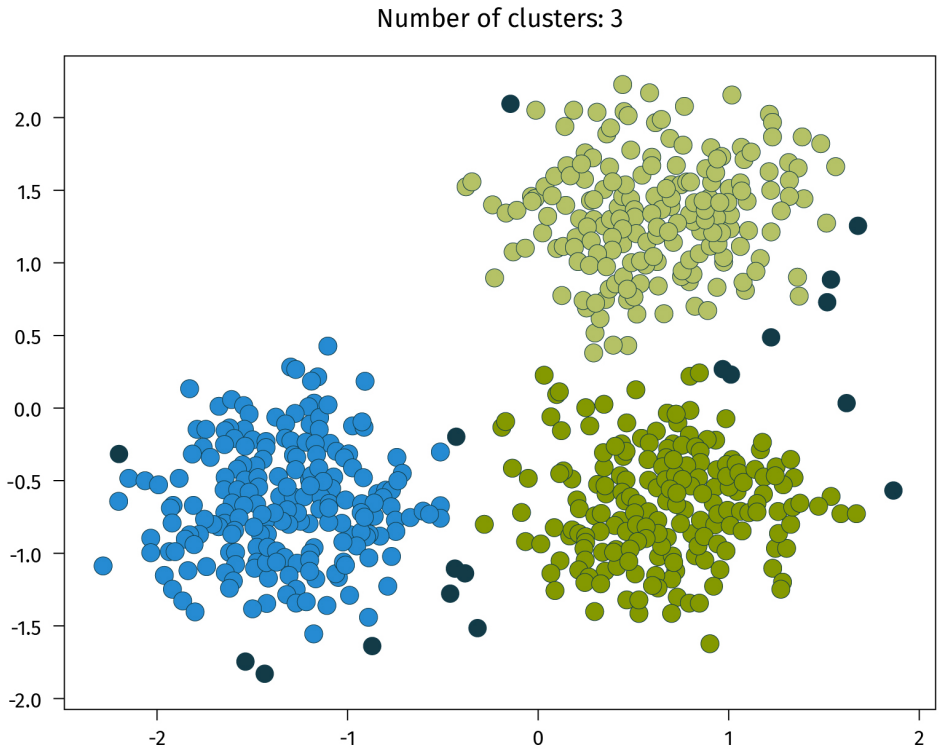
```
>>> core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
>>> core_samples_mask[db.core_sample_indices_] = True

>>># the number of clusters
>>> n_clusters_ = len(set(labels)) - (1 if -1 in labels else 0)
>>> n_clusters_
3
>>> labels = db.labels_
>>> labels
array([ 0, 1, 0, 2, 0, 1, 1, 2, 0, 0, 1, 1, 1, 2, 1, 0, -1, 1, 1,
2, 2, 2, 2, 2, 1, 1, 2, 0, 0, 2, 0, 1, 1, 0, 1, 0, 2, 0, 0, 2, 2,
1, 1, 1, 1, 1, 0, 2, 0, 1, 2, 2, 1, 1, 2, 2, 1, 0, 2, 1, 2, 2, 2,
2, 2, 0, 2, 2, 0, 0, 0, 2, 0, 0, 2, 1, -1, 1, 0, 2, 1, 1, 0, 0, 0,
0, 1, 2, 1, 2, 2, 0, 1, 0, 1, -1, 1, 1, 0, 0, 2, 1, 2, 0, 2, 2, 2,
2, -1, 0, -1, 1, 1, 1, 1, 0, 0, 1, 0, 1, 2, 1, 0, 0, 1, 2, 1, 0, 0,
2, 0, 2, 2, 2, 0, -1, 2, 2, 0, 1, 0, 2, 0, 0, 2, 2, -1, 2, 1, -1, 2,
1, 1, 2, 2, 2, 0, 1, 0, 1, 0, 1, 0, 2, 2, -1, 1, 2, 2, 1, 0, 1, 2,
2, 2, 1, 1, 2, 2, 0, 1, 2, 0, 0, 2, 0, 0, 1, 0, 1, 0, 1, 1, 2, 2, 0,
0, 1, 1, 2, 1, 2, 2, 2, 2, 0, 2, 0, 2, 2, 0, 2, 2, 2, 0, 0, 1, 1, 1,
 2, 2, 2, 2, 1, 2, 2, 0, 0, 2, 0, 0, 0, 1, 0, 1, 1, 1, 2, 1, 1, 0, 1,
2, 2, 1, 2, 2, 1, 0, 0, 1, 1, 1, 0, 1, 0, 2, 0, 2, 2, 2, 2, 2, 1, 1,
0, 0, 1, 1, 0, 0, 2, 1, -1, 2, 1, 1, 2, 1, 2, 0, 2, 2, 0, 1, 2, 2,
0, 2, 2, 0, 0, 2, 0, 2, 0, 2, 1, 0, 0, 0, 1, 2, 1, 2, 2, 0, 2, 2, 0,
0, 2, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 0, 0, 1, 1, 1, 0, 2, 0, 1, 2, 2,
0, 0, 2, 0, 2, 1, 0, 2, 0, 2, 0, 2, 2, 0, 1, 0, 1, 0, 2, 2, 1, 1, 1,
2, 0, 2, 0, 2, 1, 2, 2, 0, 1, 0, 1, 0, 0, 0, 0, 2, 0, 2, 0, 1, 0, 1,
2, 1, 1, 1, 0, 1, 1, 0, 2, 1, 0, 2, 2, 1, 1, 2, 2, 2, 1, 2, 1, 2, 0,
2, 1, 2, 1, 0, 1, 0, 1, 1, 0, 1, 2, -1, 1, 0, 0, 2, 1, 2, 2, 2, 2, 1,
0, 0, 0, 0, 1, 0, 2, 1, 0, 1, 2, 0, 0, 1, 0, 1, 1, 0, -1, 0, 2, 2, 2,
1, 1, 2, 0, 1, 0, 0, 1, 0, 1, 1, 2, 2, -1, 0, 1, 2, 2, 1, 1, 1, 1, 0,
0, 0, 2, 2, 1, 2, 1, 0, 0, 1, 2, 1, 0, 0, 2, 0, 1, 0, 2, 1, 0, 2, 2,
1, 0, 0, 0, 2, 1, 1, 0, 2, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 0, 0, 2, 0,
1, 1, 2, 1, 1, 0, 1, 0, 2, 1, 0, 0, 1, 0, 1, 1, 2, 2, 1, 2, 2, 1, 2,
1, 1, 1, 1, 2, 0, 0, 0, 1, 2, 2, 0, 2, 0, 2, 1, 0, 1, 1, 0, 0, 1, 2,
1, 2, 2, 0, 2, 1, 1, 1, 2, 0, 0, 2, 0, 2, 2, 0, 2, 0, 1, 1, 1, 1, 0,
0, 0, 2, 1, 1, 1, 1, 2, 2, 2, 0, 2, 1, 1, 0, 0, 1, 0, 2, 1, 2, 1, 0,
2, 2, 0, 0, 1, 0, 0, 2, 0, 0, 0, 2, 0, 2, 0, 0, 1, 1, 0, 0, 1, 2, 2,
0, 0, 0, 0, 2, -1, 1, 1, 2, 1, 0, 0, 2, 2, 0, 1, 2, 0, 1, 2, 2, 1, 0,
0, -1, -1, 2, 0, 0, 0, 2, -1, 2, 0, 1, 1, 1, 1, 1, 0, 0, 2, 1, 2, 0,
1, 1, 1, 0, 2, 1, 1, -1, 2, 1, 2, 0, 2, 2, 1, 0, 0, 0, 1, 1, 2, 0, 0,
2, 2, 1, 2, 2, 2, 0, 2, 1, 2, 1, 1, 1, 2, 0, 2, 0, 2, 2, 0, 0, 2, 1,
2, 0, 2, 0, 0, 0, 1, 0, 2, 1, 2, 0, 1, 0, 0, 2, 0, 2, 1, 1, 2, 1, 0,
1, 2, 1, 2], dtype=int64)

>>># if the label equals “-1”, this means the data record is an outlier.
>>># find the outliers
>>> outliers = X[labels == -1]
>>> outliers
```

```
array([[-1.4302079 , -1.82380068],
   [-0.13447918, 2.11039748],
   [ 1.22425496, 0.50161091],
   [ 1.53624088, 0.89460489],
   [ 1.68745105, 1.27830756],
   [-1.53199232, -1.74146844],
   [-0.42227599, -0.19034456],
   [ 1.52307352, 0.74115752],
   [-0.46307453, -1.26531795],
   [ 1.62198217, 0.05020132],
   [ 1.01170962, 0.24102378],
   [-0.43185167, -1.10093485],
   [-0.3164503 , -1.51218921],
   [-2.19229513, -0.30228576],
   [ 0.9807011 , 0.27373202],
   [-0.86338803, -1.63431796],
   [ 1.87043803, -0.56476419],
   [-0.37447929, -1.13624183]])

>>> # Get the contents of each cluster
>>> cluster1 = X[labels == 0]
>>> cluster1

Squeezed text (243 lines).
>>> cluster2 = X[labels == 1]
>>> cluster2

Squeezed text (244 lines).
>>> cluster3 = X[labels == 2]
>>> cluster3

Squeezed text (245 lines).
>>> # Plot the results with a specific color for each cluster,
and a black color for the noise points
>>> unique_labels = set(labels)
>>> unique_labels
{0, 1, 2, -1}
>>> colors = ['y', 'b', 'g', 'r']
>>> for k, col in zip(unique_labels, colors):
 if k == -1:
   col = 'k'

 class_member_mask = (labels == k)

 xy = X[class_member_mask & core_samples_mask]
 plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
                  markeredgecolor='k',
                  markersize=6)
```

```
 xy = X[class_member_mask & ~core_samples_mask]
 plt.plot(xy[:, 0], xy[:, 1], 'o', markerfacecolor=col,
                  markeredgecolor='k',
                  markersize=6)
```

```
[<matplotlib.lines.Line2D object at 0x000000CB17FFA080>]
[<matplotlib.lines.Line2D object at 0x000000CB17FFA1D0>]
[<matplotlib.lines.Line2D object at 0x000000CB17FFA5C0>]
[<matplotlib.lines.Line2D object at 0x000000CB17FFA908>]
[<matplotlib.lines.Line2D object at 0x000000CB17FFAC50>]
[<matplotlib.lines.Line2D object at 0x000000CB17FFAF98>]
[<matplotlib.lines.Line2D object at 0x000000CB17FFA4A8>]
[<matplotlib.lines.Line2D object at 0x000000CB18005668>]
>>> plt.title('number of clusters: %d' %n_clusters_)
Text(0.5, 1.0, 'number of clusters: 3')
>>> plt.show()
```

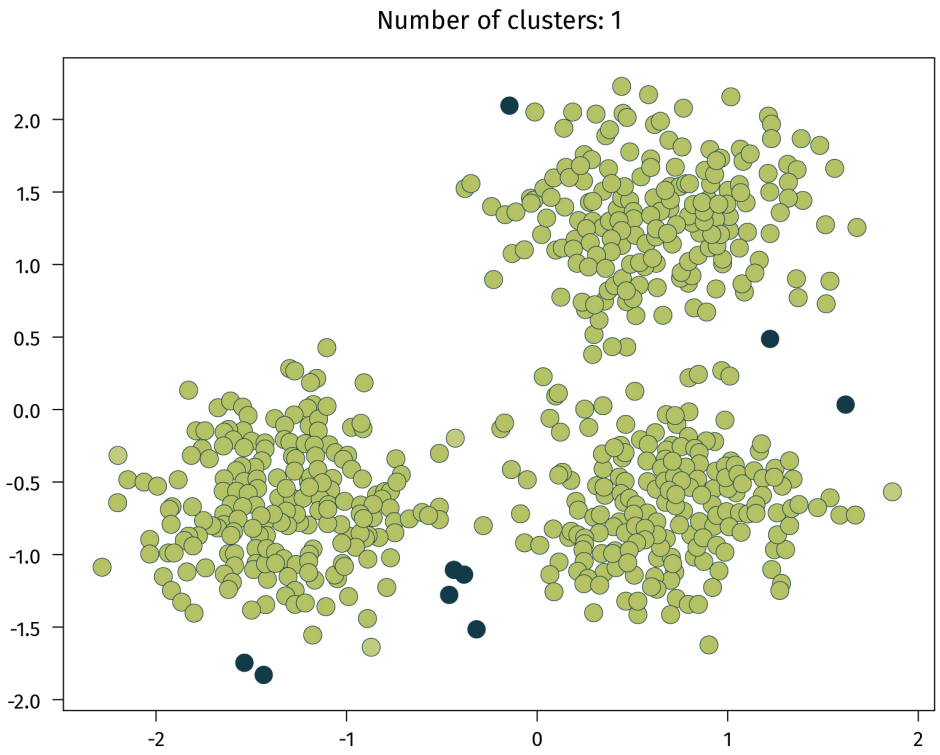
Figure 31: DBSCAN in Python 1b



Source: Walid Hussein, 2020.

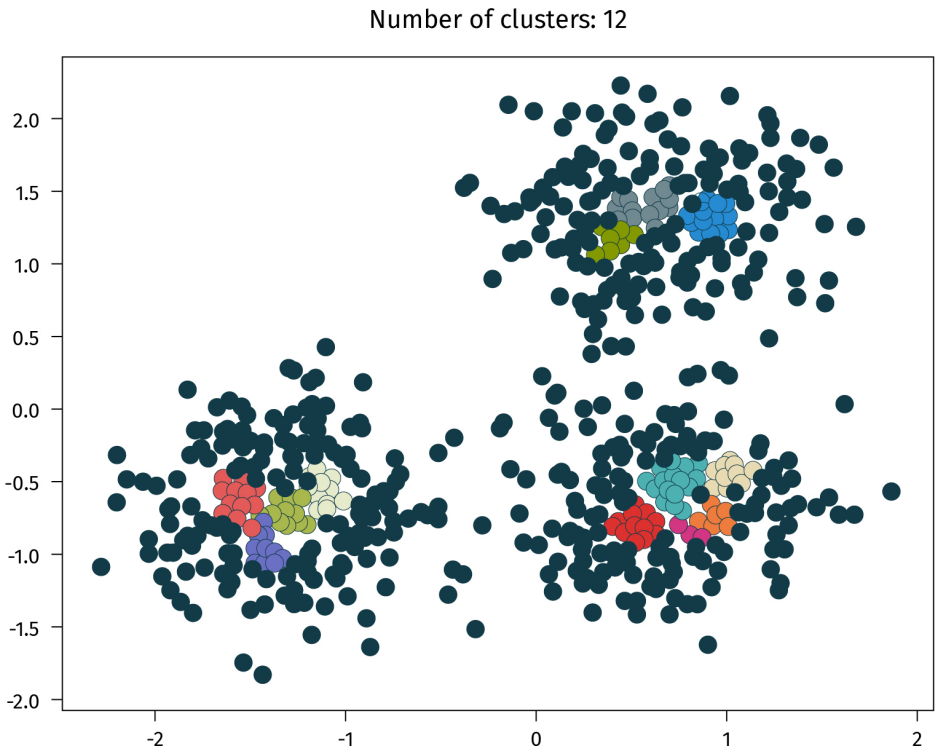
By changing the “eps” and the “MinPts” values, we arrive at different cluster configurations, as shown in the following two figures for two different DBSCAN settings.

Figure 32: DBSCAN Clustering Example (eps = 0.3, min_samples = 5)



Source: Walid Hussein, 2020.

Figure 33: DBSCAN Clustering Example (eps = 0.1, min_samples = 10)



Source: Walid Hussein, 2020.

Limitations of DBSCAN

An important advantage of DBSCAN is that it does not require us to estimate the number of clusters in the beginning of the algorithm. DBSCAN infers the number of clusters according to the underlying dataset, and it discovers arbitrarily shaped clusters. This technique efficiently separates clusters of high density versus clusters of low density within a given dataset, while pragmatically handling the contained outliers.

However, if the dataset has data records that form clusters of varying density, then DBSCAN fails to perform as effectively, since the ε and MinPts parameters cannot be chosen separately for each of the clusters. Furthermore, if the dataset and its involved features are not well understood by a domain expert, then setting up ε and MinPts could be tricky and may need comparisons for several iterations with different values of ϵ and MinPts.

SUMMARY

Clustering analysis is an unsupervised learning tool that helps the data analyst to infer data-driven patterns that may warrant further investigation, without the need for labeled outcomes. The data records that are more similar to each other are grouped together in one cluster.

The major clustering methods are centroid-based clustering, where the k-means approach is the commonly used technique; Gaussian mixture models clustering, where the expectation maximization algorithm is implemented to find the best parameters for probability distributions that maximize the likelihood; hierarchical clustering, applied either by agglomerative (bottom-up) or by divisive (top-down) clustering; and density based clustering, commonly applied using the DBSCAN technique, which groups the data records according to the density of placement within the dataset.

UNIT 3

REGRESSION

STUDY GOALS

On completion of this unit, you will have learned …

- what is meant by regression.
- the different types of regression analysis.
- the simple linear regression model.
- the different metrics to evaluate regression models.
- the logistic and quantile regression models.
- the regularization approaches in regression analysis.

3. REGRESSION

Introduction

The performance of a business organization can be typically measured with a number of performance indicators that reflect relevant aspects of the work. Examples of these indicators include employee performance, product specifications, and customer satisfaction. One of the most important outcome-oriented measures that is easy to measure objectively is achieved profit. In practice, there is a relationship between process-oriented business indicators and achieved profit.

Is there a mathematical tool to define such a relationship? What is each of the business's parameters share in this relationship? Can we develop a tool that can be used to accurately predict the achieved profit if one of the involved parameters of the business is adjusted?

Applying regression analysis to the business data can answer all of these questions.

3.1 Linear & Nonlinear Regression

Regression
This statistical method attempts to determine the strength and character of the relationship between dependent variable(s) and independent variable(s).

Regression is a supervised learning approach, where the given dataset is labeled (i.e., has one or more target variables). The aim of the regression analysis is to find an expectation for the relationship between the target variable (i.e., dependent variable) and the existent dependent variables within the dataset.

The regression analysis of a dataset with a single independent variable is called a simple regression. If there are multiple independent variables, the regression analysis is called a multiple regression.

The relationship between the dependent variable and the independent variable(s), which is retrieved through a regression analysis, can be either linear or nonlinear. In linear regression, the relationship is a straight-line equation, as given below.

$$\mathrm{y} = \mathrm{a}_1 \cdot \mathrm{x}_1 + \mathrm{a}_2 \cdot \mathrm{x}_2 + \ldots + \mathrm{a}_\mathrm{n} \cdot \mathrm{x}_\mathrm{n} + \mathrm{b}$$

where $\{\mathrm{x}_1,\mathrm{x}_2,\ldots,\mathrm{x}_\mathrm{n}\}$ are the independent variables of the dataset, and $\{\mathrm{a}_1,\mathrm{a}_2,\ldots,\mathrm{a}_\mathrm{n}\}$ are the coefficients of these variables, implying the weight that each independent variable shares in the resultant target variable *y*. The $\{\mathrm{b}\}$ value is the constant or bias term.

On the other hand, in **nonlinear regression**, the relationship is a nonlinear equation (e.g., polynomial, exponential), as described below.

$$\mathrm{y} = \mathrm{a}_1 \cdot (\mathrm{x}_1)^2 + \mathrm{a}_2 \cdot \mathrm{e}^{\mathrm{x}_2} + \ldots + \mathrm{a}_\mathrm{n} \cdot (\mathrm{x}_\mathrm{n})^3 + \mathrm{b}$$

Hence, in both cases of linear and nonlinear regression analysis, the objective is to find the optimum values for the coefficients $\{a_1,a_2,...,a_n,b\}$. In general, the regression analysis follows four steps: model selection, model fitting, model prediction, and model evaluation. The model selection step involves the choice of the model (i.e., relationship) shape. This means to decide whether the regression is linear or nonlinear, and simple or multiple.

Nonlinear regression
In this form of regression analysis, observational data are modeled by a function which is a nonlinear combination of the model parameters.

Afterwards, the model-fitting step is employed to the given dataset and finds the unknown coefficients of the chosen model. Next, the developed model is applied in the model prediction step to estimate the target variable for some other hitherto unseen dataset elements. Finally, the inferred model is evaluated in the model evaluation step by checking how close the model's predictions are to the desired target values.

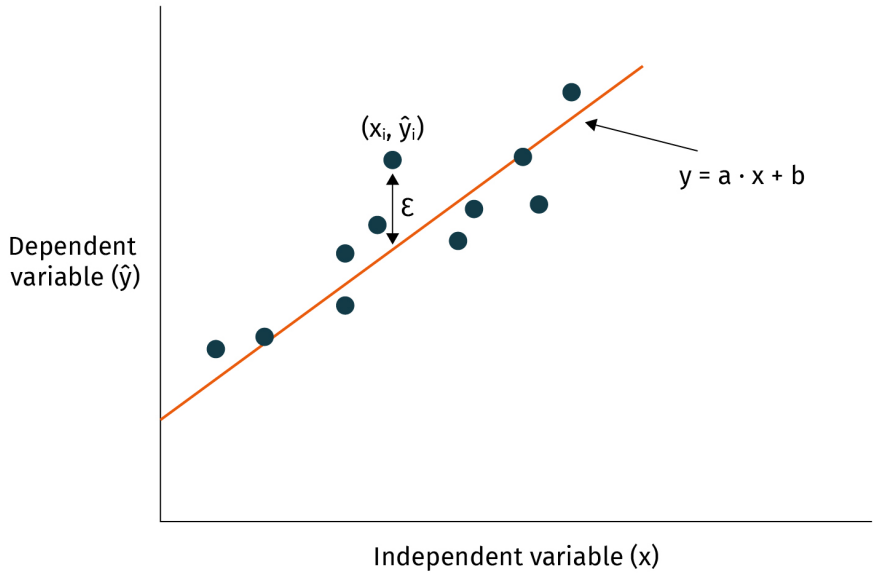
Simple Linear Regression

For a dataset with one independent variable (x) and one dependent variable $(\hat{y})$, the simple linear regression analysis will result in a straight-line relationship between x and $\hat{y}$, as shown below.

$$y = a \cdot x + b$$
$$\hat{y} = y + \varepsilon$$

The aim is to find the values of a and b, such that the error term ε is minimized, and y will be close to the desired target variable $\hat{y}$, as described in the following figure.

Figure 34: Simple Linear Regression



Source: Walid Hussein, 2020.

Residue
A small amount that remains after the main part has been taken away is called residue.

For each data point, the difference between the predicted value y and the actual observation $\hat{y}$ is the **residue** ε. The evaluation metric for the simple regression model is the sum of the squared residues (i.e., sum of the squared errors (E)), which we aim to minimize.

Hence,

$$E = \sum (\hat{y} - y)^2 = \sum (\hat{y} - (a \cdot x + b))^2$$

From differential calculus, the minimum value of E can be derived by setting

$$\left(\frac{\partial E}{\partial a} = 0\right) \text{and} \left(\frac{\partial E}{\partial b} = 0\right)$$

which can be solved to get the optimum settings for both a and b, as follows:

$$\begin{aligned}
&\frac{\partial E}{\partial b} = 0 \\
&2\sum (\hat{y} - (b + a \cdot x)) \cdot (-1) = 0 \\
&\sum \hat{y} - nb - a\sum x = 0 \\
&\therefore b = \frac{1}{n}\sum \hat{y} - \frac{a}{n}\sum x
\end{aligned}$$

and,

$$\begin{aligned}
&\frac{\partial E}{\partial a} = 0 \\
&2\sum (\hat{y} - (b + a \cdot x)) \cdot (-x) = 0 \\
&-\sum \hat{y} \cdot x + b\sum x + a\sum x^2 = 0 \\
&-\sum \hat{y}x + \left(\frac{1}{n}\sum \hat{y} - \frac{a}{n}\sum x\right)\sum x + a\sum x^2 = 0 \\
&a\left(\frac{-1}{n}\left(\sum x\right)^2 + \sum x^2\right) - \sum \hat{y}x + \frac{1}{n}\sum \hat{y}\sum x \\
&\therefore a = \frac{n\sum \hat{y}x - \sum \hat{y}\sum x}{n\sum x^2 - \left(\sum x\right)^2}
\end{aligned}$$

For datasets with more than one independent variable, the same procedure needs to be followed for each of the independent variables to find all the involved parameters.

R^2 Metric for Correlation Analysis

An important metric, the **coefficient of determination** $(R$, or its squared value, $R^2)$ is routinely employed to gauge the quality of fit of the derived regression relationship. R^2 describes the proportion of variance in the dependent variable that is explained by the independent variables. If the correlation between independent and dependent variables

is strong, a robust implementation of the developed regression model that is predictive for new data inputs can be achieved. R^2 ranges from zero to one, where a value close to one indicates a good quality of fit of the regression model.

Coefficient of determination
In statistics, we use the coefficient of determination to measure the strength of the relationship between two variables.

Test of Linearity

For the developed model to satisfy the linearity assumption, it should pass the test of linearity. The resultant ε values should be small with respect to a specified threshold, and there should be no significant pattern for the distribution of these ε values. Otherwise, the test of linearity fails and a nonlinear regression model has to be developed for better representation of the dataset.

When a linear regression fails to fit a linear model with strong correlation from the independent variable to the dependent variable, a nonlinear or polynomial regression is advised to correlate the data variables in a curvilinear model.

3.2 Logistic Regression

Simple (and multiple) linear regression assumes a continuous dependent variable with additive noise that follows a normal distribution with constant variance. Indeed, these preconditions are not always met. In particular, the dependent variable may follow a binomial distribution and take on categorical values that might be represented in text form (e.g., yes/no) or as discrete values (e.g., $\{0, 1\}$).

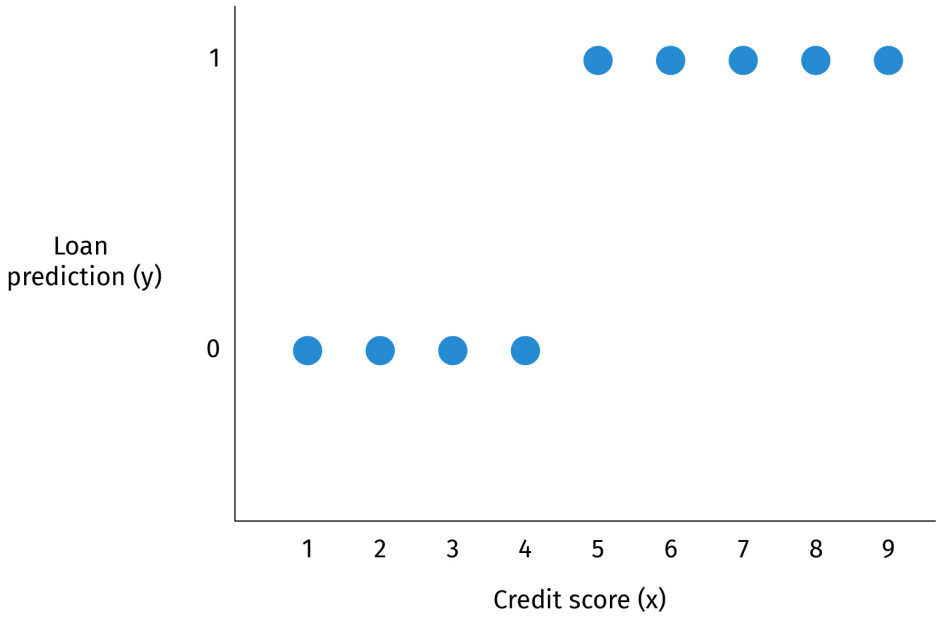
For these cases, a different regression model has to be developed: the **logistic regression** model.

Logistic regression
This statistical model uses a logistic function to model a binary dependent variable.

Logistic regression is considered an extension of the linear regression analysis, and its corresponding model can be used to classify input data records into a set of given categories or discrete values that form the dependent variable.

For example, the dependent variable (y) may represent a loan prediction (approved/rejected) based on a credit score (i.e., the independent variable) x. We may assume two values for y: “0” to denote a rejected loan status and “1” to denote an approved loan status, as shown in the following figure.

Figure 35: Logistic Regression Example



Source: Walid Hussein, 2020.

In this example, a linear regression model could possibly present unbounded values for y, i.e., $y =]-\infty, +\infty[$. In particular, y will cover a continuous range of values including "0" and "1",but it will not exactly produce { "0", "1" } output. Therefore, a logit function "S" shape is introduced to the linear regression equation, and the underlying linear model will map the predictions with the logistic function to produce binary values {"0", "1" }.

The rationale for logistic regression is based on odds ratio, where the odds of a specific event occurring are defined as the probability of an event occurring divided by the probability of that event not occurring, as given by the following equation.

$$\text{odds ratio of "1"} = \frac{\text{probability}(y = 1)}{1 - \text{probability}(y = 1)}$$

The maximum likelihood method generates the logit function that predicts the natural logarithm of the odds ratio (Bhattacharyya, 2018).

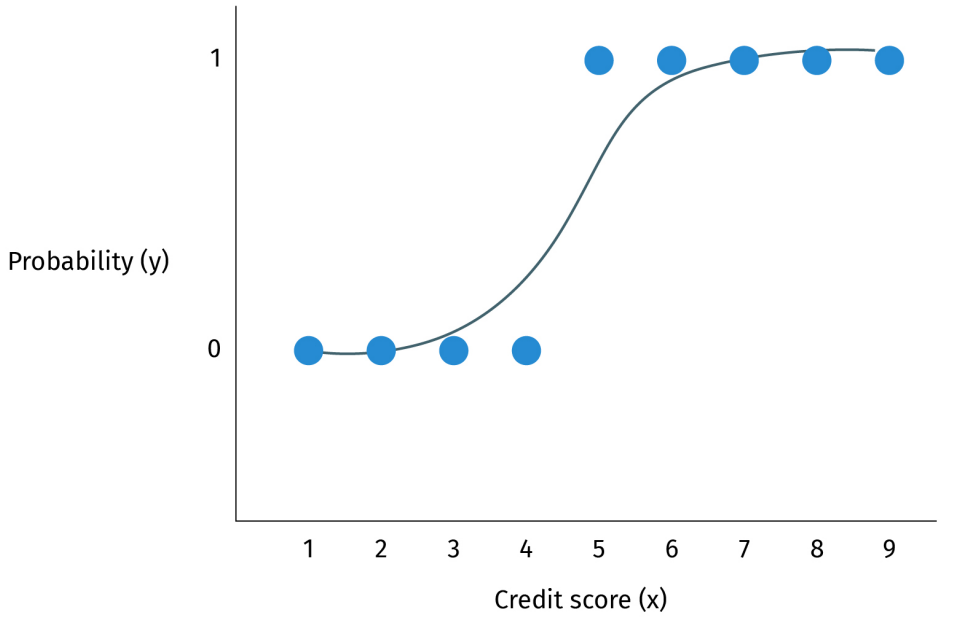
$$\log\left(\frac{p}{(1-p)}\right) = a \cdot x + b$$

The predicted odds ratio and the predicted probability of success are found next. Hence, the logistic regression equation is

$$p(y) = \frac{1}{1 + e^{-(a \cdot x + b)}}$$

The plot of the logistic equation on the given dataset is shown in the following figure.

Figure 36: Logistic Regression Function



Source: Walid Hussein, 2020.

3.3 Quantile Regression

In classical linear regression models, the model coefficients are constant across all data records. These coefficients are obtained by minimizing the sum of squares of the residuals. This implies that the "average" changes in the dependent variable across the interval spanned by the values of the independent variable(s) control the resulting model.

However, such a rough assumption is not always true for all datasets. There are many cases, such as skewed data, multimodal data, or data with outliers, where the focus on the average changes in the dependent variable fails to fully capture the structure of the underlying data.

Thus, a good approach is to divide the dependent variable into segments (i.e., quantiles) from its lowest value to its highest value and develop a linear regression model for each quantile, known as **quantile regression**. The regression coefficients that define an independent variable's influence on any of the quantiles with respect to their dependent variable are calculated.

Quantile regression
An extension of linear regression used when the conditions of linear regression are not met is known as a quantile regression.

While the quantile regression model is developed differently, the interpretation is basically the same as that of a linear regression analysis.

Recall the linear regression model in the following equation

$$\begin{aligned} \mathrm{y} &= \mathrm{a} \cdot \mathrm{x} + \mathrm{b} \\ \widehat{\mathrm{y}} &= \mathrm{y} + \varepsilon \end{aligned}$$

Instead of minimizing the sum of the squares of the residuals (E), the objective in quantile regression is to minimize a "weighted" sum of the absolute errors at each quantile of the dependent variable, as given below.

$$\min\left(\tau \sum_{\text{above } \tau}^{\text{segments}} |\widehat{\mathrm{y}} - \mathrm{y}| + (1-\tau) \cdot \sum_{\text{below}\tau}^{\text{segments}} |\widehat{\mathrm{y}} - \mathrm{y}|\right)$$

where τ is the quantile level. The dependent variable can be divided into a set of quantile levels such as {5%, 10%, 50%, 90%, and 95%}. By optimizing this loss function, the regression coefficients (a and b) are estimated, resulting in a linear relationship between y and x at this specific τ quantile.

Koenker and Hallock (2001) provide the example of the relationship between infants' birth weight (dependent variable, y) and a set of independent variables: the infant's gender $(\mathrm{x1})$, mother's marital status $(\mathrm{x2})$, existence of pregnancy care $(\mathrm{x3})$, and mother's smoking status $(\mathrm{x4})$. The dataset covers 50, 000 observations on birth weight, and the corresponding "average" linear regression model is given by

$$\mathrm{y} = 115.\ 9 \cdot \mathrm{x1} + 161.1 \cdot \mathrm{x2} - 227 \cdot \mathrm{x3} - 200.9 \cdot \mathrm{x4} + 3224$$

Although this model can convey whether a particular independent variable is important for the relationship between independent and dependent variables, it cannot indicate whether this particular independent variable varies according to low or high birth weights.

By using the quantile regression with 5, 10, 50, 90, and 95 percent quantile levels, the coefficients of each of the derived regression models are summarized in the following table.

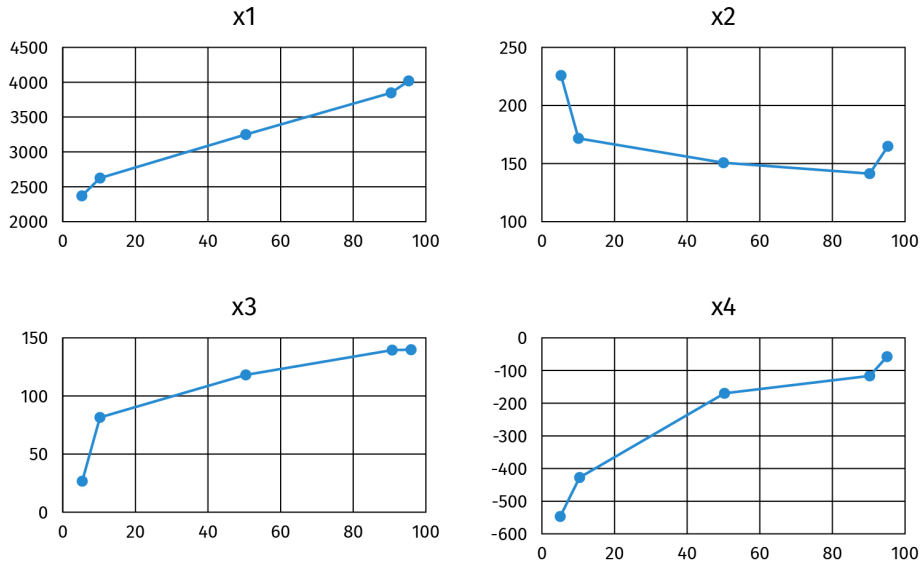
Table 9: Quantile Regression Example

Coefficient	Basic model	$\tau = 5\,\%$	$\tau = 10\,\%$	$\tau = 50\,\%$	$\tau = 90\,\%$	$\tau = 95\,\%$
b	3224	2353	2608	3252	3856	4031
a1	115.9	28	84	121	142	142
a2	161.1	227	171	149	141	165
a3	–227	–536	–418	–164	–111	–57
a4	–200.9	–255	–226	–190	–177	–199

Source: Walid Hussein, 2020.

Based on these coefficients, the following figure presents how the influence of each independent variable is adjusted by changing the investigated quantile of the dependent variable.

Figure 37: Weights of the Independent Variables for Each Quantile



Source: Walid Hussein, 2020.

The graphs above help to acknowledge how independent variables have different weights for each quantile and indicate that a basic linear regression with average weight for each independent variable might not be an optimal solution to assess the correct mapping to the dependent variable.

3.4 Regularization in Regression Analysis

Regression analysis tries to develop the best fit between the independent variables and the dependent variable such that the loss function is minimized and/or the value of R^2 is maximized. However, the developed model can overfit the training dataset and lose the value of generalization when applied to a different testing dataset. Furthermore, while tuning the model for its best fit, the existence of any wrong data points (i.e., outliers) affect the development, leading to an incorrect regression model.

To avoid the issues of overfitting and outliers and to have a more robust model, we penalize the loss function by adding a penalty term to the regression model. Such a penalty is known as regularization. For the case of regression, it comes in two common forms: ridge and lasso regularization.

Ridge Regression

The ridge regression (or L_2 regularization) proceeds by adding a term to the loss function that penalizes the sum of squares of the model coefficients, as given below.

$$\mathrm{E} = \sum(\hat{\mathrm{y}} - \mathrm{y})^2 + \lambda \sum \mathrm{W}^2$$

where λ is a constant that controls the level of the penalty, and W stands for the model coefficients. The higher λ is, the greater the emphasis on the reduction of the coefficients magnitudes, at the expense of tolerating higher residuals.

Lasso Regression

Ridge regression mainly punishes the largest coefficients because their influences are increased when squaring their values, as implied by the penalty term in the above equation. As an alternative, in lasso regression (or L_1 regularization), the objective is to minimize the sum of the absolute values of the coefficients instead of their squares. As a result, both large and small coefficients values are addressed and driven down, as shown below.

$$\mathrm{E} = \sum(\hat{\mathrm{y}} - \mathrm{y})^2 + \lambda \sum |\mathrm{W}|$$

Although the concepts of lasso regression look very similar to that of the ridge regression, the results can differ remarkably. Lasso regression preferentially sets some model coefficients to exactly zero, thus favoring sparse models where possible.

Selecting the optimum value of the regularization parameter λ is a trade-off. By investigating the model performance over a range of λ, we get a graph of which value could act better to avoid any unwelcomed overfitting issues.

3.5 Regression Analysis in Python

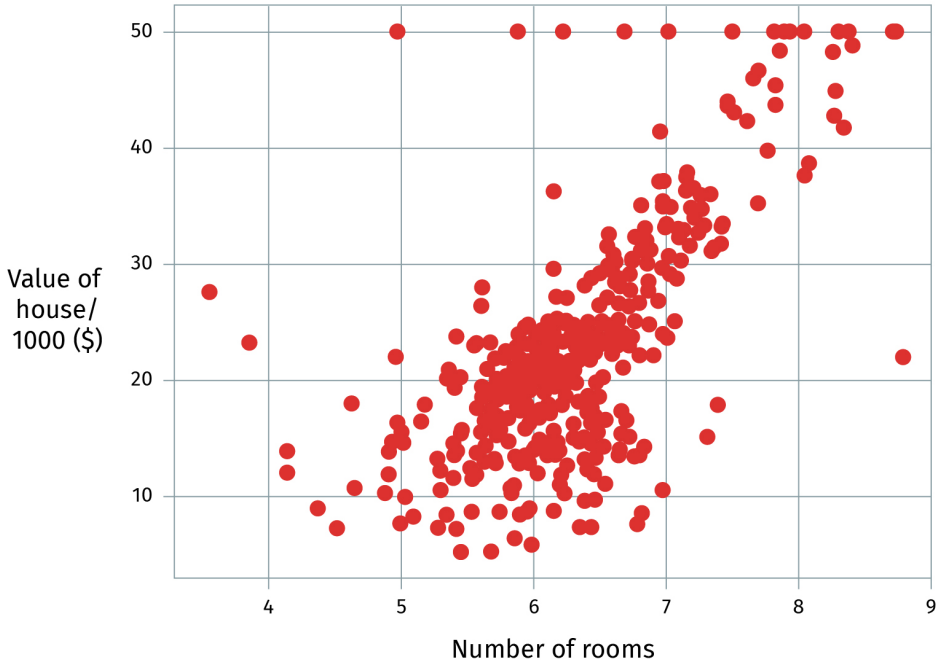
Regression analysis techniques can be easily implemented in Python through the capabilities of the scikit-learn package. In our first implementation, a simple linear regression model uses Boston housing data (a scikit-learn dataset).

Code

```
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> plt.style.use('ggplot')
>>> from sklearn import datasets
```

```
>>> from sklearn import linear_model
>>> import numpy as np

>>> # Load dataset
>>> bostonData = datasets.load_boston() % built-in dataset
>>> yb = bostonData.target.reshape(-1, 1)
>>> Xb = bostonData['data'][:,5].reshape(-1, 1)

>>> # Plot the variables
>>> plt.scatter(Xb,yb)
<matplotlib.collections.PathCollection object at 0x000000165B40DD30>
>>> plt.ylabel('value of house /1000 ($)')
Text(0, 0.5, 'value of house /1000 ($)')
>>> plt.xlabel('number of rooms')
Text(0.5, 0, 'number of rooms')
>>> plt.show()
```

Figure 38: Linear Regression Model: Boston Housing Data I



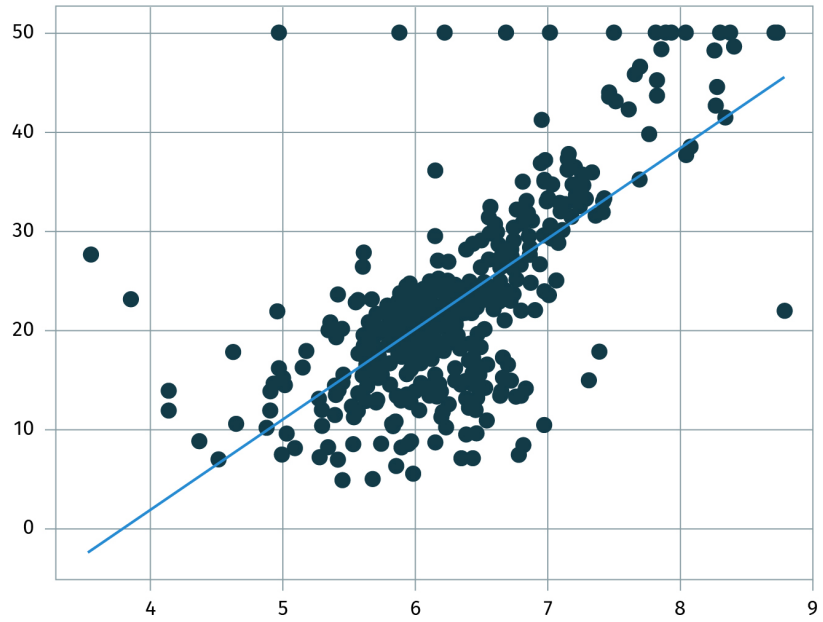
Source: Walid Hussein, 2020.

Code

```
>>> # Create the model
>>> regr = linear_model.LinearRegression()
```

```
>>> # Train the model
>>> regr.fit( Xb, yb)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None,
normalize=False)

>>> # Generate plots
>>> plt.scatter(Xb, yb, color='black')
<matplotlib.collections.PathCollection object at 0x000000166858C6D8>
>>> plt.plot(Xb, regr.predict(Xb), color='blue', linewidth=3)
[<matplotlib.lines.Line2D object at 0x0000001669755128>]
>>> plt.show()
```

Figure 39: Linear Regression Model: Boston Housing Data II



Source: Walid Hussein, 2020.

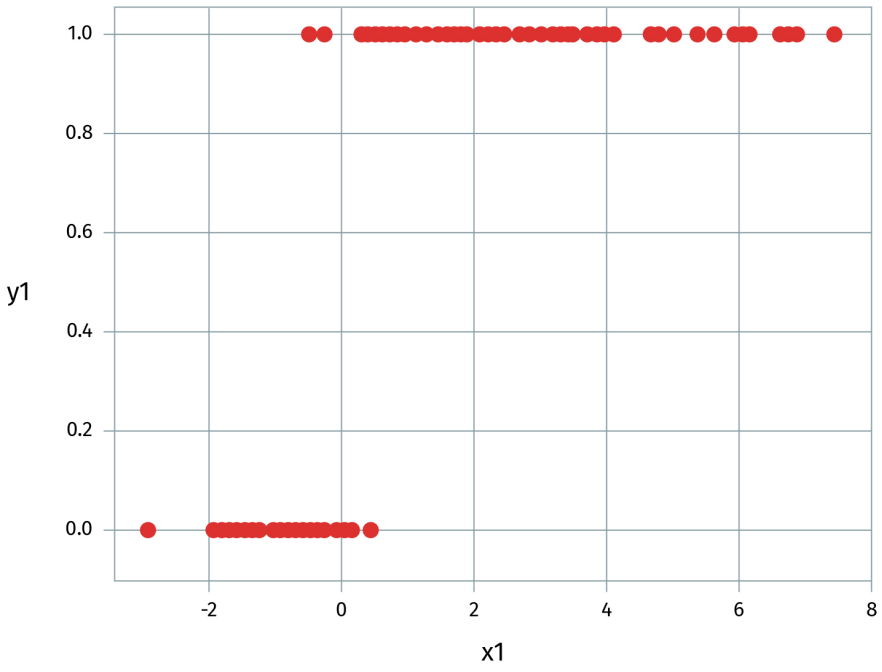
In our second implementation, a logistic regression model will be developed in Python for a random dataset of binary output {0,1}.

Code

```
>>> # dataset
>>> X1 = np.random.normal(size=150)
>>> y1 = (X1 > 0).astype(np.float)
>>> X1[X1 > 0] *= 4
>>> X1 += .3 * np.random.normal(size=150)
>>> X1= X1.reshape(-1, 1)
```

```
>>> # Plot data
>>> plt.scatter(X1,y1)
<matplotlib.collections.PathCollection object at 0x00000016685F3710>
>>> plt.ylabel('y1')
Text(0, 0.5, 'y1')
>>> plt.xlabel('X1')
Text(0.5, 0, 'X1')
>>> plt.show()
```

Figure 40: Logistic Regression Model: Boston Housing Data I



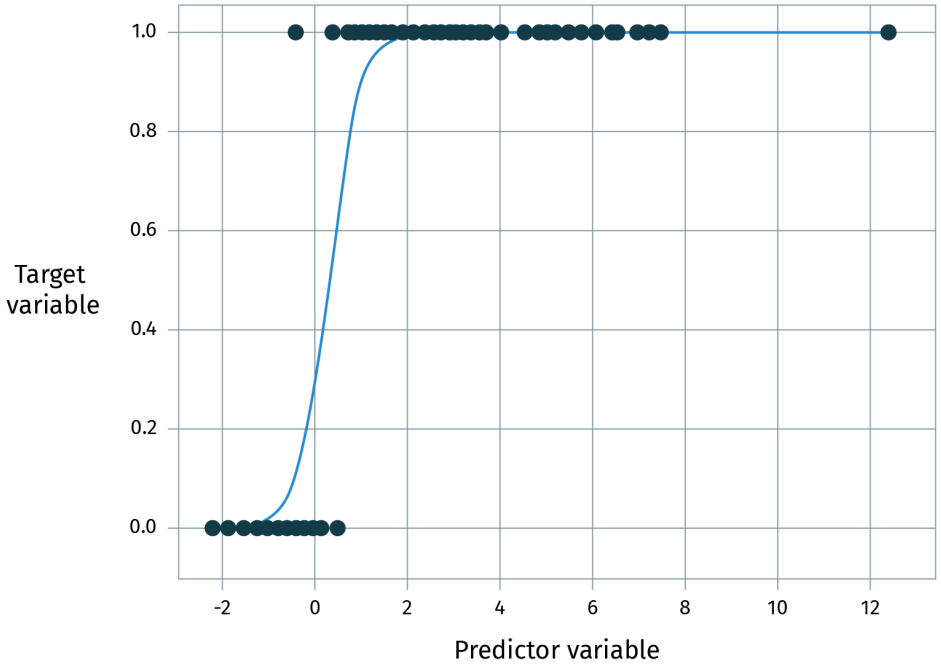
Source: Walid Hussein, 2020.

Code

```
>>> # Run the logistic regression analysis
>>> lm_log = linear_model.LogisticRegression()
>>> lm_log.fit(X1, y1)

>>> # Plot the model
>>> X1_ordered = np.sort(X1, axis=0)
>>> plt.scatter(X1.ravel(), y1, color='black', zorder=20 , alpha = 0.5)
<matplotlib.collections.PathCollection object at 0x0000001665B055F8>
>>> plt.plot(X1_ordered, lm_log.predict_proba(X1_ordered)[:,1],
color='blue' , linewidth = 3)
```

```
>>> plt.ylabel('target variable')
Text(0, 0.5, 'target variable')
>>> plt.xlabel('predictor variable')
Text(0.5, 0, 'predictor variable')
>>> plt.show()
```

Figure 41: Logistic Regression Model: Boston Housing Data II



Source: Walid Hussein, 2020.

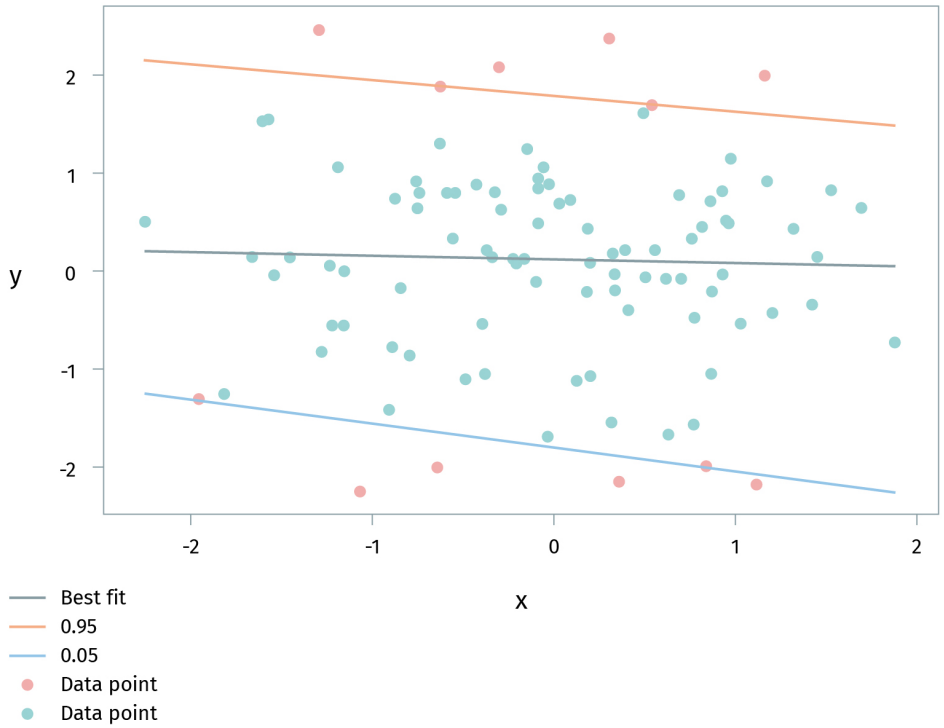
As a third implementation, quantile regression, will be performed on a random dataset, as shown below.

Code

```
>>> import numpy as np
>>> import pandas as pd
>>> import matplotlib.pyplot as plt
>>> import statsmodels.formula.api as smf

>>> # generate a random dataset with two variables
>>> df = pd.DataFrame(np.random.normal(0, 1, (100, 2)))
>>> df.columns = ['x', 'y']

>>> # develop a linear regression model (for comparison)
>>> x = df['x']
```

```
>>> y = df['y']
>>> fit = np.polyfit(x, y, deg=1)
>>> _x = np.linspace(x.min(), x.max(), num=len(y))

>>> # develop the quantile regression model for six quantiles
>>> model = smf.quantreg('y ~ x', df)
>>> quantiles = [0.05, 0.1, 0.25, 0.5, 0.75, 0.95]
>>> fits = [model.fit(q=q) for q in quantiles]

>>> # the quantile lines
>>> _y_005 = fits[0].params['x'] * _x + fits[0].params['Intercept']
>>> _y_095 = fits[5].params['x'] * _x + fits[5].params['Intercept']

>>> # start and end coordinates of the quantile lines
>>> p = np.column_stack((x, y))
>>> a = np.array([_x[0], _y_005[0]]) #first point of 0.05 quantile fit line
>>> b = np.array([_x[-1], _y_005[-1]]) #last point of 0.05 quantile fit line
>>> a_ = np.array([_x[0], _y_095[0]])
>>> b_ = np.array([_x[-1], _y_095[-1]])

>>> #mask for coordinates above 0.95 or below 0.05 quantile lines
>>> mask = lambda p, a, b, a_, b_: (np.cross(p-a, b-a) > 0) | (np.cross(p-a_,
    b_-a_) < 0)
>>> mask = mask(p, a, b, a_, b_)

>>> # generate the plots
>>> figure, axes = plt.subplots()
>>> axes.scatter(x[mask], df['y'][mask], facecolor='r',
edgecolor='none', alpha=0.3, label='data point')
<matplotlib.collections.PathCollection object at 0x0000005763CCCEF0>
>>> axes.scatter(x[~mask], df['y'][~mask], facecolor='g',
edgecolor='none', alpha=0.3, label='data point')
<matplotlib.collections.PathCollection object at 0x000000576533D2E8>
>>> axes.plot(x, fit[0] * x + fit[1], label='best fit', c='lightgrey')
[<matplotlib.lines.Line2D object at 0x0000005765F7B668>]
>>> axes.plot(_x, _y_095, label=quantiles[5], c='orange')
[<matplotlib.lines.Line2D object at 0x0000005763CCCDA0>]
>>> axes.plot(_x, _y_005, label=quantiles[0], c='lightblue')
[<matplotlib.lines.Line2D object at 0x0000005763CCC6A0>]
>>> axes.legend()
<matplotlib.legend.Legend object at 0x0000005763CCC9B0>
>>> axes.set_xlabel('x')
Text(0.5, 0, 'x')
>>> axes.set_ylabel('y')
Text(0, 0.5, 'y')
>>> plt.show()
```

Figure 42: Quantile Regression Model: Boston Housing Data



Source: Walid Hussein, 2020.

Code

```
>>> ## Note that you can plot also the quantile fits for the other
quantiles: 0.1, 0.25, 0.5, and 0.75
>>>
>>> ## Note that data points colored in green can be considered as
outliers and removed later from the dataset for better interpretation
```

The final implementation is the Ridge and Lasso regression approaches, in comparison to the simple linear regression model, as provided below. The dataset that has been utilized is the house sales in King County dataset (2016), USA, which is available on the Kaggle website.

Code

```
>>> import pandas as pd
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> from sklearn.linear_model import LinearRegression, Ridge, Lasso
>>> from sklearn.model_selection import train_test_split,
cross_val_score
```

```
>>> from statistics import mean

>>> # Loading the data
>>> data = pd.read_csv('kc_house_data.csv')

>>> # Drop the non-numerical variables and those with missing values
>>> dropColumns = ['id', 'date', 'sqft_above', 'zipcode']
>>> data = data.drop(dropColumns, axis = 1)

>>> # Determine the dependent and independent variables
>>> y = data['price']
>>> X = data.drop('price', axis = 1)

>>> # Divide the data into training and testing set
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)

>>> # Develop a Linear Regression model
>>> linearModel = LinearRegression()
>>> linearModel.fit(X_train, y_train)
LinearRegression(copy_X=True, fit_intercept=True, n_jobs=None, normalize=False)

>>> # Evaluate the Linear Regression model
>>> print(linearModel.score(X_test, y_test))
0.7151818916082039

>>> # Develop Ridge(L2) Regression Model:
>>> # Estimate different values for lamda
>>> alpha = []
>>> cross_val_scores_ridge = []

>>> # Loop to compute the different scores
>>> for i in range(1, 9):
     ridgeModel = Ridge(alpha = i * 0.25)
     ridgeModel.fit(X_train, y_train)
     scores = cross_val_score(ridgeModel, X, y, cv = 10)
     avg_cross_val_score = mean(scores)
     cross_val_scores_ridge.append(avg_cross_val_score)
     alpha.append(i * 0.25)


Ridge(alpha=0.25, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
Ridge(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
Ridge(alpha=0.75, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
Ridge(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
```

```
Ridge(alpha=1.25, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
Ridge(alpha=1.5, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
Ridge(alpha=1.75, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)
Ridge(alpha=2.0, copy_X=True, fit_intercept=True, max_iter=None,
  normalize=False, random_state=None, solver='auto', tol=0.001)

>>> # Loop to print the different scores
>>> for i in range(0, len(alpha)):
     print(str(alpha[i])+' : '+str(cross_val_scores_ridge[i]))


0.25 : 0.6909039943128887
0.5 : 0.690905755445855
0.75 : 0.6909073194564496
1.0 : 0.6909086883699769
1.25 : 0.6909098641885285
1.5 : 0.690910848891299
1.75 : 0.690911644434875
2.0 : 0.6909122527535538

>>> # the best value of lamda for the data is 2
>>> # Build the Ridge Regression model for the best lamda
>>> ridgeModelChosen = Ridge(alpha = 2)
>>> ridgeModelChosen.fit(X_train, y_train)
Ridge(alpha=2, copy_X=True, fit_intercept=True, max_iter=None,
normalize=False, random_state=None, solver='auto', tol=0.001)
>>> # Evaluate the Ridge Regression model
>>> print(ridgeModelChosen.score(X_test, y_test))
0.7151525243289275



>>> # Develop Lasso(L1) Regression Model:
>>> # Estimate different values for lamda
>>> lamda = []
>>> cross_val_scores_lasso = []
>>> # Loop to compute the different scores
>>> for i in range(1, 9):
     lassoModel = Lasso(alpha = i * 0.25, tol = 0.0925)
     lassoModel.fit(X_train, y_train)
     scores = cross_val_score(lassoModel, X, y, cv = 10)
     avg_cross_val_score = mean(scores)
     cross_val_scores_lasso.append(avg_cross_val_score)
     lamda.append(i * 0.25)
```

```
Lasso(alpha=0.25, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=0.5, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=0.75, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=1.0, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=1.25, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=1.5, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=1.75, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)
Lasso(alpha=2.0, copy_X=True, fit_intercept=True, max_iter=1000,
  normalize=False, positive=False, precompute=False, random_state=None,
  selection='cyclic', tol=0.0925, warm_start=False)


>>> # Loop to print the different scores
>>> for i in range(0, len(alpha)):
     print(str(alpha[i])+' : '+str(cross_val_scores_lasso[i]))


0.25 : 0.6909020889338366
0.5 : 0.6909021436304988
0.75 : 0.6909021980337026
1.0 : 0.6909022521931557
1.25 : 0.6909023060760678
1.5 : 0.6909023596857342
1.75 : 0.6909024130374362
2.0 : 0.6909024661093868

>>> # the best value of lamda for the data is 2
>>> # Build the Lasso Regression model for the best lamda
>>> lassoModelChosen = Lasso(alpha = 2, tol = 0.0925)
>>> lassoModelChosen.fit(X_train, y_train)
Lasso(alpha=2, copy_X=True, fit_intercept=True, max_iter=1000,
normalize=False, positive=False, precompute=False, random_state=None,
selection='cyclic', tol=0.0925, warm_start=False)
```

```
>>> # Evaluate the Lasso Regression model
>>> print(lassoModelChosen.score(X_test, y_test))
0.7151811532899977
```

SUMMARY

This unit details different algorithms for performing regression analysis. These algorithms include simple linear regression, logistic regression, quantile regression, and their implementations in Python. The suitability of each algorithm for a particular case is elaborated. Regularization of the regression analysis to solve for the problem of overfitting is also explained through the routinely used regularization techniques: ridge and lasso regression.

UNIT 4

SUPPORT VECTOR MACHINES

STUDY GOALS

On completion of this unit, you will have learned …

- about the support vector machine (SVM) model.
- what is meant by the term hyperplane and how it is represented.
- the difference between linearly and nonlinearly separable data.
- what the kernel function is and how it is used in the SVM algorithm.
- what is meant by the kernel trick.

4. SUPPORT VECTOR MACHINES

Introduction

Support Vector Machine (SVM) is a mathematical model within the class of supervised machine learning algorithms. This approach is widely used in various domains that require high discrimination power. It is used in decision-making applications that require accurate classification and prediction algorithms. Moreover, this algorithm can operate on both continuous and categorical data and can solve both linear and nonlinear problems. This allows the SVM algorithm to work efficiently and effectively on various **real-life** problems.

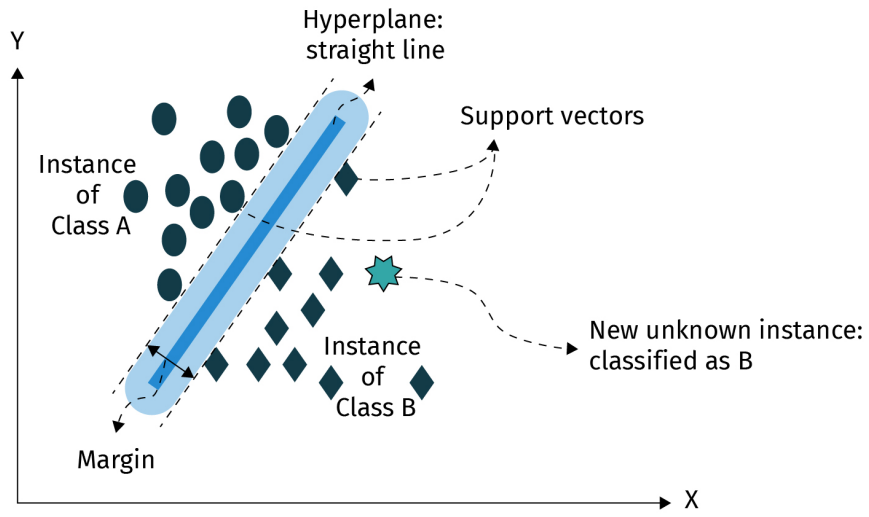
Real-life data
These data are retrieved from different sources like sensors, surveys, and observations in practical application scenarios.

4.1 Introduction to Support Vector Machines (SVM)

In its basic form, the SVM algorithm is a binary classifier that maps entities to one of two possible output classes. It is assumed that these entities can be represented as a set of points in a multi-dimensional space where each dimension corresponds to one of the entities' properties. The two distinct classes of the classification problem then correspond to different subspaces of the feature space that are occupied by the respective entities. The SVNM model assumes that these subspaces can be separated by subspace that has one dimension less than the overall feature space. This sub-space is the separating hyperplane. Each side of the hyperplane, thus, represents a class of the underlying classification problem. This hyperplane is detected (learned) during the training phase. While in the testing or application phase, the algorithm assigns a class to any new hitherto unlabeled instance. The classification is based on the location of the data point corresponding to that instance, relative to the learned hyperplane. Consider the example in the following figure that contains the instances of two classes, A and B. Each instance is represented by a data point in a two-dimensional feature space (X,Y) and has a shape that corresponds to its labeled class. The SVM works as follows:

1. The position of the hyperplane is derived in the training phase of the SVM algorithm.
2. In the testing phase, the new non-labeled instance (represented as the star shape) is classified as class B according to its location relative to the hyperplane.

Figure 43: Hyperplane in SVM



Classification of the instances of two classes (Class A[●] and Class B[◆])

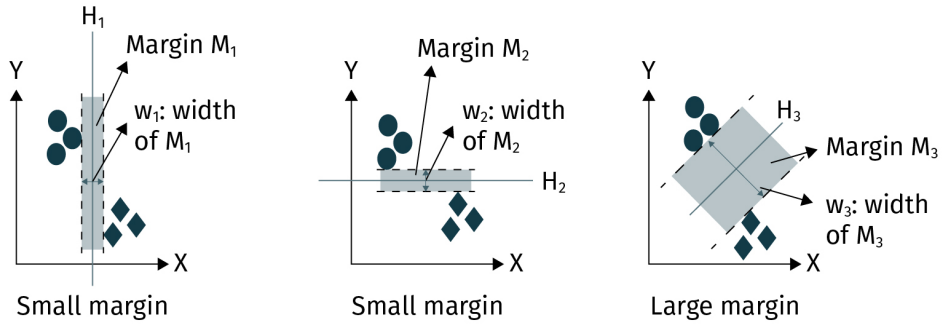
Source: Walid Hussein, 2020.

The hyperplane above is represented as a straight line that passes through the middle of a gap at the boundary between the two point-sets. This gap lies between the closest instances of the different classes. This boundary gap is defined as the margin of the hyperplane; the closest instances that determine the margin are defined as the support vectors. These vectors "support"" the separating hyperplane. The goal of the SVM algorithm is to detect (or learn) the hyperplane that maximizes the width of the margin. The width of this margin is measured based on the perpendicular distance from the separating hyperplane to the support vectors. This algorithm is memory efficient, as it uses only a subset of training points (support vectors) in the classification of the testing instances. The large margin ensures good generalization of the classifying algorithm in discriminating between the instances of the two classes.

SVM Example

Consider a problem of six instances of two different classes, and each instance has two features that describe its measured characteristics. Accordingly, this problem can be represented as a two-dimensional space of X-axis and Y-axis. This space contains six data points, and each class in the two classes of the problem contains three points of the same shape. The figure below shows three potential hyperplanes H_1, H_2, and H_3, between the two classes. The width w_1 and the width w_2 of the margins M_1 and M_2 are small compared to the width w_3 of the Margin M_3. Then the hyperplane H_3 is optimally separating the two classes with the largest margin M_3 with the widths: $\mathrm{w}_3 > \mathrm{w}_1$ and $\mathrm{w}_3 > \mathrm{w}_2$. The support vectors are the data points that lie on the dashed lines shown below. These points are used to establish the margin between the two classes.

Figure 44: Optimal Hyperplane in SVM (Two-Dimensional Space)



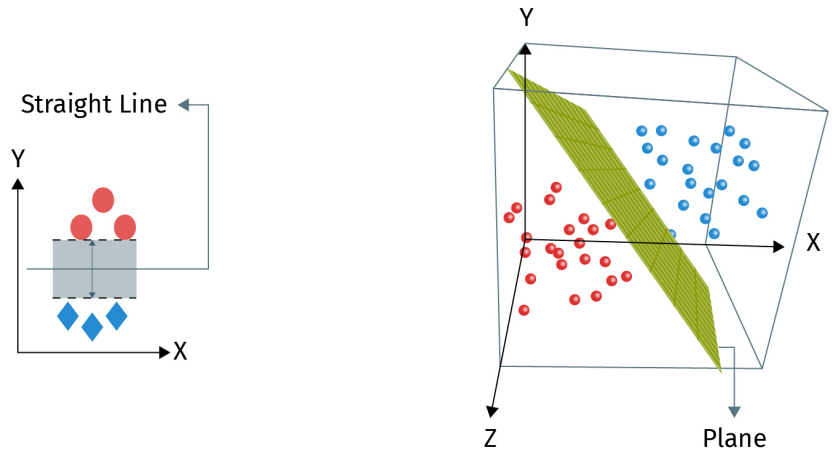
Source: Walid Hussein, 2020.

Since the instances of this problem have only two features, they are mapped to two-dimensional data points, and the separating hyperplane is represented by a solid line.

The Hyperplane

The hyperplane in SVM forms a space that separates the instances of different classes. This hyperplane is a straight line if the space is two-dimensional, and is a plane if the space is three-dimensional, and so on. In other words, in n-dimensional space, the hyperplane constitutes an $(n-1)$-dimensional subspace. The following figure shows an example of a two-dimensional and three-dimensional feature space. Each dimension in the space represents a feature or characteristic of the classified instance. Instances in real-life problems are usually characterized by hundreds of features, making these settings impossible to visualize.

Figure 45: Optimal Hyperplane in SVM (Three-Dimensional Space)



- 2-dimensional space problem
- Hyperplane is a line (1-dim)

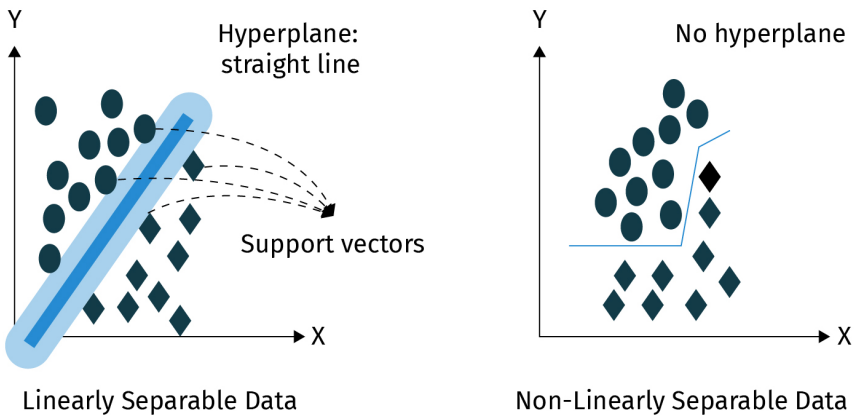
- 3-dimensional space problem
- Hyperplane is a plane (2-dim)

Source: Walid Hussein, 2020.

4.2 SVM for Classification

The SVM classifier is defined as a linear binary classifier. The term binary refers to the number of categories of the problem instances. Every new instance x exists in only one side from the two sides of the hyperplane. The term linear refers to that the fact that SVM operates on problems whose instances are linearly separable. The problem instances are considered as linearly separable data points only if a separating hyperplane can be constructed, as this hyperplane is described by a single linear equation. Real-life problems are usually not linearly separable, and in this case the separating line or plane can be a curved line or curved plane, as shown in the following figure. How the SVM approach deals with these kinds of settings is described in subsequent sections (Schölkopf & Smola, 2001).

Figure 46: Linearly Separable Data



Source: Walid Hussein, 2020.

Linearly Separable Problems (Hard Margin)

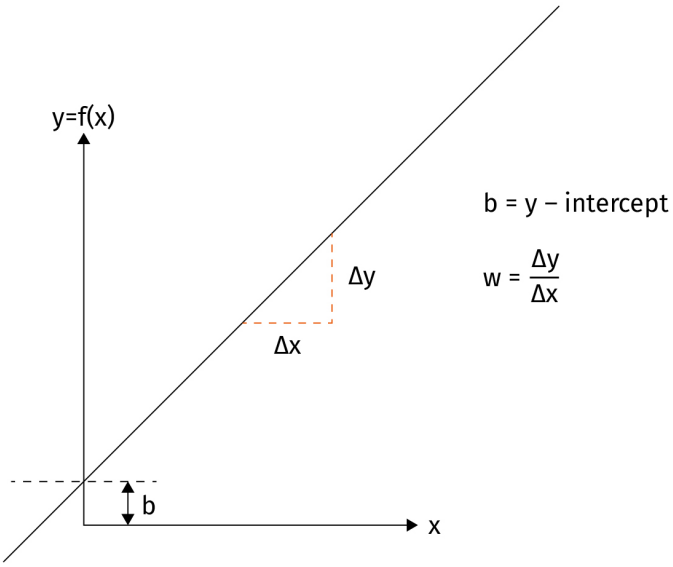
SVM searches for a hyperplane that maximizes the margin between binary labeled instances. In the case of linearly separable instances, a separating hyperplane without training errors, of a hard margin, can be detected. The SVM algorithm uses the training instances to construct the linear function of the hyperplane. The goal is to use that description of the hyperplane to construct a function that maps each training instance x to a function value $\mathrm{f}(\mathrm{x})$ for which holds $\mathrm{f}(\mathrm{x}) > 0$ for instances of one class, and $\mathrm{f}(\mathrm{x}) < 0$ for members of the other class.

Let us first look at the problem of a one-dimensional separation, i.e., a line. The general equation of a straight line (referred to as equation 1) is as follows:

$$\mathrm{y}(\mathrm{x}) = \mathrm{wx} + \mathrm{b}(1)$$

The value $\mathrm{y}(\mathrm{x})$ changes according to the change in the x variable, where w and b are constant values. The constant w represents the slope of the straight line; its value is equal to the ratio of the change $\Delta\mathrm{y}$ in the value of $\mathrm{y}(\mathrm{x})$ to the corresponding change $\Delta\mathrm{x}$ in the value of x. The value of b represents the intercept of the straight line with the y-axis. The figure below shows how the points in the straight line vary on the x-axis (value of x) and the y-axis (value of $\mathrm{y}(\mathrm{x})$). From this we can construct the function $\mathrm{f}(\mathrm{x})$ in a straightforward manner as $\mathrm{f}(\mathrm{x}) = \mathrm{y}(\mathrm{x}) = \mathrm{wx} + \mathrm{b}$.

Figure 47: Formation of the Hyperplane



Source: Walid Hussein, 2020.

For a dataset of multiple features, the decision function $\mathrm{f(x)}$ is a linear combination of the n features $(\mathrm{x_1}$,$\mathrm{x_2}$, ...,$\mathrm{x_n})$. These n-features represent a single instance x in the dataset. The values of the instance x's features are mapped linearly to the decision value $\mathrm{f(x)}$ in analogy to equation (1). Below is equation 2:

$$\mathrm{f(x) = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_n \cdot x_n + b} (2)$$
$$= \sum_{\mathrm{i=1}}^{\mathrm{n}} \mathrm{w_i \cdot x_i + b}$$
$$= \mathrm{w \,.\, x + b}$$

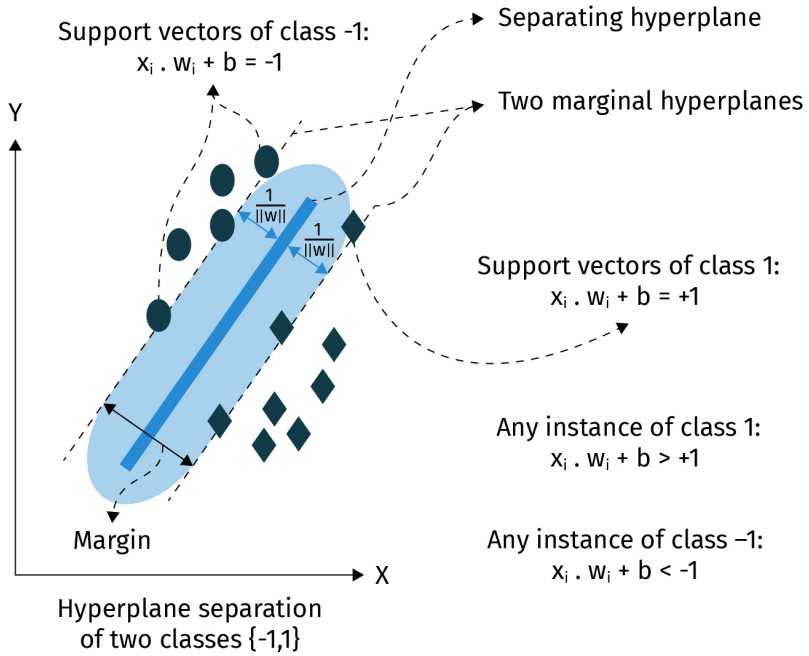
where **.** denotes the dot product.

Here, $\mathrm{w} = (\mathrm{w_1}, \quad \mathrm{w_2}, \ldots, \quad \mathrm{w_n})$ is the weight vector that describes the linear function of the SVM's hyperplane, which is a normal vector to the separating hyperplane. Any point on the hyperplane has a decision value $\{\mathrm{f(X) = 0}\}$ since this point cannot be classified in either of the two sides of the plane. In the training phase, the decision value of any instance is either $\{\mathrm{f(X) > 0}\}$ if the data point of the instance is on one side of the hyperplane, or $\{\mathrm{f(X) < 0}\}$ if this data point lies on the other side of the hyperplane. The determination of the hyperplane linear equation (2) requires the calculation of the weight vector w. The input to the SVM classifier is a training set of m pairs $(\mathrm{x_i}, \quad \mathrm{y_i})$, such that each instance $\mathrm{x_i}$ has a known label $\mathrm{y_i}$. The class label $\mathrm{y_i}$ has two possible values {1, –1}. The required output of the training phase is the calculation of the w vector and the displacement value b, such that the margin between the support vectors is maximized by this specific hyperplane.

As shown in the following figure, the separating hyperplane lies in the middle between parallel marginal hyperplanes. Each marginal hyperplane is supported by a set of critical training instances (support vectors) that lie on it. Since rescaling the feature samples does not change their linear separability, we can transform the input data in such a way that our classification function assumes the value 1 for all support vectors of the first class and –1 for all support vectors in the second class. Having performed this transformation step, the equation of the two class decision problem can be stated as follows

$$f(x_i) = x_i \,.\, w + b \geq +1, \text{ if } \quad y_i = 1 (\text{for all vectors of class } 1)$$
$$f(x_i) = x_i \,.\, w + b \leq -1, \text{ if } \quad y_i = -1 (\text{for all vectors of class -1})$$

Figure 48: Hyperplane Separation



Source: Walid Hussein, 2020.

The distance between the two marginal hyperplanes (margin width) is $\left\{\frac{2}{||w||}\right\}$. The problem here is to find the most appropriate (optimal) w and b that maximize this margin width $\left\{\frac{2}{||w||}\right\}$ to reach a high classification accuracy and good generalizability. This optimization problem can be initially stated as follows in the equation 2

Maximize

$$\left\{\frac{1}{||w||}\right\}$$

subject to

$$y_i f(x_i) = y_i(x_i \,.\, w + b) \geq +1 (2)$$

And for simplicity, the maximization of the margin width $\left\{\frac{2}{||w||}\right\}$ is equivalent to the minimization of $\{||w||\}$.

Accordingly, the optimization problem can be simplified and stated finally as shown in the equation 3:

Minimize

$$\{||w||\}$$

subject to

$$y_i f(x_i) = y_i(x_i \,.\, w + b) \geq +1 (3)$$

The product $y_i f(x_i) \geq +1$ indicates that the two values, which are the class label y_i of instance i and the predicted value $f(x_i)$ of this instance, are of the same sign. This means that i is classified correctly according to the hyperplane linear function $f(x_i)$. If the value of $y_i \,.\, f(x_i)$ of each instance x_i is less than one, then this instance is misclassified. The error in classifying this instance x_i is used to adjust the vector w such that the value of $\|w\|$ is minimized, and the constraint is fulfilled. The algorithm repeats calculating the value of $y_i f(x_i)$ and adjusting the weight vector until all instances are classified correctly, and the maximum margin is reached.

However, it is not always possible to find a hyperplane {represented by w and b}, where all instances are satisfying this optimization formula (hard margin). It pre-supposes that all training instances are on the correct side of the hyperplane, and this case is often not feasible in real-life data.

Nonlinearly Separable Problems (Soft Margin/Primal Form)

The formula of the hard margin is extended to consider data instances that lie on the wrong side of the margin. This adjusted form of the hard margin is named as the primal form of soft margin. This form allows SVM to classify instances that are almost linearly separable. The constraint of the optimization problem in the case of soft margin is as follows

$$y_i(x_i \,.\, w + b) \geq +1 (\text{constraint in } 3)$$
$$1 - y_i(x_i \,.\, w + b) \leq 0 (\text{constraint in } 3)$$

In the case of non-linearly separable data, the SVM margin is softened by allowing any instance i to violate the separation constraint by a slack error ξ_i. This constraint can be represented as follows

$$1 - y_i(x_i \,.\, w + b) \leq \xi_i (\text{adjusted form of constraint in } 3)$$

This error is considered the loss function of having instances on the wrong side of the hyperplane. The loss function can be formulated as follows to consider all instances on the correct and on the wrong side of the hyperplane:

$$\xi_i = \max(0, 1 - y_i \cdot (x_i \,.\, w + b))(4)$$

For every instance i on the correct side, $\xi_i = 0$

For every instance i on the wrong side, $\xi_i = 1 - y_i(x_i \,.\, w + b)$, where in this case, the value of the loss function is proportional to the distance of this incorrectly classified instance from the marginal hyperplane.

Primal form
The constraint optimization problem for detecting a soft-margin is known as primal form. The terms primal and dual are taken from the nomenclature used in Lagrangian Optimization.

Accordingly, the optimization problem of the hard margin (3) can be relaxed to state the **primal form** of the soft margin as follows

Minimize

$$\frac{C}{n} \sum_{i=1}^{n} \xi_i + ||w||^2 (5)$$

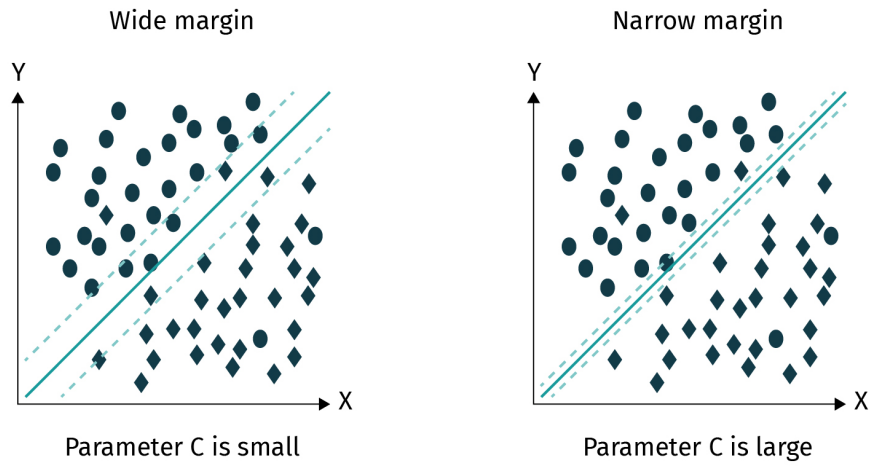
subject to

$$y_i \cdot (x_i \,.\, w + b) \geq 1 - \xi_i, \text{ and } \quad \xi_i \geq 0$$

where n represents the number of instances in the training data of the problem. The regularization parameter C is a problem-dependent parameter. Its value refers to penalty strength of having misclassifications. As shown in the following figure, the parameter C is considered as a trade-off between:

- If the value of C is small, then the classifier can decrease or ignore the penalty of misclassified instances. In this case, the classifier considers that the data are linearly separable, and the separating hyperplane is smooth and of a hard margin.
- If the value of C is large, then the classifier does not allow misclassification where the penalty strength increases. In this case, the margin is enlarged to form a soft margin that is appropriate for nonlinearly separable data. This is because the closer points get more weight, and this results in the decision boundary which then takes on a piecewise linear form.

Figure 49: Large versus Small C Parameter



Source: Walid Hussein, 2020.

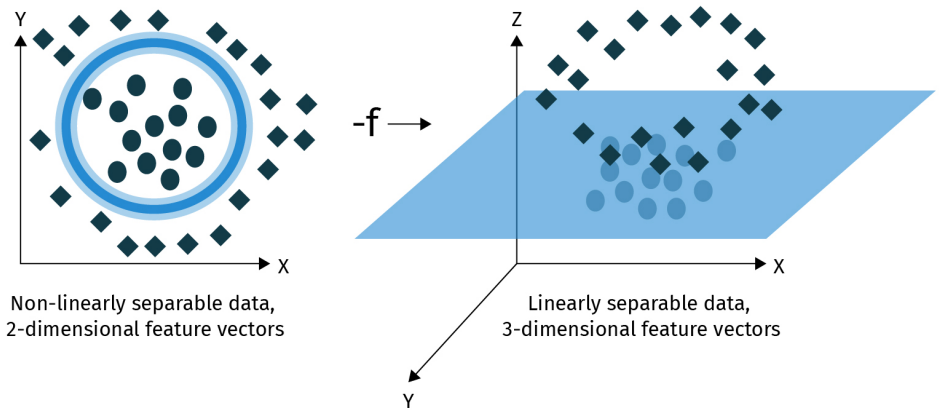
Based on the input data for small values of the C parameter, the classification accuracy decreases as more error is allowed. For larger values of the C parameter, the classification accuracy increases due to overfitting of the training data.

Transformation to Linearly Separable Data

The SVM linear classifier cannot operate effectively on nonlinearly separable data. However, real-life datasets are usually not linearly separable, making it impossible to construct a separating hyperplane. A possible solution for this problem is to transform the nonlinearly separable data to linearly separable data. This transformation typically involves increasing the number of dimensions in the feature space of the data. The original feature space is converted to a higher-dimensional feature space. To show an example of this transformation, consider a nonlinearly separable problem as shown in the following figure. There is no straight line (linear hyperplane) that can separate between the instances of the two classes. The decision boundary that can separate between the two classes is a circle. Accordingly, a new dimension Z is added to the two-dimensional feature vector (X,Y), to form a three-dimensional space instead of a two-dimensional space. The added dimension represents the radius of the circular boundary as shown in equation (6). Thus, the data becomes linearly separable along the z-axis. The data in this case are separable by the following plane:

$$\mathrm{Z} = \mathrm{X}^2 + \mathrm{Y}^2 (6)$$

Figure 50: Mapping Function f



Source: Walid Hussein, 2020.

This transformation process performs an explicit mapping from an n-dimensional feature space to a higher m-dimensional feature space $(m > > n)$. While solving the problem of linearly separable data, this principle suffers from two main problems. Overfitting is the first problem that is caused by the curse of dimensionality. The second problem is the high cost in terms of the computational power needed due to the large size of the transformed feature vectors.

Nonlinearly Separable Problems (Soft Margin/Dual Form)

For nonlinearly separable problems, the SVM replaces the explicit transformation of the underlying feature space by what is referred to as the kernel trick, to reduce the expensive computational cost. In this setting, the SVM algorithm adjusts the primal form of the optimization problem to what is called the dual form. The dual form of SVM uses a Lagrangian multiplier α in defining the weight vector as follows

$$w = \sum_{j=1}^{n} \alpha_j y_j x_j (7)$$

This defined weight is used in the loss function calculation (4) to obtain the dual form as follows:

Subject to specific linear constraints, the problem is to find a multiplier α that maximizes the value of a quadratic function Q.

Maximize

$$Q(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i \alpha_i \left(x_i \, . \, x_j\right) y_j \alpha_j (8)$$

subject to

$$\sum_{i=1}^{n} \alpha_j y_j = 0$$

and for all $i, 0 \leq \alpha_j \leq \frac{1}{2nC}$.

For nonlinearly separable data, the original feature space x_i is replaced by another feature space $\varphi(x_i)$. The mapping function φ performs an explicit transformation of the feature space of an instance x_i to higher dimension feature space $\varphi(x_i)$. Accordingly, the dot product $(x_i \,.\, x_j)$ in the dual form of the optimization problem is replaced by $(\varphi(x_i) \,.\, \varphi(x_j))$. Fortunately, the dot product between pairs of mapped data vectors $(\varphi(x_i) \,.\, \varphi(x_j))$ has the same value as the kernel function K of the original (non-mapped) feature space of these vectors x_i and x_j. The mathematical definition of the kernel function is as follows

$$K(x_i, x_j) = \varphi(x_i) \,.\, \varphi(x_j) (9)$$

The kernel function $K(x_i,x_j)$ measures the dot product of two data vectors x_i,x_j. Additionally, the kernel function $K(x_i,x_j)$ can be used as an indicator of the similarity between pairs of data vectors x_i and x_j, in the sense that the distance between two data vectors is inversely proportional to the similarity between these vectors. The SVM kernel trick is the replacement of every dot product by a nonlinear kernel function. This replacement avoids the explicit transformation of the feature space to a higher dimensional feature space. This is because the kernel function is applied to the original feature space instances without transformation.

In the case of linearly separable data, the mapping function $\varphi(x_i)$ does not change the dimension of the input vector, such that $\varphi(x_i) = x_i$. Accordingly, the kernel function K of any two vectors x_i and x_j is the dot product of these vectors $(x_i \,.\, x_j)$ in their original space. The mathematical form of the linear kernel function is

$$K(x_i,x_j) = x_i \,.\, x_j (10)$$

In the case of nonlinearly separable data problems, the calculations of the kernel function are applied on the original feature space. However, the dot product between every pair of vectors is replaced by the non-linear kernel function. Various nonlinear forms of the kernel function are available like polynomial, radial basis function (RBF), and sigmoid kernels. The selection of the kernel function form is dependent on the problem input data. The polynomial kernel allows representation of curved lines in the input space, while the radial kernel allows closed curved lines, such as circles and closed polygons in the input space. Here are some common forms of the kernel function:

- Polynomial kernel: d is the degree of the polynomial

$$K(x_i,x_j) = (x_i \,.\, x_j + 1)^d (11)$$

- Gaussian kernel (example of radial basis function kernel):

$$K\left(x_i, x_j\right) = e\left(-\frac{\left\|x_i - x_j\right\|^2}{2\sigma^2}\right) = e\left(-\gamma\left\|x_i - x_j\right\|^2\right) (12)$$

  The gamma $\gamma = \frac{1}{2\sigma^2}$ is inversely proportional to the Gaussian width σ. It defines the radius of influence of the training instances. if the gamma value is high, then only close instances are affecting the learning outcomes. If the gamma value is low, then far-away instances have an influence on the classification problem too.

- The exponential kernel:

$$K\left(x_i,x_j\right) = e\left(-\frac{\left\|x_i - x_j\right\|}{\sigma}\right) (13)$$

- The Laplacian kernel:

$$K\left(x_i,x_j\right) = e\left(-\frac{\left\|x_i - x_j\right\|}{2\sigma^2}\right) (14)$$

- Sigmoid kernel:

$$K\left(x_i,x_j\right) = \tanh\left(kx_i^T . x_j + c\right) (15)$$

4.3 SVM for Regression (SVR)

The SVR is the regression version of the SVM classifier. Instead of generating a hyperplane for the prediction of instance class labels in SVM, SVR predicts a numerical output value for each instance. It is typically not feasible to find a linear function that computes the target values for all the instance points exactly, without any errors. In other words, for any instance (x_i,y_i), it is hard to construct a function f such that $f(x_i) = y_i$ for every instance i. The goal of SVR is to find a predicting function $f(x_i)$ for instances x_i that tolerates some deviation (epsilon error) from the actual target y_i for all the training instances. This deviation is allowed if it is not greater than a threshold epsilon (ε) for each instance x_i. Furthermore, SVR provides the ability to define how much error is acceptable in the resulting model by adjusting ε.

The objective function of the SVR model is to minimize the coefficient w of the model, not the squared error. The SVR model uses an error term denoted as ε in the problem constraint as depicted in the following figure. This constraint states that the absolute error must be less than or equal to ε. The value of ε can be tuned to reach the desired accuracy of the regression model. The simplified form of the constraint optimization problem in this model can be stated as follows

Minimize

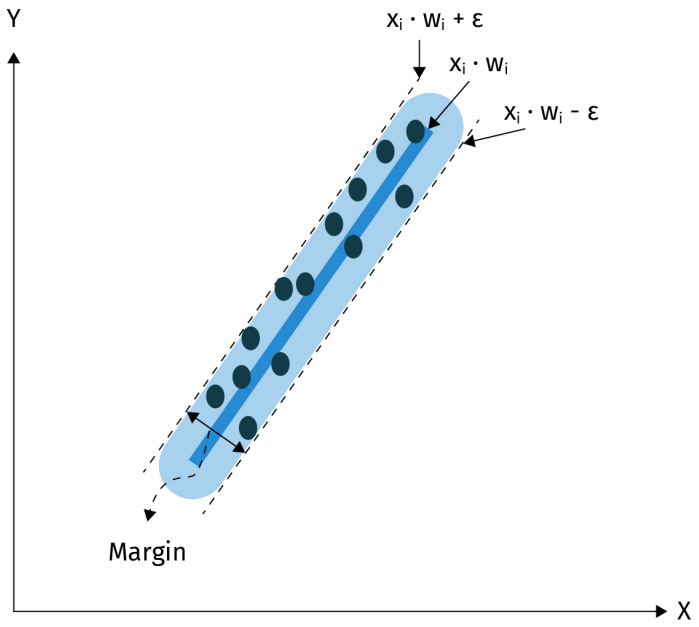
$$\frac{1}{2}||\mathrm{w}||^2 (16)$$

subject to

$$|\mathrm{y_i} - \mathrm{f}(\mathrm{x_i})| \leq \varepsilon$$
$$\mathrm{f}(\mathrm{x_i}) = \mathrm{w_i x_i}$$

where $\mathrm{i} = 0, ..., \mathrm{n}, \mathrm{w}_0 = \mathrm{b}$ and $x_0 = 1$.

Figure 51: ε in SVR



Source: Walid Hussein, 2020.

To consider the error values that are larger than ε for the points that lie outside the margins, slack variables are introduced, analogous to the soft margin SVM. The cost C represented by the slack variable ξ_i is imposed only on instances that lie outside the margin ε. The SVR regression model uses two slack variables $\xi\mathrm{i}\,.\,\xi\mathrm{i}\,*$ such that the constraint is formulated as follows

Minimize

$$\frac{1}{2}||\mathrm{w}||^2 + \mathrm{C}\sum_{\mathrm{i}=1}^{\mathrm{n}} |\xi_\mathrm{i} + \xi_\mathrm{i}^*| (17)$$

subject to

$$|y_i - f(x_i)| \leq \varepsilon + |\xi_i|$$
$$f(x_i) = w_i x_i$$

and

$$|f(x_i) - y_i| \leq \varepsilon + |\xi_i{}^*|$$
$$f(x_i) = w_i x_i$$

The additional cost parameter C can also be tuned to manage the tolerance to points outside of ε. This parameter is considered as a trade-off parameter between the flatness of the function f and the amount up to which deviations larger than ε are tolerated. As the value of C decreases, the tolerance for points outside of ε increases. And as the value of C approaches zero, the constraint collapses into the simplified form.

Case Study: Implementation of SVM in Python

Consider a dataset of a specific problem saved in a trialSVM.csv file. This file contains a title row and 100 data rows, each row containing six columns. The first five columns represent the data of a five-featured-instance, with the features of the instances lled $\{a, b, c, d, e\}$. The feature values are continuous values in the range from 0 to 100. The last column represents the class label of the corresponding instance {class}. The categorization of the class labels in this problem is binary, either Class 0 or Class 1. The class labels are equally (fairly) distributed, such that half of the input data is labeled as Class 0, and the other half is labeled as Class 1.

Table 10: Mapping Function f

a	b	c	d	Class
4.3634	0.46351	1.4281	2.0202	0
3.482	–4.1634	3.5008	–0.07846	0
0.51947	–3.2633	3.0895	–0.98492	0
2.3164	–2.628	3.1529	–0.08622	0
–1.8348	11.0334	3.1863	–4.8888	0
–1.7279	–6.841	8.9494	0.68058	1
–3.3793	–13.7731	17.9274	–2.0323	1
–3.1273	–7.1121	11.3897	–0.08363	1
–2.121	–0.05588	1.949	1.353	1
–1.7697	3.4329	–1.2144	–2.3789	1

Source: Walid Hussein, 2020.

This problem can be solved by a support vector machine algorithm that is implemented in sklearn (Scikit-Learn) library for the Python programming language. The steps of applying the SVM algorithm can be summarized as follows: (1) Reading and exploring the data, (2) Data preprocessing, (3) Training phase, and finally (4) Testing (or evaluating) phase.

Reading and Preprocessing of the Input Data

Consider a dataset of a specific problem is saved in a trialSVM.csv file. Read the data from trialSVM file using the read_csv function in the pandas library:

Code

```
import pandas as pd
trialdata = pd.read_csv("E:/trialSVM.csv")
```

The data preprocessing involves the dividing of the data into attributes and labels. The value of the shape property is the number of rows and columns.

Code

```
print(trialdata.shape)
x = trialdata.drop('Class', axis=1)
y = trialdata['Class']
```

The preprocessing function that receives the name of the used kernel.

Code

```
def classify(kernelName):
```

Then, the data are divided into training(x,y) and testing(x,y) sets.

Code

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size = 0.20)
```

Training of the SVM Learning Model

The function for the training step of the support vector machine has an important parameter, which is the kernel type. The kernel can be linear, polynomial, Gaussian, or sigmoid. This code uses Scikit-Learn/svm library that contains SVC Classifier class. This class takes one parameter, which is the kernel type. The training method of the SVM is the fit function to learn the SVM model.

Code

```
from sklearn.svm import SVC
if kernelName == 'poly':
                svclassifier = SVC(kernel='poly', degree=8)
```

```
                print(kernelName)
else:
                svclassifier = SVC(kernel=kernelName)
                print(kernelName)
svclassifier.fit(x_train, y_train)
```

Testing and Evaluation of the SVM Learning Model

The testing step of the support vector machine involves the prediction of the class labels of the instances x_test in the testing part of the dataset. The predict function returns the predicted class labels y_pred. Then the code uses the accuracy_score function in the Scikit-Learn/metrics library to calculate the classification accuracy of the SVM classifier. The function finally returns the accuracy value to the calling statement.

Code

```
y_pred = svclassifier.predict(x_test)
from sklearn.metrics import accuracy_score
return accuracy_score(y_test, y_pred)
```

Apply the classification using the kernels, linear, polynomial, Gaussian, and sigmoid.

Code

```
print("Linear Accuracy:",classify('linear'))
print("Polynomial Accuracy:",classify('poly'))
print("Gaussian Accuracy:",classify('rbf'))
print("Sigmoid Accuracy:",classify('sigmoid'))
```

The output of this program is as follows:

Code

```
Linear Accuracy: 0.875
Polynomial Accuracy: 1.0
Gaussian Accuracy: 1.0
Sigmoid Accuracy: 0.75
```

The analysis of the above results show that the sigmoid kernel presented degraded model outputs because it is suitable for binary classification tasks (i.e., to present {0,1} outputs). Meanwhile, the model with the Gaussian kernel achieved the ideal prediction with an accuracy of 100%, while a model with a polynomial kernel misclassified one instance.

SUMMARY

A standard SVM classifier works only if the data categories are well separated by a hyperplane. The detection of the best position of the hyperplane is an optimization problem, which is subject to a margin constraint on the hyperplane. A good solution to classify non-linearly separable data can be attained by converting this data to a linearly separable setting by increasing the number of its dimensions. This solution, however, is not applicable for large datasets with a high number of features due to the computational cost of this transformation process.

The SVM algorithm replaces the transformation process using the kernel trick. This trick involves the replacement of the dot product between two feature-vectors in the transformed feature space by the kernel function. The type of the kernel function is selected according to the type of the input dataset. The kernel function has different types like linear, polynomial, Gaussian, or sigmoid functions.

The SVR algorithm is the regression version the SVM algorithm. Unlike the SVM that maps the input domain to specific classes, the SVR algorithm maps the input domain to real numbers. The mapping function of this algorithm produces an output for all of the training data which has a deviation from the target of at most ε.

UNIT 5

DECISION TREES

STUDY GOALS

On completion of this unit, you will have learned …

- what a decision tree is and how it is used in supervised learning.
- the different approaches to construct a decision tree.
- the purpose and the types of feature-ranking techniques used in decision trees.
- what is meant by tree pruning and whether it occurs before or after the tree construction.
- how a decision tree classifier helps in building up an ensemble model.

5. DECISION TREES

Introduction

Decision trees represent a group of classification techniques that are based on the construction of a tree like structure. This structure represents a set of nested decisions that make use of the provided feature information in a sequential manner to categorize the input object. These classifiers are widely used in various fields such as image processing and character recognition. Many applications in different areas are implementing this classifier, including medicine, financial analysis, astronomy, manufacturing, production, and molecular biology.

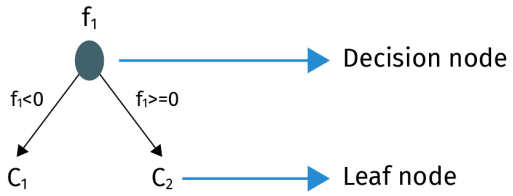
5.1 Introduction to Decision Trees

Decision trees are classifiers and as such belong to the field of supervised learning. Building the decision tree structure and using it for the classification of new samples is a two-step process, made up of a learning step and a decision step. The learning step is the induction of a tree structure based on the training data of labeled data instances. The deciding step includes the classification of testing data of non-labeled instances. Each instance in the dataset of a problem is characterized by a set of features, where the labeled instances in the training set are those of a known class label. The structure of the constructed decision tree includes decision nodes, branches, and leaves. Each internal node in the tree represents a decision based on a feature; the branch from this node is selected based on the value of this feature. Each branch in the tree represents a conditional statement (if-condition). The root node represents the most important feature of the data objects with respect to the classification problem at hand.

Consider a dataset of three instances {1, 2, and 3}, such that each instance is classified by a class label C_1 or by a class label C_2. If each instance is characterized by only one feature f_1, then this feature represents the root node of the constructed tree. The following figure shows the dataset table and the corresponding constructed tree. According to the dataset, if the value of the feature is less than zero, as shown in the condition on the left branch, then the classifier returns the class label C_1 as the result of classification. Otherwise, the classifier returns the class label C_2. This figure shows a simple decision tree, where feature $\mathrm{f}_1$is represented by a decision node that is branched to two leaf nodes representing the classes C_1 and C_2.

Figure 52: Sample Decision Tree 1

Object	f_1	Class
Instance 1	-3	C_1
Instance 2	-2	C_1
Instance 3	4	C_2

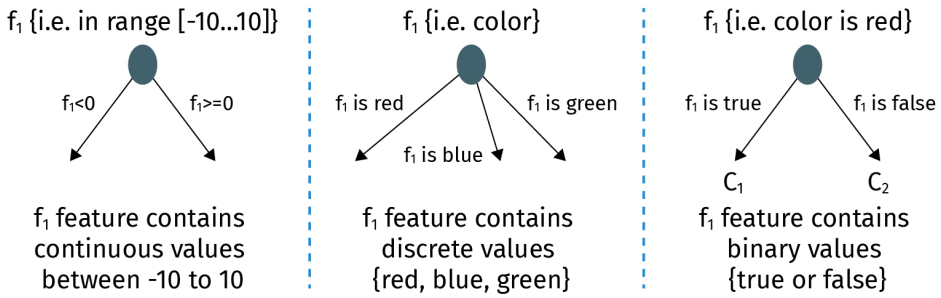


Source: Walid Hussein, 2020.

The feature represented at the internal node is also known as a splitting attribute, and this internal node is known as a decision node. The feature f_1 in the above figure can take on continuous values. In splitting criterion here, zero is the splitting point for the values of this feature. Features can also be discrete in nature. In this case, each branch corresponds directly to one of the distinct possible values $\{a_1,a_2, ...,a_n\}$ of that **discrete feature**. For example, if the feature represents the color of an object, then the branches coming out from this node include red, blue, etc., as shown in the following figure. This figure also shows another example of discrete features, namely one whose values are binary, true or false values.

Discrete feature
A property of a certain value can be characterized as a discrete feature when the value has a finite set of distinct values.

Figure 53: Features of Continuous versus Feature of Discrete Values



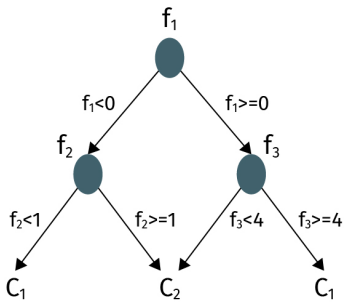
Source: Walid Hussein, 2020.

For a dataset that contains more than one feature, the decision tree classifier uses a ranking technique to detect their degree of importance to the given classification problem. Accordingly, the classifier selects the most salient feature for representing the root node and then the remaining features in decreasing importance for the rest of the tree nodes. The following figure shows an example of a dataset of four objects that are classified into two target class labels C_1 and C_2. Each instance is represented by a row of three features f_1, f_2, and f_3, and a target class. The decision tree classifier constructs a decision tree

based on the values of this dataset during the learning step of the classifier. The root node of this tree pertaining pertains to feature f_1, where f_2 and $\mathrm{f}_3$are sub-nodes from node f_1. In the decision step, the detection of the class of any instance, whether it is C_1 or C_2, is dependent on the values of these features. If the value of feature $\mathrm{f}_1$for an instance is less than zero, then the value of feature f_2 of this instance is tested, otherwise the value of feature $\mathrm{f}_3$of this instance is tested.

Figure 54: Sample Decision Tree 2

Object	f_1	f_2	f_3	Class
Instance 1	-3	0	-1	C_1
Instance 2	-2	5	4	C_2
Instance 3	4	3	4	C_1
Instance 4	2	0	2	C_2



Source: Walid Hussein, 2020.

A decision rule is a group of conditions that enable the classifier to determine the class membership of an instance. Any path from the root node of the tree to a leaf node represents a decision or classification rule. For example, according the branch rules in the above figure, the decision tree classifier assigns instance 1 to C_1based on the following decision rule:

A DECISION RULE

If $\mathrm{f}_1 < 0$ and $\mathrm{f}_2 < 1$, then the class label $= \mathrm{C}_1$

The decision tree algorithm inducts a tree structure that visualizes a set of decision rules. This visual form enables experts of the domain problem to understand the reason for classifying the instances. As the number of decision rules in the inducted tree increases, more training instances are required. In addition, the complexity of the decision rules increases as the number of features increases. Having many of these complex rules increases the size and the complexity of the decision tree. The induction of a complex decision tree reduces the advantage of interpreting its visual form. However, the main problem is that the decision tree begins to fit too closely to a specific set of instances. The occurrence of this phenomenon is known as overfitting, where the decision tree loses its ability to classify new data instances. On the other hand, if the classifying model is not trained enough, the inducted decision tree is going to be too simple to classify instances accurately. This is

called underfitting, which is the opposite of overfitting. A successfully accurate decision tree is a classifying model that has the ability to generalize, which avoids both overfitting and underfitting. The inferred decision tree is effective when it correctly covers as many instances in the training dataset as possible without overfitting. Another problem is that the decision trees may include branches that represent outliers or noise in the input dataset. Decision tree classifiers commonly apply some pruning method to remove these branches in order to improve the classification accuracy when being applied to new samples.

The success behind the various forms of decision tree classifiers is their ability to encode the decision-making knowledge in the form of decision rules. The tree that is constructed in the learning step enables the analyst or decision maker to understand the reason for classifying a tested instance in a particular way. The decision rules in the learned tree are easy to interpret due to this natural representation. Other classifiers such as support vector machines and neural networks are black box classifiers, where the decision logic is unknown. Another advantage of the decision tree classifiers is their independence on the statistical distribution of the input data. Furthermore, these classifiers are able to deal with the input datasets where the relationship between the features and the class labels of the associated objects is nonlinear.

5.2 Decision Tree Approaches for Classification

Different approaches are proposed for learning decision trees. The most popular decision trees are ID3 (Quinlan, 1986), C4.5 (Quinlan, 1993), and CART (Breiman et al., 1984). These approaches differ in the way that features are ranked and selected while creating the decision tree, as well as with respect to the utilized pruning mechanism. Popular feature-ranking techniques used in decision tree approaches are information gain, gain ratio, and Gini index. The ID3 approach uses the information gain technique, the C4.5 approach uses the gain ratio technique (an extension of the information gain technique), and the CART approach uses the Gini index technique.

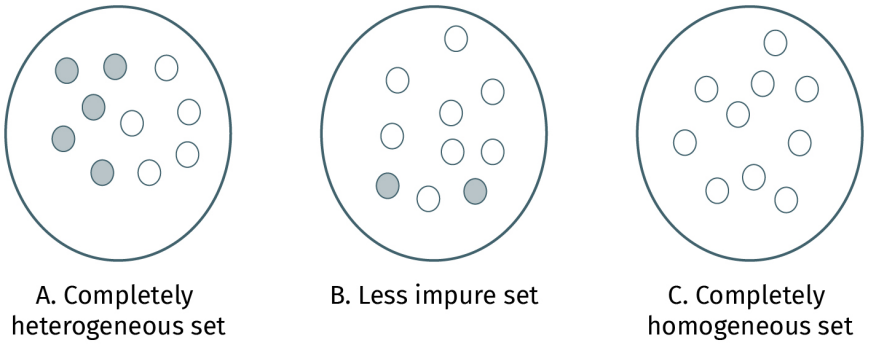
ID3 and Information Gain

ID3, Iterative Dichotomiser, uses the information gain (IG) to select the best splitting features. IG characterizes the distribution of the class data in a feature vector. It measures the degree of homogeneity of the classes induced by a decision node, i.e., a split of feature values. The calculation of the IG of dataset features is based on measuring the **entropy** of the class assignments with respect to a split in that feature. This quantity is inversely proportional to IG. In other words, information gain is the reduction of entropy of class assignment based on the feature split under examination. In this way, the calculation of the amount of information gained from a feature is based on the calculation of its entropy.

Entropy
Generally speaking, the entropy is the degree of disorganization and randomness in a set of observations.

To clarify the concept of entropy, consider the three sets of objects in the following figure. The sample set (a) in this figure has its objects as equally divided between the two existing classes, therefore, the entropy of this set is one. The impurity decreases in set (b) and this manifests in the decreased value of entropy. Finally, the entropy of set (c) is zero since that this set is completely homogenous.

Figure 55: The Entropy of a Sample Set



Source: Walid Hussein, 2020.

The entropy of a dataset D is represented by the function $\mathrm{Info}(\mathrm{D})$. This function measures how the class labels in this dataset can partition its instances. If the number of class labels in the dataset D is m, then the entropy of this dataset is calculated as follows

$$\mathrm{Info}(\mathrm{D}) = \sum_{\mathrm{k}=1}^{\mathrm{m}} -\mathrm{P}_{\mathrm{k}} \cdot \log(\mathrm{P}_{\mathrm{k}}) (1)$$

where P_{k} is the probability that any instance in the dataset D belongs to class k. The probability $_{\mathrm{P}_{\mathrm{k}}}$is calculated as the ratio between the number of instances that belong to k, and the total number of instances in D.

The entropy of a dataset D whose instances are partitioned by a feature f is represented by the function $\mathrm{Info}_{\mathrm{f}}(\mathrm{D})$. This function measures the ability of any feature to split the dataset into equally divided partitions, where each partition corresponds to a single class label. A feature f partitions the instances in the dataset into v subsets, considering that this feature is of v distinct values. The instances in each subset d_{i} has the same value of the feature f. The function $\mathrm{Info}_{\mathrm{f}}(\mathrm{D})$ is calculated as the weighted sum of the entropy of the v subsets, as follows:

$$\mathrm{Info}_{\mathrm{f}}(\mathrm{D}) = \sum_{\mathrm{i}=1}^{\mathrm{v}} \frac{|\mathrm{d}_{\mathrm{i}}|}{|\mathrm{D}|} \cdot \mathrm{Info}(\mathrm{d}_{\mathrm{i}}) (2)$$

The function $\mathrm{Info}(\mathrm{d_i})$ represents the entropy of subset $\mathrm{d_i}$, and term $\left\{\frac{|\mathrm{d_i}|}{|\mathrm{D}|}\right\}$ acts as the weight of this partition $\mathrm{d_i}$. The value of $|\mathrm{d_i}|$ is the number of instances in partition i, and $|\mathrm{D}|$ is the total number of instances.

The entropy $\{\mathrm{Info}(\mathrm{D})\}$ in the dataset D is reduced to $\{\mathrm{Info_f}(\mathrm{D})\}$ when a feature f splits the instances in D. This reduction in the entropy due to a specific feature is equivalent to the information gain of this feature. The information gain of feature f is represented by function $\mathrm{Gain}(\mathrm{f})$, and it is calculated as follows

$$\mathrm{Gain}(\mathrm{f}) = \mathrm{Info}(\mathrm{D}) - \mathrm{Info_f}(\mathrm{D}) (3)$$

The information gain of a feature is inversely proportional to entropy of the dataset when this feature is used. The feature divides the dataset into a group of subsets. If all the values of this feature are distinct, then each subset includes only one instance. Accordingly, the information gain is calculated for discrete features only.

C4.5 and Gain Ratio

The information gain technique provides a higher-ranking value to the features having values that are more distinct. The reason for this behavior is that typically, the purity of the corresponding partitions is high. To illustrate this point, consider the example of an identifier property. By design, all the values in such a feature $\mathrm{f_I}$ are distinct, and this implies that each corresponding partition contains a single instance of a single class. The entropy $\mathrm{Info_{f_I}}\left(\mathrm{D_I}\right)$ of such feature $\mathrm{f_I}$ is zero, as the partition is completely homogenous. So, while splitting the tree based on that feature does yield high information gain, using an ID-field for classification is a poor choice in terms of generalization ability of the resulting classifier. The C4.5 approach handles this problem by normalizing the $\mathrm{gain}(\mathrm{f})$ to form a new value $\mathrm{gainRatio}(\mathrm{f})$. The ranking measure $\mathrm{gainRatio}(\mathrm{f})$ of the feature f is stated as

$$\mathrm{gainRatio}\left(\mathrm{f}\right) = \frac{\mathrm{gain}(\mathrm{f})}{\mathrm{SplitInfo}(\mathrm{f})}$$

The term $\mathrm{SplitInfo}(\mathrm{f})$ is the normalizing factor which is inversely proportional to the ranking measure $\mathrm{gainRatio}(\mathrm{f})$. Its calculation is based on the average sum of the ratio between two numbers: $(|\mathrm{d_i}|)$ which is the number of instances in the $^{\mathrm{ith}}$ partition among the v splits partitioned by a feature f, and $(|\mathrm{D}|)$ which is the total number of instances in the dataset D.

$$\mathrm{SplitInfo_f}(\mathrm{D}) = \sum_{\mathrm{i}=1}^{\mathrm{v}} \frac{|\mathrm{d_i}|}{|\mathrm{D}|} \cdot \log\left(\frac{|\mathrm{d_i}|}{|\mathrm{D}|}\right)$$

The value of $\mathrm{SplitInfo}(\mathrm{f})$ decreases as the number of instances in each partition decreases with respect to the total number instances. However, the $\mathrm{SplitInfo}(\mathrm{f})$ value may decrease to approach the zero value, and this leads to an unstable ranking measure (divide by zero).

The C4.5 algorithm has other advantages over the ID3 algorithm. The C4.5 can deal with both continuous and discrete features. It handles missing values and applies tree pruning after the process of the tree induction.

CART and Gini Index

The gain ratio ranking technique provides a high rank to the features that split the dataset into unbalanced partitions, such that one of the partitions is larger than the other. The CART decision tree classifier uses a Gini index $\mathrm{Gini(D)}$ to measure the impurity in a dataset of instances. The rank of any feature f is measured by the amount of the reduction in impurity. This reduction occurs when this feature is used in binary, splitting the instances into two partitions. The feature that maximizes the impurity reduction is selected as an important feature for the classification problem. The rank $\Delta\mathrm{Gini(f)}$ of a feature f is calculated as the difference between the Gini index $\mathrm{Gini(D)}$ of the total dataset of instances and the Gini index $\mathrm{Gini_f(D)}$ of the datasets partitioned by this feature.

The impurity reduction is calculated as follows

$$\Delta\,\mathrm{Gini(f)} = \mathrm{Gini(D)} - \mathrm{Gini_f(D)}$$

The value $\mathrm{Gini(D)}$ that represents the impurity in a dataset D is calculated as follows

$$\mathrm{Gini(D)} = 1 - \sum_{\mathrm{k}=1}^{\mathrm{m}} \mathrm{P_k^2}$$

where $\mathrm{P_k}$ is the probability that an instance in a dataset D belongs to class k in the m classes of D.

The Gini impurity index $\mathrm{Gini_f(D)}$ considers only a binary split for a feature f in D. The selected splitting point is a value in the feature vector f that is best at splitting the instance in D into two partitions $\mathrm{d_1}$ and $\mathrm{d_2}$ only. The value of $\mathrm{Gini_f(D)}$ calculates the weighted sum of the impurity in each of the two sets $\mathrm{d_1}$ and $\mathrm{d_2}$ as follows

$$\mathrm{Gini_f(D)} = \frac{\mathrm{d_1}}{\mathrm{D}}\mathrm{Gini(d_1)} + \frac{\mathrm{d_2}}{\mathrm{D}}\mathrm{Gini(d_2)}$$

Example on Information Gain Calculation for ID3

Consider a database that represents a computer shop of 14 customers, where each customer has four characteristics including age, income, graduation, and credit rating (Han et al., 2012). The shop classifies each customer as buying or not buying a computer. The following CShop table represents a dataset of 14 instances, four features, and two class labels (Wasilewska, 2008).

Table 11: Dataset for the Computer Shop Example

ID	age	income	student	credit_rating	buys_computer
1	Young	high	no	Fair	no
2	Young	high	no	Excellent	no
3	Middle	high	no	Fair	yes
4	Old	medium	no	Fair	yes
5	Old	low	yes	Fair	yes
6	Old	low	yes	Excellent	no
7	Middle	low	yes	Excellent	yes
8	Young	medium	no	Fair	no
9	Young	low	yes	Fair	yes
10	Old	medium	yes	Fair	yes
11	Young	medium	yes	Excellent	yes
12	Middle	medium	no	Excellent	yes
13	Middle	high	yes	Fair	yes
14	Old	medium	no	Excellent	no

Source: Wasilewska, 2008.

The calculation of the information gain of feature age is as follows

- The dataset CShop is partitioned into two sets which are $\{\text{buys\_computer} = \text{yes};\ \ 9\ \ \text{instances}\}$ and $\{\text{buys\_computer} = \text{no};\ 5\ \text{instances}\}$. The entropy Info(CShop) in CShop is calculated as follows

$$\begin{aligned}\text{Info}(\text{CShop}) &= \text{Info}(\text{yes},\ \text{no}) = \text{Info}(9{,}5) \\ &= -\left(\frac{9}{14}\right)\log_2\left(\frac{9}{14}\right) + \ -\left(\frac{5}{14}\right)\log_2\left(\frac{5}{14}\right) = 0.94\end{aligned}$$

- The entropy of the values {Young, Middle, Old} of the feature age are calculated as follows

$$
\begin{aligned}
\text{Info}\big(\text{CShop}_{\text{Young}}\big) &= \text{Info}(\text{yes \& Young, no \& Young}) \\
&= \text{Info}(2,3) \\
&= -\big(\tfrac{2}{5}\big)\log_2\big(\tfrac{2}{5}\big) - \big(\tfrac{3}{5}\big)\log_2\big(\tfrac{3}{5}\big) = 0.971 \\
\text{Info}(\text{CShop}_{\text{Middle}}) &= \text{Info}(\text{yes \& Middle, no \& Middle}) \\
&= \text{Info}(4,0) \\
&= -\big(\tfrac{4}{4}\big)\log_2\big(\tfrac{4}{4}\big) - \big(\tfrac{0}{4}\big)\log_2\big(\tfrac{0}{4}\big) = 0 \\
\text{Info}(\text{CShop}_{\text{Old}}) &= \text{Info}(\text{yes \& Old, no \& Old}) \\
&= \text{Info}(3,2) \\
&= -\big(\tfrac{3}{5}\big)\log_2\big(\tfrac{3}{5}\big) - \big(\tfrac{2}{5}\big)\log_2\big(\tfrac{2}{5}\big) = 0.971
\end{aligned}
$$

- The entropy of feature age Infoage(CShop) is calculated as the weighted sum of the entropy of all possible values of this feature

$$
\begin{aligned}
\text{Infoage}(\text{CShop}) = &\big(\tfrac{5}{14}\big) \cdot \text{Info}\big(\text{CShop}_{\text{Young}}\big) + \big(\tfrac{4}{14}\big) \cdot \text{Info}(\text{CShop}_{\text{Middle}}) + \\
&\big(\tfrac{5}{14}\big) \cdot \text{Info}(\text{CShop}_{\text{Old}}) \\
= &\big(\tfrac{5}{14}\big) \cdot 0.971 + \big(\tfrac{4}{14}\big) \cdot 0 + \big(\tfrac{5}{14}\big) \cdot 0.971 \\
= &0.694
\end{aligned}
$$

- The information gained from partitioning the dataset according to feature age Gain(Age) is calculated as follows

$$
\begin{aligned}
&\text{Gain}\big(\text{Age}\big) = \text{Info}\big(\text{CShop}\big) - \text{Info}_{\text{age}}\big(\text{CShop}\big) = 0.940 - 0.694 \\
&= 0.246
\end{aligned}
$$

- The information gained for features age, student, income, and credit_rating are 0.246, 0.151, 0.029, and 0.048 respectively. The age is considered as the feature of the highest information gain and expected to act as the best splitting feature.
- The features in the dataset CShop are sorted according to their importance as follows: age, student, credit_rating, and income.

5.3 Decision Tree for Regression

Regression
This is a method of predicting the numerical value of a target variable given one or more independent variables.

A decision tree for **regression** can be constructed by an ID3 algorithm that uses the standard deviation reduction method instead of the information gain method. The standard deviation is a measure of the degree of variation in a set of numerical values. A feature vector of similar values is considered homogenous. The standard deviation of a completely homogenous feature vector is zero.

Consider the following dataset containing two feature vectors {Day, Temp} and one target variable {Target}. The task is to create a regression model by using a decision tree to predict the value of the target variable {Target} based on the feature vectors (predictors).

Table 1

Day	Temp.	Target
Working	Cold	23
Working	Hot	41
Weekend	Mid	32
Weekend	Cold	10
Weekend	Cold	5
Working	Cold	4
Working	Mid	7
Weekend	Hot	17
Working	Hot	19
Weekend	Mid	61
Weekend	Mid	36
Weekend	Hot	54
Weekend	Hot	20
Weekend	Cold	24

The class or target variable in a regression problem is a vector of numerical values, not categorical ones as in the previously considered classification problem. Consider that the target vector $\mathrm{c} = \{\mathrm{target}\}$ consists of $\mathrm{n} = 14$ values, then the average value $\overline{\mathrm{c}}$ of this vector is 25.21. The value of $\overline{\mathrm{c}}$ is calculated as the sum $(\sum \mathrm{c})$ of the values in vector c divided by the number of the values (n) in c. The standard deviation $\mathrm{S}(\mathrm{c})$ of the target vector c is calculated as follows

$$\text{Standard Deviation S of target feature } \mathrm{c} = \mathrm{S_c} = \sqrt{\frac{\sum (\mathrm{c} - \overline{\mathrm{c}})^2}{\mathrm{n}}} = 17.07$$

We now consider the standard deviation $\mathrm{S}(\mathrm{c},\mathrm{x})$ of the target vector $\mathrm{c} = \{\mathrm{Target}\}$ partitioned based on the feature vector $\mathrm{x} = \{\mathrm{Temp}\}$. The feature x is a categorical variable of three discrete values, while each value v is corresponding to a subset of values in c. Let the $\mathrm{P}(\mathrm{v})$ refer to the probability of v occurring in the target vector c. Its value is equal to the ratio of the number of instances whose value is v to the total number of instances. The standard deviation $\mathrm{S}(\mathrm{c},\mathrm{x})$ is calculated as follows

$$\begin{aligned}&\text{Standard Deviation S of c partitioned by x} = \mathrm{S}(\mathrm{c},\mathrm{x})\\ &= \sum_{\mathrm{v} \in \mathrm{x}} \mathrm{P}(\mathrm{v}) \cdot \mathrm{S_c}(\mathrm{v})\end{aligned}$$

Table 2

Feature Vector	v	$\overline{c}$	$S_c(v)$	Count	$P(v)$	$S_c(v) \cdot P(v)$
Temp.	Cold	13.2	8.65	5	$\frac{5}{14} = 0.357$	3.090886
	Hot	30.2	14.74	41	$\frac{5}{14} = 0.357$	5.263299
	Mid	34	19.14	32	$\frac{4}{14} = 0.285$	5.456094
$S(c,x) =$						13.81028

The reduction in the standard deviation is evaluated as the decrease in the standard deviation $S(c)$ of the target feature when the instances are split by a feature vector x , resulting in a standard deviation of $S(c,x)$. The value of the reduction in standard deviation is calculated as follows

$$\text{Reduction in Standard Deviation} = SDR(c,x) = S(c) - S(c,x) = 3.26$$

The reduction in the standard deviation SDR of the target vector is measured for each feature. The features are ranked based on the SDR value, such that the feature whose SDR value is the largest is considered the most important feature.

The coefficient of variation CV is then calculated, which is the "percentage ratio" of the standard deviation S to the mean value $\overline{x}$ of a feature vector x. CV is used to stop the branching in the decision tree induction process. The value of CV is calculated as follows

$$\text{Coefficient of variation} = CV = \frac{S}{\overline{x}} \cdot 100\% = 67\%$$

5.4 Decision Tree Construction

The decision tree in the three approaches (ID3, C4.5, and CART) is constructed in a greedy, top-down manner. The algorithm pertaining to these approaches uses a recursive function to iterate over the set features. First, the algorithm selects the highest ranked feature (greedy) from the feature set. A root-node is created to represent this feature, which is also removed from the list of features to consider in the next iteration. Branches are created down for this parent node (top-down), with each branch labeled by a distinct value (or range of values) for the corresponding feature value. If the instances in the input dataset that have this specific feature value are of different class labels, then the next highly

ranked feature is selected to be added as a child decision node from this branch. Otherwise, if all the instances of this feature value are of the same class C, then the child node from this branch is a leaf node that is labeled by this class C. A node that represents a feature is called a decision node while a node that represents a class label is represented as a leaf node. This process continues recursively until all terminal nodes of the tree are leaf nodes. The following section presents the steps of the tree induction in the ID3 algorithm, and an example of these steps.

The Decision Tree Creation

The algorithm for constructing the decision tree is a recursive function $\mathrm{BuildTree}()$. This function has two input parameters, the first parameter is the input dataset, and the second parameter is the set of attributes. When using the function $\mathrm{BuildTree}()$, it utilizes the original dataset D and the set of all features A. The output of this function is a decision tree that describes the classification rules of instances in the dataset D.

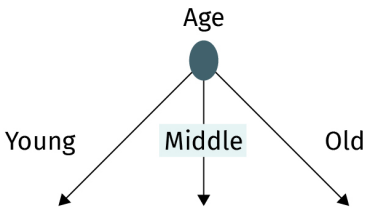
Function $\mathrm{BuildTree(D,A)}$:

- Create a node N.
- If $\mathrm{Info(D)} = 0$, where all the instances belong to the same class$\mathrm{C_l}$:
  - Returns the node N as a leaf node with class label $\mathrm{C_D}$.
- If $|\mathrm{A}| = 0$, where A is empty:
  - Returns the node N as a leaf node with class label $\mathrm{C_m}$ for the majority of instances in D.
- The rank of each feature f inthe input dataset D is (re-)calculated. The ID3 algorithm operates on features of discrete values only. A discretization process is applied to features of continuous values in the pre-processing step.
- Select the feature f of the highest rank from the set of features in A.
- Label node N as a decision node for feature f.
- For each value x of the values v in feature f:
  - Create a branch output from the node N, and label the branch by the condition $\mathrm{f} = \mathrm{x}$.
  - Construct a subset of instances $\mathrm{D_{f=x}}$, where the value of feature f of the instances in $\mathrm{D_{f=x}}$ is x.
  - If $\mathrm{D_{f=x}}$ is not empty:
    - Call recursively $\mathrm{BuildTree(D_{f=x}, A - f)}$. Remove feature f from the set of features A.
  - ELSE
    - Add leaf node and label this node with class label $\mathrm{C_m}$ for the majority of instances in D.

Example

Consider the dataset CShop in the previous section, the function $\mathrm{BuildTree(CShop,Age,student,credit\_rating,income)}$ selects a feature as the root node of three branches as shown in the following figure.

Figure 56: Building the Decision Tree in a Top-Down Manner (Part 1)



Source: Walid Hussein, 2020.

The function `BuildTree()` is recursively called three times for each branch.

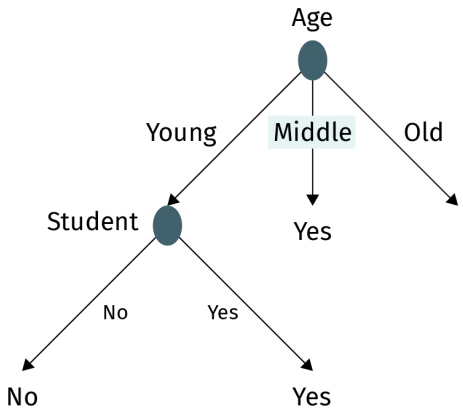
For the branch where the feature `Age` that has the value `Middle`, the function `BuildTree(`$\text{CShop}_{\texttt{Age=Middle}}$`, {student, income, credit_rating})` is recursively called. This function detects that all the instances in the dataset `[`$\text{CShop}_{\texttt{Age=Middle}}$`]` have the same class label `yes`. Accordingly, the function returns the label *yes* to be placed as the target of this branch.

For the branch where the feature `Age` has the value `Young`, the function `BuildTree(`$\text{CShop}_{\texttt{Age=young}}$`, {student, credit_rating, income})` is recursively called. The entropy `Info(D)` of the input dataset $\text{CShop}_{\texttt{Age=young}}$is calculated, and the ranking information `gain(f)` value of each feature `f` is re-calculated. The best-ranked splitting feature is the `student` attribute, it is thus selected, and a corresponding node is created. We have two output branches, one for each value `{yes, no}` in the `student` feature. Accordingly, the function `BuildTree()` is called recursively again as

- `BuildTree(`$\text{CShop}_{\texttt{Age=young \& student=yes}}$`, {credit_rating, income})`
- `BuildTree(`$\text{CShop}_{\texttt{Age=young \& student=no}}$`, {credit_rating, income})`

For both of these recursively called functions, the datasets $\text{CShop}_{\texttt{Age=young\&student=yes}}$and $\text{CShop}_{\texttt{Age=young\&student=no}}$contain instances of the same class label, yes and no, respectively. Both functions return leaf nodes.

Figure 57: Building the Decision Tree in a Top-Down Manner (Part 2)



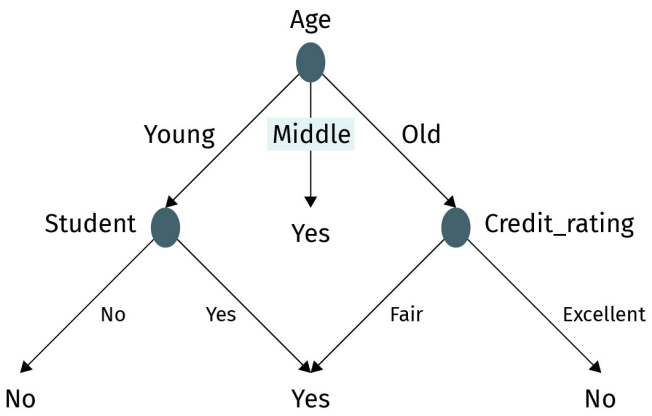
Source: Walid Hussein, 2020.

For the branch where the feature `Age` has the value `old`, the function `BuildTree(CShop`$_{\texttt{Age=old}}$`, {student, credit_rating, income})` is recursively called. The entropy `Info(D)` of the input dataset `CShop`$_{\texttt{Age=old}}$is calculated, and the ranking information `gain(f)` value of each feature `f` is re-calculated. The best-ranked splitting feature is the `credit_rating`, so the algorithm creates a corresponding node for this feature. There are two output branches, one for each value `{fair, excellent}` in the `credit_rating` feature. Accordingly, the function `BuildTree()` is called recursively again as

- `BuildTree(CShop`$_{\texttt{Age=young\&credit\_rating=fair}}$`, {income})`
- `BuildTree(CShop`$_{\texttt{Age=young\&credit\_rating=excellent}}$`, {income})`

For both of these recursively called functions, the datasets `CShop`$_{\texttt{Age=young\&credit_rating=excellent}}$and `CShop`$_{\texttt{Age=young\&credit_rating=fair}}$contain instances of the same class label, yes and no, respectively. Both functions return leaf nodes, as demonstrated below.

Figure 58: Building the Decision Tree in a Top-Down Manner (Part 3)



Source: Walid Hussein, 2020.

5.5 Decision Tree Pruning

Generally, the complexity of the constructed tree increases as the size and heterogeneity of the input dataset increases. This is particularly valid when the dataset contains large-scale, real-life data that may contain noise and outliers. A decision tree with a high number of decision rules would be very complex and hard to interpret. The complexity of the decision tree may even increase to a point where it causes an overfitting problem. This problem occurs when the rules incorporated by the decision tree fit the limited set of training instances perfectly, yet fail to reliably classify new instance exemplars. Clearly, this problem decreases the performance of the decision tree classifier. Tree pruning handles overfitting by decreasing the size of the tree to make it less complex. There are many pruning techniques; one of them works by removing subsections in the decision tree that have low classification power. This low degree of discriminating power is measured against a threshold, below which this section in the decision tree is considered unnecessary. This threshold should be low to avoid that the pruned tree becomes very small and too trivial to sufficiently classify the instances, which creates the problem of underfitting. This technique replaces each unnecessary section by a leaf node. The tree pruning technique is applied in two different ways: pre-pruning and post-pruning.

The pre-pruning process avoids building up these low-discriminating sections while the decision tree is construction. When a specific condition is met, the constructing algorithm avoids adding a new decision node to split the instances among the resulting branches, instead adding a leaf node with the label of the most frequent class among the pertaining instances. The concrete form of the pruning condition is dependent on the feature-ranking measure used in the decision tree induction. The information gain, gain ratio, and Gini

impurity index measure can all be used to gauge the impact of adding the decision node in question. If the used ranking measure is below a pre-specified threshold, then further partitioning is halted.

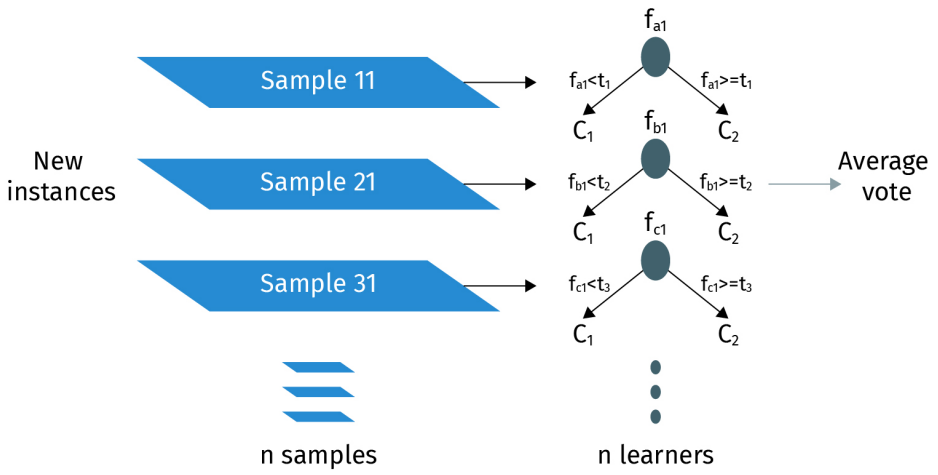
The post-pruning process removes spurious sub-trees from the fully constructed decision tree, and replaces them with a leaf node with the label of the most frequent class among this sub-tree. The CART approach implements the post-pruning process. It measures the cost complexity of the tree for each decision node in a bottom-up manner. The cost complexity is a function of the number of leaves and the rate of error (misclassified instances) in the tree. If the cost complexity of the tree increases by replacing a decision node with a leaf node, then the descendent sub-tree of this decision node is pruned.

5.6 Decision Trees in Ensemble Methods

An ensemble method is meta-method that combines a series of machine learning algorithms to improve the learning performance. The learning algorithms are used as base learners that are often weak learners. A weak learner is a poor learning algorithm that performs slightly better than a random classifier. The error rate of the results of this classifier is nearly 50 percent for binary classification problems. A decision tree can be used as a weak learner by controlling its maximum depth. For example, a simple decision tree of one level (decision stump) or a shallow decision tree can be used as a weak classifier. A decision stump finds a threshold for which a single feature can be partitioned into two parts around it. The main principle behind the ensemble method is that instead of training a single strong classifier on the whole input space, it is better to train many weak base-classifiers that are good at different parts of the input space. The aggregated base classifiers are either a single learning algorithm like decision trees (homogenous weak learners), or different learning algorithms (heterogeneous weak learners). Two prominent types of ensemble methods of homogenous weak learners are the bagging method and the boosting method, while the stacking method is an example of the heterogeneous weak learners.

The first type of ensemble method for homogenous weak learners is the bagging, or bootstrap aggregation ensemble method."Bootstrapping" refers to the method of generating bootstrap samples, which are the samples that are subsets of a specific number of instances x that is less than the total number of instances X. An initial subset of x instances is generated, and then n subsets are generated by random drawing and replacement of instances. Each of these n subsets is used to form a decision tree that is controlled to be a decision stump or a simple shallow-depth decision tree. Here, the bagging method is trained to build n weak learners. In the classification of new instances, each instance is tested by each of the n learners. The contribution of each learner is considered a vote to one of the class labels of the problem. A majority vote of these n learners is determined by the class label of this instance. This resulting vote is determined by calculating the average (mean) voted class labels. This aggregation step in the bagging method provides an equal weigh to each vote, yielding an unweighted average. This is because the homogenous weak learners are formed independently and in parallel. An example of this is illustrated below.

Figure 59: Bagging Method

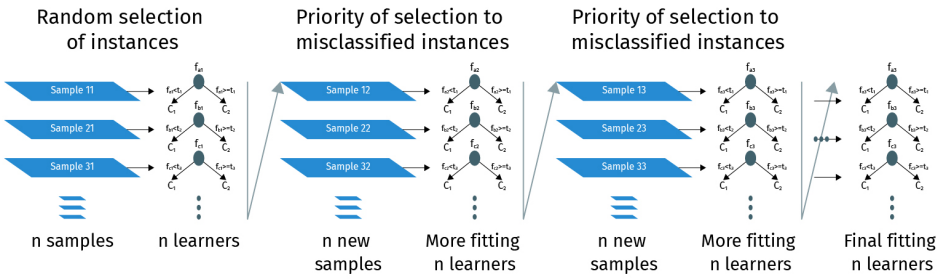


Source: Walid Hussein, 2020.

Random forest method is another example of a bagging method, with a change in the bootstrapping step. In addition to the random generation of the n samples, the instances in each sample have a random subset of m features rather than the original set of features M. Therefore, the random forest generates n samples of m features to benefit from two main advantages. The first advantage is that this enables the forest to efficiently handle a high number of features (dimensionality) in a problem. The second advantage is that it reduces the correlation between the predicted class label from highly different learners.

In the boosting method, the performance of the homogeneous weak learners is updated by the training samples, iteratively. The boosting technique is based on the bagging method, except that it applies a sequential fitting of the formed learners in an iterative manner. Each learner is fitted by giving more weight to instances that were incorrectly classified in the previous iteration. The first iteration in the boosting method performs just like the bagging method, where the probability of selecting any instance in the formed samples is equal for all instances in the dataset. In this iteration, each instance is provided with an initial weight of $\frac{1}{k}$, where k is the number of class labels in the dataset. Then, in the next iteration, the weight of misclassified instances is updated by increasing its value. The instance of higher weight has a higher chance or probability of being selected in the samples. The performance of the ensemble method should be increasing at each iteration where the learners of the model are fitting more instances correctly. This iterative model continues until a stopping condition, where the performance of the method is not rising any more. The stopping condition is dependent on the domain problem. For example, it is sufficient to have some maximum accuracy threshold. Boosting methods may risk overfitting because of the focus on misclassified instances that may be outliers or noise. An example of the boosting method is shown below.

Figure 60: Boosting Method



Source: Walid Hussein, 2020.

The adaptive boosting, or Adaboost, method is another example of the boosting methods. The Adaboost method assigns the higher weight to misclassified instances so that these instances receive the high probability for classification in the next iteration. In addition, it assigns the higher weight to the learners that shows a higher classification accuracy. Therefore, that vote of the learner of higher accuracy receives a higher weight in calculating the average vote. The final vote in the Adaboost method is a weighted average of the votes of the learners, unlike the boosting method. Like the previous ensemble models, this iterative model continues until a stopping condition is met.

The gradient boosting method is an Adaboost method that weighs based on a loss function. This loss function of each of the learners is the opposite of the gradient descent of the current fitting error. Another method is proposed based on the gradient boosting method, which is the Extreme Gradient Boosting: XGBoost. The loss function of this method computes the second order gradient to provide more information to classifier and reaches higher accuracy.

Implementation in Python

Here is a sample code for implementing the decision tree classifier using Python. The code below is applied to the dataset used in the previous example in this unit, (Navlani, 2018). The table is saved in a trialDT.csv and tested as follows

Code

```
import pandas as pd
trialdata = pd.read_csv("E:/trialDT.csv")
print(trialdata.shape)

#Data preprocessing involves
#(1) Dividing the data into attributes and labels
x = trialdata.drop('buys_computer', axis=1)
y = trialdata['buys_computer']

#(2) Dividing the data into training(x,y) and testing(x,y) sets.
```

```
from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(x, y,
test_size = 0.20)

# Processing of the classifier using Decision Tree classifier object
from sklearn.tree import DecisionTreeClassifier
from sklearn import metrics
clf = DecisionTreeClassifier()
clf = clf.fit(x_train,y_train)
#Predict the response for test dataset
y_pred = clf.predict(x_test)
# Model Accuracy, how often is the classifier correct?
print("Accuracy:",metrics.accuracy_score(y_test, y_pred))
```

SUMMARY

Decision trees represent a group of classification techniques that is based upon the construction of a tree like structure. Several approaches are proposed for constructing decision trees like ID3, C4.5, and CART. The decision tree classifier uses statistical-ranking techniques like information gain, gain ratio, or Gini impurity index. These techniques help in identifying the important features to use in building the decision tree. The constructed tree is typically very complex, especially when large datasets with a high probability of noise are used. A pruning process is applied to simplify the decision tree and to avoid the possible occurrence of overfitting. Decision trees are commonly used in ensemble classification models because they are considered weak learners.

UNIT 6

GENETIC ALGORITHMS

STUDY GOALS

On completion of this unit, you will have learned …

- the motivation for genetic algorithms.
- the definition of a genetic algorithm.
- the different phases of genetic algorithms.
- the application of a genetic algorithm to solve the knapsack problem.
- the implementation of genetic algorithms in Python.

6. GENETIC ALGORITHMS

Introduction

According to our best current scientific understanding, the earliest forms of life on this planet existed more than 3.5 billion years ago. Looking at the history of life and its development throughout the ages shows its remarkable potential to adapt to a multitude of environmental conditions, finding extraordinary solutions to existential challenges during this process. In this sense, genetic evolution by natural selection is the driving force behind variations across individuals of a **population**. Individuals who fit the environment better will survive and reproduce, thereby giving rise to the next generation. Since the environment changes constantly, evolution in nature is a continuous process with no defined end-state.

Population
For use in machine learning, a population is simply a group of objects in a particular space.

Abstractly, one can view the workings of biological evolution as an optimization technique, whose inventiveness and success are documented in the records of natural history. It is thus unsurprising that this particular take on optimization inspired by the Darwinian **theory of evolution** described above has been used as a model for technical optimization procedures. In this adaptation, however, the environment is considered static. Consequently, this process of evolutionary adaptation will continue until the fitness of individuals cannot be improved further, which stops the evolution process. Thus, one of the individuals in the last generation is considered optimal and no individual can be better. An evolutionary algorithm initially generates a set of solutions to a problem in a random way, and tests each of these solutions through a fitness function. The best possible solutions are evolved to the next generation in the evolution process. The process continues until a near optimal solution is obtained, and the output of the fitness function does not change.

Theory of evolution
A hallmark of biological change across time, Darwin's theory of evolution explains natural biological changes in and across species as adaptations to the species' habitat and environment.

Evolutionary algorithms constitute a generic metaheuristic model for optimization problems that operates on a population-based representation of the solution space (Chaiyaratana & Zalzala, 1997). A population in the evolutionary algorithm is a set of feasible solutions that may contain the best solution for a specific problem. The technique involves the transformation of a population from one generation to the next, where properties of individuals are represented by a code that can be modified to yield new exemplars. It uses and adapts the concept of biological DNA sequences **(genes)** in which a child shares many of the characteristics of the two DNA sequences of their parents. Individuals that show a high amount of fit to the optimization objective are allowed to breed the next generation.

Gene
A heredity unit that is transferred from a parent to their offspring, or gene, holds some characteristic of the offspring.

Being a heuristic technique, the approach cannot guarantee global optimality of the resulting solution, but the achieved solutions commonly prove to be sufficient for many practical purposes (Chaiyaratana & Zalzala, 1997).

Genetic algorithms (GA) are one subclass of evolutionary algorithms of the wider field of evolutionary computation and biologically inspired algorithms and heuristics. Other noteworthy approaches are

- Swarm algorithms simulate the movement of a group of animals (e.g., fish or birds) towards a specific target (Awange et al., 2018).
- Ant Colony algorithms simulate the communication between insects to find the shortest path between their nest and a food source. When traveling from their nest to any food source, an ant deposits a chemical material on the ground called a pheromone to trace its movement. The ant uses this trace to find its way back to the nest, while depositing more pheromones along the way. The other ants move in the direction where these chemical deposits are more frequent than in other directions. This indirect communication mechanism is called "stigmergy".

As outlined above, genetic algorithms are heuristic optimization methods. Since most, if not all, machine learning algorithms can be formulated as optimization problems, or at least contain an optimization task as part of their definition, possible applications of GA in machine learning are manifold. One typical use case is feature selection, where the set of all possible features to be used in a machine learning application is evolved in order to find the subset that provides the best overall model performance. Apart from that, GAs are commonly used in hyperparameter tuning. A hyperparameter is a variable that influences a model's behavior and output but is not adjusted by the training process. Examples of hyperparameters are the number of clusters in k-means, or limits on the depth of trees in decision tree learning. In particular, GA gained notable popularity in the optimization of structures and learning rules for artificial neural networks.

This unit goes into the details of one class of these algorithms, which is the genetic algorithm. As a practical example, it will be demonstrated how this algorithm can be used to solve one of the classic problems in optimization theory: the knapsack problem.

6.1 Genetic Algorithm Definition

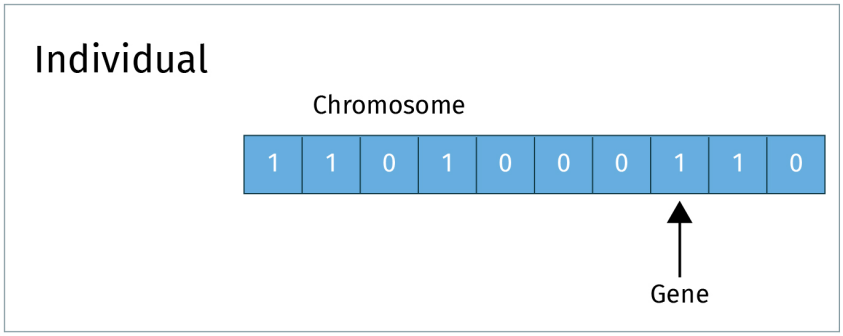
Genetic algorithms are a class of evolutionary algorithms that is inspired by the biological genetic evolution of living organisms (Sammut, 2011). The algorithm simulates the reproduction of individuals in a population in order to solve a specific optimization problem. This simulated evolution aims at iteratively exploring the space of possible solutions to find the optimum or near-optimum solution to the given problem.

An individual in a population is made-up of cells of different shapes and functionalities. Ultimately, these constituent parts are encoded by a set of entities known as genes or **DNA**. All the cells in an individual have the same DNA sequence that is different from that of other individuals. Genes are contained in chromosomes, which reside mainly in the cell nucleus. Every human cell contains 23 pairs of chromosomes, and each chromosome may contain hundreds to thousands of genes.

DNA
Deoxyribonucleic acid, or DNA, is a self-replicating material which is present in all living organisms, functioning as a carrier of genetic information.

The DNA, and consequently the chromosome, of an individual can be encoded technically as a list of bits, each bit representing a gene. Consider r an individual of with the following chromosome [1101000110], where the value (genotype) of the first gene in the chromosome is 1.

Figure 61: Chromosome Example: Gene Relation



Source: Walid Hussein, 2020.

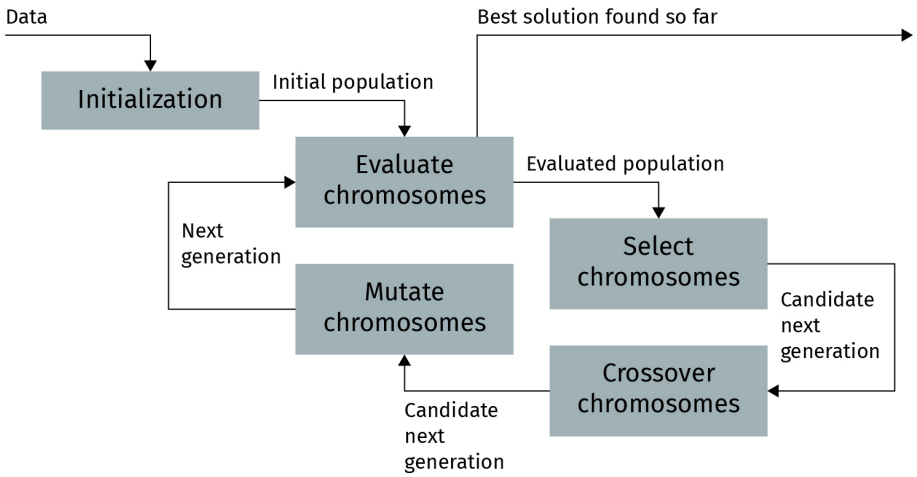
- This function varies according to the domain problem. The fittest individuals are selected for producing potentially even better offspring in the next generation. This process is based on Darwin's theory of natural selection which posits that the opportunity for survival is large for highly fitting individual, in accordance with the principle of the survival of the fittest.
- The biological evolution process describes the genetic changes in a population through successive generations. The genetic changes are the variations that occur in the reproduction process on the genetic level and are, thus, inherited by the next generations. These changes occur in two important ways in genetic algorithms: crossover and mutation of the genetic structure.

6.2 Genetic Algorithm Phases

The general outline of a genetic algorithm consists of five phases: (1) creation of initial population, (2) evaluation of the fitness function, (3) selection of the fittest, (4) crossover, and (5) mutation of the genetic structure (Sivanandam & Deepa, 2008). At the end of these five phases, a new population is produced based on the current population. The genetic algorithm goes through multiple iterations, where each iteration is formed by the above mentioned five phases. The iteration stops when an acceptable level of fitness is achieved by at least one individual, when no further improvement in fitness can be achieved, or when a pre-defined maximum number of iterations has been reached.

The iterating process in the genetic algorithm can thus be interpreted as a searching process that aims at finding successively better solutions.

Figure 62: The Five Phases in Genetic Algorithms



Source: Walid Hussein, 2020.

Initialization Phase

Initially, the individuals of the population are generated randomly in such a way that the corresponding chromosomes have a high coverage of the solution domain. The number of genes in individuals' chromosomes is dependent on the problem.

Evaluation Phase

The fitness of each individual is measured by a **fitness function**. The input to this function is the chromosome of the individual, and the resulting output of this function is the fitness score of the individual. The fitness function operates according to the objective function of the problem.

Fitness function
To measure how well a chromosome performs in an environment, a fitness function is used as an objective function that summarizes how well the designed solution is achieving a set of aims.

Selection Phase

The individuals with the highest fitness scores are selected for generating the new population. Individuals with a fitness score that is higher than a specific threshold are selected, where the threshold constitutes a parameter that is defined according to the problem. This phase ensures that the newly generated population has better fitness than the current population.

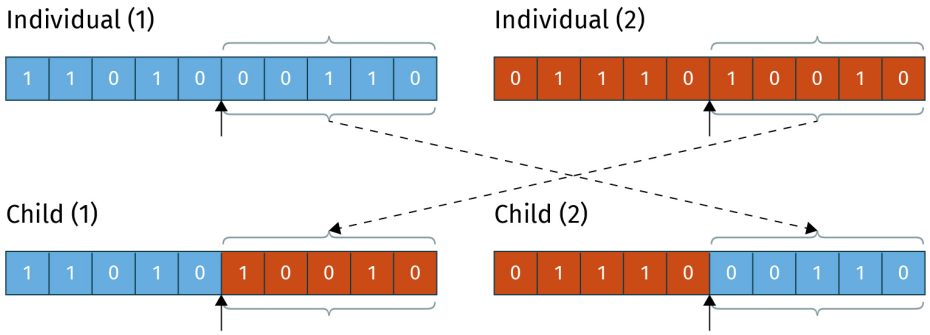
Crossover Phase

The creation of the new offspring takes place in this phase through mixing the genes from the parents' pair of chromosomes. In order for this crossover to take place, some random point(s), the crossover point(s), have to be determined in the chromosomes of the parents. Then, gene threads are cut and exchanged around this crossover point to create new threads. There exist two types of crossover methods: single- and multi-point.

Locus
A specific location, or locus, is a point in the chromosome.

- In single-point crossover, the procedure randomly chooses a **locus** and exchanges the subsequences before and after that locus between two chromosomes to create two offspring.

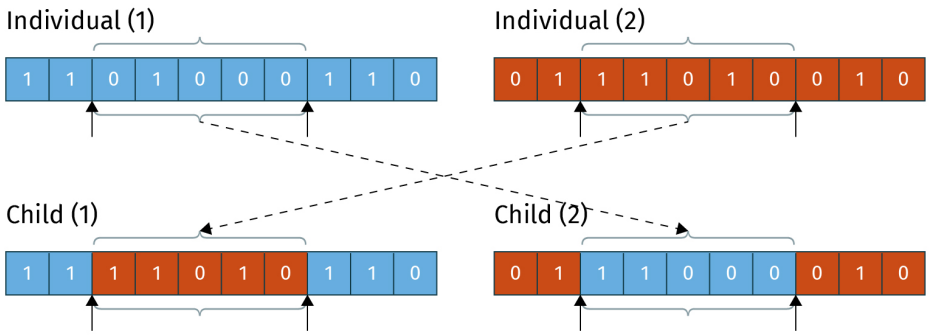
Figure 63: Single-Point Crossover



Source: Walid Hussein, 2020.

- Multi-point crossover selects a set of points at random. Chromosomes are cut at the crossover points. Corresponding sections are swapped, as demonstrated below.

Figure 64: Multi-Point Crossover



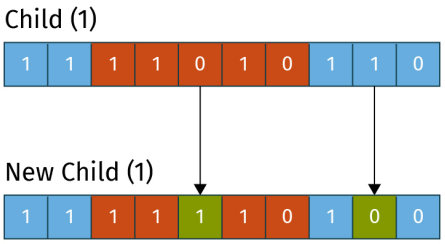
Source: Walid Hussein, 2020.

Mutation Phase

Mutation is the flipping of random genes at arbitrary locations in the individual chromosome. It prevents the population from falling into local optimum solutions too quickly. Mutation changes the new offspring by flipping its genes from 1 to 0 or from 0 to 1. Mutation can occur at each bit position in the string with some probability (Sivanandam & Deepa, 2008).

Mutation
Genes can undergo a change in their structure caused by the deletion, insertion, or rearrangement of the chemical code of which they are composed.

Figure 65: Mutation



Source: Walid Hussein, 2020.

Genetic Algorithm Structure

The overall outline of the basic genetic algorithm can be described in the following steps:

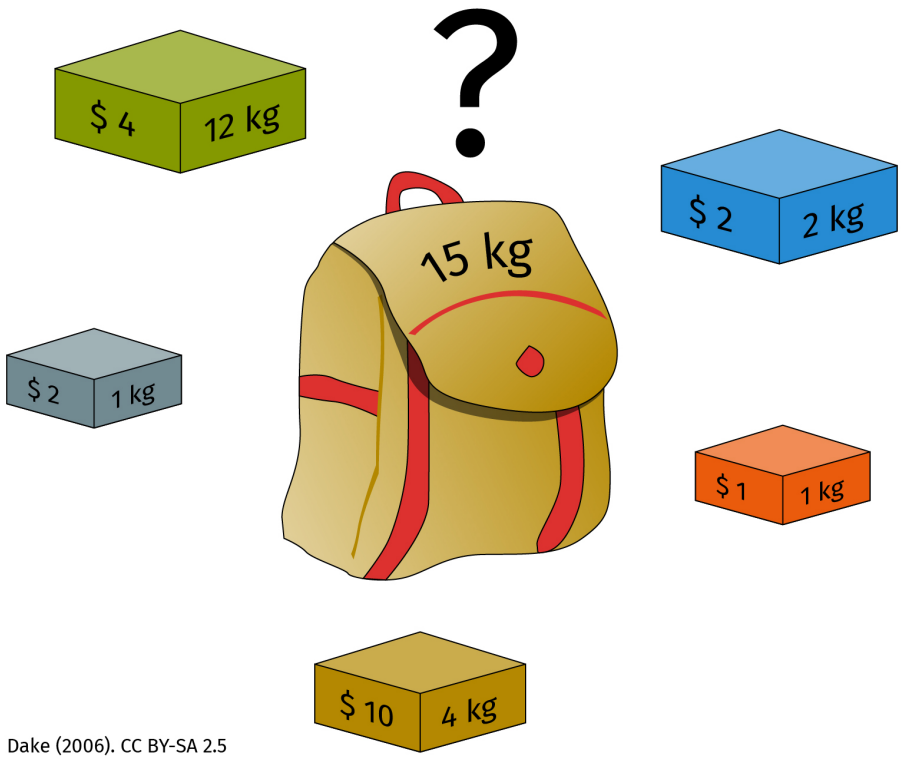
1. Initialization phase. The algorithm generates a random population of n chromosomes; this population represents the initial set of solutions. Each chromosome in the population represents a possible solution for the domain problem.
2. Repeat the following steps until convergence:
   - Fitness. Evaluate the fitness $\mathrm{f}(\mathrm{S})$ of each chromosome S in the current population.
   - Selection. Select the chromosomes having the highest fitness score from the population. Each pair of selected chromosomes are mated together to create the new offspring.
   - Crossover. Apply crossover between the parents to form a new offspring (child) that shares some of the characteristics of both parents.
   - Mutation. Mutate the new offspring at some positions in the chromosome.
   - Replace. Replace the current generation by the new generation.
3. Return the final population.

6.3 Genetic Algorithm Example: Knapsack Problem

This section discusses the knapsack problem which is a famous problem in combinatorial optimization. The problem can be stated as follows:

- Given a set of items, each item has a weight and a value.
- Given a knapsack of a maximum capacity, considered as the overall weight that the knapsack is limited to hold.
- The problem is to select a subset of items so that:
  - The total value is as large as possible, and,
  - The total weight is less than or equal to the limit, as described in the following figure.

Figure 66: Knapsack Problem



Dake (2006). CC BY-SA 2.5

Source: Dake, 2006.

Knapsack Problem Mapping

The knapsack problem can be formulated as follows:

- The problem is to select a subset of items from a set of n items. Each item $\mathrm{x_j}$ in n items has a profit $\mathrm{p_j}$ and weight $\mathrm{w_j}$. The requirement for the selected subset of items is that the total profit of these items is maximized, under the constraint that the total weight of these items does not exceed a specific capacity M.
- Each subset of items $\mathrm{x_j}$ represents a possible solution S for the problem. In other words, the target is to maximize the value of the fitness function $\mathrm{f(S)}$

$$f\big(S\big) = \sum_{j=1}^{n} p_j x_j (1)$$

subject to the constraint that the maximum allowed capacity is not exceeded.

Where

- The $f(S)$ function is a fitness function that evaluates the solution.
- The value x_j is a binary variable that indicates $x_j = 1$ if the item x_j has been selected to be stored in the knapsack, and $x_j = 0$ if it is not selected and remains out.

- The solution (chromosome) S is considered as a list of n bits, the location j of each bit represents the item x_j in the problem. For example, if items 2, 3, 5, and 7 are selected out of n items, then this solution is represented as follows:
  Items 2, 3, 5, and 7 → Chromosome $S = [0110101]$
- In evolutionary algorithms, a set of solutions is tested in each generation during the evolution. Each solution S is a possible solution that contains a set of items in the knapsack. The number of knapsack solutions in the evolutionary algorithm is m. The feasibility of each solution must be tested as follows:
  The knapsack constraint → $\sum_{j=1}^{n} w_j x_j \leq M$
- Infeasible solutions that break the constraint are allowed to exist during the evolution steps, but a penalty is imposed to the fitness of this solution. This penalty is proportional to the total amount of excess in the knapsacks as follows:

$$\text{Penalty} = Q \cdot t\big(M - \sum_{j=1}^{n} w_j x_j\big) (2)$$

  Where an integer number Q is considered such that it is a sufficiently large positive number, e.g., $Q = 6$.

  And the function $t(z)$ works as follows:

  - $t(z) = 0$, if $z >= 0$
  - $t(z) = -z$, if $z < 0$
- The fitness function is adjusted according to the penalty part as follows:

$$f(S) = \sum_{j=1}^{n} p_j x_j - \big(Q \cdot t\big(M - \sum_{j=1}^{n} w_j x_j\big)\big) (3)$$

Solving the Knapsack Problem

Consider a knapsack whose maximum carry weight is $M = 22$. Suppose the number of available items is $n = 7$, and each of the different benefits (value) and weights are as follows:

- Item: 1 2 3 4 5 6 7
- Benefit: 5 8 3 2 7 9 4
- Weight: 7 8 4 10 4 6 4

The task is to find the set of items of a total weight less than M having the best possible total benefit (optimal solution).

- Initialization: A number of solutions are randomly created; each solution is represented by a chromosome of $\mathrm{n} = 7$ genes. Each gene $\mathrm{x_i}$ at a location i is, either of a value $\mathrm{x_i} = 1$ if the item is selected to be added in the knapsack, or of a value $\mathrm{x_i} = 0$ if the item is not selected. The number of the randomly generated chromosomes in the population is set to $\mathrm{m} = 4$. The following list represents a population of four chromosomes (solutions) that is randomly generated:
  - Chromosome 1 : Items(2, 3, 5, and 7) → $\mathrm{S_1} = [0110101]$
  - Chromosome 2 : Items(1, 2, 3, and 4) → $\mathrm{S_2} = [1111000]$
  - Chromosome 3 : Items(2, 3, 4, and 5) → $\mathrm{S_3} = [0111100]$
  - Chromosome 4 : Items(3, 5, 6 and 7) → $\mathrm{S_4} = [0010111]$
- Repeat the following steps until the optimal solution is found:
  - Fitness: The fitness is calculated based on the fitness function:

$$\mathrm{f(S)} = \sum_{\mathrm{j}=1}^{\mathrm{N}} \mathrm{p_j x_j} - \left(\mathrm{Q} \cdot \mathrm{t}\left(\mathrm{M} - \sum_{\mathrm{j}=1}^{\mathrm{N}} \mathrm{w_j x_j}\right)\right)$$

  - This function is calculated for the solutions $\{\mathrm{S_1}, \mathrm{S_2}, \mathrm{S_3}, \mathrm{S_4}\}$ in the population as follows:

$$\mathrm{f(S_1)} = (8+3+7+4) - (6 \cdot \mathrm{t}((22) - (8+4+4+4))) = 22 - (6 \cdot \mathrm{t}(2)) = 22$$

$$\mathrm{f(S_2)} = (5+8+3+2) - (6 \cdot \mathrm{t}((22) - (7+8+4+10))) = 18 - (6 \cdot \mathrm{t}(-7)) = -24$$

$$\mathrm{f(S_3)} = (8+3+2+7) - (6 \cdot \mathrm{t}((22) - (8+4+10+4))) = 20 - (6 \cdot \mathrm{t}(-4)) = -4$$

$$\mathrm{f(S_4)} = (3+7+9+4) - (6 \cdot \mathrm{t}((22) - (4+4+6+4))) = 23 - (6 \cdot \mathrm{t}(4)) = 23$$

  - Selection: Based on these resulting fitness scores, the solutions $\mathrm{S_1}$ and $\mathrm{S_4}$ are of better scores than $\mathrm{S_2}$ and $\mathrm{S_3}$. The solutions $\mathrm{S_1}$ and $\mathrm{S_4}$ of the highest fitness are selected, such that the population in the next generation includes the solution set $\{\mathrm{S_1}, \mathrm{S_4}\}$.
  - Crossover: A crossover process takes place for each pair of solutions $[\mathrm{S_1 and S_4}]$ in the new population. A crossover process between the solutions $\mathrm{S_1}[0110101]$ and $\mathrm{S_4}[0010111]$ creates two new solutions $\mathrm{S_5}[0110111]$ and $\mathrm{S_6}[0010101]$. Here, the population in the next generation includes the solution set $\{\mathrm{S_1}, \mathrm{S_4}, \mathrm{S_5}, \mathrm{S_6}\}$.
  - Mutation: The mutation is applied randomly, for example, on a parent solution $\mathrm{S_1}$ and a child solution $\mathrm{S_5}$. The mutation in $\mathrm{S_1}$ is applied by flipping its first and second genes, and in $\mathrm{S_5}$ is applied by flipping its last gene. So $\mathrm{S_1}[0110101]$ and $\mathrm{S_5}[0110111]$ in the population becomes $\mathrm{S_1}[\underline{10}10101]$ and $\mathrm{S_5}[0110\underline{1}0]$.
  - Replace: The current generation is replaced by the population in the new generation. Here, the current population is as follows $\{\mathrm{S_1}[\underline{10}10101], \mathrm{S_4}[0010111], \mathrm{S_5}[0110\underline{1}0], \mathrm{S_6}[0010101]\}$.
- Repeat the following steps until the optimal solution is found:

- Fitness: The fitness is calculated for the solutions $\{S_1,S_4,S_5,S_6\}$ in the population as follows

$$f(S_1) = 19$$
$$f(S_4) = 23$$
$$f(S_5) = 27$$
$$f(S_6) = 14$$

- Selection: Based on these resulting fitness scores, the solutions S_4, from the initial (first) generation, and S_5, from current (second) generation, are selected in the population for the next (third) generation. Here, the population in the next generation is the solution set $\{S_4,S_5\}$.
- This process continues until the fitness values of solutions no longer increase.

6.4 Genetic Algorithm in Python

The following code shows how to implement the genetic algorithm for solving the knapsack problem in Python (Gad, 2018).

Main Steps

The main function in this program is the function that runs all the steps of the genetic algorithm. It calls two important functions in the algorithm: initialization and evolution. The steps of the main function are as follows:

- This main function starts by setting some values like the knapsack maximum capacity `KCap`, number of solutions in the population `POP_SIZE`, the number of generations `GEN_MAX`, and the number of items to be considered `NUM_ITEMS` in each processing the step.
- It creates a list of items of size (`NUM_ITEMS`); each item has a random weight and value from 0 to 20.
  - `(random.randint(0,20), random.randint(0,20))`: This piece of code creates two random integers, with each one having any value between 0 and 20. Examples of the generated pairs of random values are (12, 4), (7, 17), etc.
- Initialization. The main function starts the genetic algorithm by creating an initial population via a call to the `Initialization` function.
  - `population = Initialization(POP_SIZE, NUM_ITEMS)`
- Then the function processes a series of steps in a loop that is repeated `GEN_MAX` number of times.
  - Fitness. The fitness of each solution in the population is calculated, then the whole population is sorted according to the calculated fitness.
    - `fitness(ind, ITEMS, KCap)`: the fitness function measures the fitness of an individual according to the weights and values of the items in this individual.

- Selection, crossover, and mutation. The function then calls the `Evolution` function which includes the selection, crossover and mutation steps. The input parameter of the `Evolution` function is the sorted population.
- Replace. The `Evolution` function returns a new generation of the population. This new population replaces the old one.
  - `population = Evolution(population)`
- The total fitness of the population is counted and printed for each generation. According to the genetic algorithm steps, the total fitness should be increasing gradually.
  - `for i in population:`
    `totalFitness += fitness(i, ITEMS, KCap)`
  - The printed `totalFitness` is increasing gradually as follows: the total fitness of the initial population (i.e., 210), then the total fitness values of the next populations are (i.e., 559, 1673, 2135, 2411, 2476, 3085, 3120).

- Finally, this function ends up with a child population (The `GEN_MAX`'s (50th) generation) that includes the best solution to the problem that has been found in the last iteration. The population is sorted again in a descending order according to the total weight of each solution using the `fitness` function. The first chromosome in this population represents the optimal solution having the highest total value of that generation.

The function Python Code is as follows:

Code

```
def main():
   import random
   KCap = 50
   POP_SIZE = 30
   GEN_MAX = 50
   NUM_ITEMS = 15
   ITEMS = [(random.randint(0,20),random.randint(0,20))
            for x in range (0, NUM_ITEMS)]
   generation = 1 #Generation counter
   population = Initialization(POP_SIZE, NUM_ITEMS)
   for g in range(0,GEN_MAX):
       population = sorted(population,
                    key=lambda ind: fitness(ind, ITEMS, KCap),
                    reverse=True)
       totalFitness = 0
       for i in population:
           totalFitness += fitness(i, ITEMS, KCap)
       print(totalFitness)
       population = Evolution(population)
       generation += 1
   population = sorted(population,
                    key=lambda ind: fitness(ind, ITEMS, KCap),
                    reverse=True)
   print(population[0])
```

Initialization Phase

The function `Initialization` accepts two parameters `sizeOfPop`, and `NUM_ITEMS`. This function creates an initial population, where the number of chromosomes (solutions) in this population is `sizeOfPop`. Each solution represents a random list of `NUM_ITEMS` genes. The resulting population is returned as the output of the function. The function Python code is as follows:

Code

```
def Initialization(sizeOfPop, NUM_ITEMS):
   import random
   Population = []
   for x in range (0, sizeOfPop):
       individual = []
       for y in range (0, NUM_ITEMS):
           individual.append(random.randint(0,1))
       Population.append(individual)
   return Population
```

Fitness Phase

The function `fitness` accepts three parameters `individual`, `ITEMS`, and `KnapsackCapacity`. This function measures the fitness value of a single solution. For simplicity, this function uses equation (1).

This function sums up the total value `total_value` and the total weight `total_weight` of the selected items (genes) in the solution. The function returns the total value `total_value` if the total weight `total_weight` is less than or equal to the maximum capacity of the knapsack `KnapsackCapacity`.

Code

```
def fitness(individual, ITEMS, KnapsackCapacity):
   total_value = 0
   total_weight = 0
   index = 0
   for i in individual:
       if index >= len(ITEMS):
           break
       if (i == 1):
           total_value += ITEMS[index][0]
           total_weight += ITEMS[index][1]
       index += 1

   if total_weight > KnapsackCapacity:
       return 0
   else:
       return total_value
```

Evolution Phase

The following function `evolution` accepts two parameters `list_of_items` and `previous_generation` and returns the next generation. The steps of this function are as follows:

1. Selection: This step creates a subset population of the best solutions. Since the population is sorted in decreasing order; then the first 0.2 (`parent_percent`) of the total number solutions in the input population includes the best solutions.
2. CrossOver: This step does the crossover between random pairs of chromosomes in the best population subset.
   a) The while loop `[while len(children) < desired_length :]` iterates until the number of solutions in the new population is the same as in the old population. In other words, the summation of the number of selected parents and the number of the new child solutions must be equal to the total number of solutions in the old population.
   b) Then, each pair of parents is selected randomly using the following function call:
   `pop[random.randint(0,len(parents)-1)]`
   This line of code creates a random number between `0,len(parents)-1` and returns the solution of the corresponding randomly generated index.
   c) Then, it extracts half of the new child chromosome (solution) from each parent, which are connected to form the new child solution. Finally, the two halves are concatenated to form the new child solution.
   d) The result of this step is a subset of new child solutions of size `desired_length`
3. Mutation: This step includes the mutation of a random gene in some child solutions. It iterates over each solution and checks if a randomly created integer if it is less than a defined number `mutation_chance(0.08)` or not. If this event happens, a random gene in this solution is inverted. The code `[random.randint(0,len(child)-1)]` creates a random number below the total gene number in the chromosome to be mutated.
4. Finally, a composite set of the selected parents and the new child solutions is created and returned as the output of the function.

Code

```
def Evolution(pop):
    import random
    parent_percent = 0.2
    mutation_chance = 0.08
    parent_lottery = 0.05

    # Selection
    parent_length = int(parent_percent *len(pop))
    parents = pop[:parent_length]

    # CROSS-OVER
    children = []
    desired_length = len(pop) - len(parents)
    while len(children) < desired_length :
```

```
            male = pop[random.randint(0,len(parents)-1)]
            female = pop[random.randint(0,len(parents)-1)]
            half = int(len(male)/2)
            child = male[:half] + female[half:]
            children.append(child)

    # Mutation
    for child in children:
        if mutation_chance > random.random():
            r = random.randint(0,len(child)-1)
            if child[r] == 1:
                child[r] = 0
            else:
                child[r] = 1

    parents.extend(children)
    return parents
```

Program Output

The result of this function is of the form [0, 1, 0, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 1] .

This result indicates the first item is not selected while the second item is, then the third item is not selected while the fourth item is selected, and so on.

SUMMARY

This unit introduces evolutionary algorithms and presents some related approaches like Genetic algorithms, Swarm algorithms, and Ant Colony algorithms. Genetic algorithms are based on two important bio-inspired concepts which are the survival of the fittest and genetic evolutionary variation such as crossover and mutation. The main phases of genetic algorithms are initialization, evaluation, selection, crossover, mutation, and finally replacement.

The knapsack problem is introduced as a popular application examples of genetic algorithms. This problem can be solved by modeling the candidate solutions of this problem as chromosomes and by setting a suitable fitness function to evaluate the solution candidates. Finally, a complete program is presented to show how this algorithm can be coded in Python.

BACKMATTER

LIST OF REFERENCES

Awange, J. L., Paláncz, B., Lewis, R. H., & Völgyesi, L. (2018). Particle swarm optimization. In *Mathematical geosciences*, (pp. 167—184). Springer.

Bhattacharyya, S. (2018). Logit of logistic regression: Understanding the fundamentals. https://towardsdatascience.com/logit-of-logistic-regression-understanding-the-fundamentals-f384152a33d1

Breiman, L., Friedman, J. H., Olshen, R. A., & Stone, C. J. (1984). *Classification and regression trees*. Wadsworth & Brooks/Cole Advanced Books & Software.

Chaiyaratana, N., & Zalzala, A. M. S. (1997). *Recent developments in the evolution strategies of genetic algorithms: Theory and applications*. Research Report. ACSE Research Report 666. http://eprints.whiterose.ac.uk/81058/1/acse%20research%20report%20666.pdf

Chebli, A., Djebbar, A., & Marouani, H. (2018, November 25). *Semi-Supervised learning for medical application: A survey*. International Conference on Applied Smart Systems (ICASS 2018). https://ieeexplore-ieee-org.pxz.iubh.de:8443/stamp/stamp.jsp?tp=&arnumber=8651980

Croteau, N., Nathoo, F. S., Cao, J., & Budney, R. (2017). High-dimensional classification for brain decoding. In S. Ejaz Ahmed (Ed.), *Big and complex data analysis* (pp. 305—324). Springer.

Dake. (2006). Knapsack. https://de.wikipedia.org/wiki/Datei:Knapsack.svg

Das, J., Mukherjee, P., Majumder, S., & Gupta, P. (2014). Clustering-Based recommender system using principles of voting theory. *International Conference on Contemporary Computing and Informatics (IC3I 2014).*

Ernst, C. (2017). DBSCAN in Python. https://github.com/chriswernst/dbscan-python

Ester, M., Kriegel, H., Sander, J., & Xiaowei, X. (1996). A density-based algorithm for discovering clusters in large spatial databases with noise. *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining,* 226—231. https://www.aaai.org/Papers/KDD/1996/KDD96-037.pdf

Gad, A. (2018). Genetic algorithm implementation in Python. https://towardsdatascience.com/genetic-algorithm-implementation-in-python-5ab67bb124a6

Han, J., Kamber, M., & Pei, J. (2012). *Data mining: Concepts and techniques.* Morgan Kaufmann Publishers.

House sales in King County [Dataset]. (2016). https://www.kaggle.com/shivachandel/kc-house-data/download

Koenker, R., & Hallock, K. F. (2001). Quantile regression. *Journal of Economic Perspectives, 15*(4), 143—156. http://doi.org/10.1257/jep.15.4.143

Navlani, A. (2018). Decision tree classification in Python. https://www.datacamp.com/community/tutorials/decision-tree-classification-python

Pragathi, K., Jayanthi, T., & Malathi V. (2018). Cluster based segmentation using K-Means algorithm. *International Journal of Data Mining Techniques and Applications*, *7*(1), 64—67. http://ijdmta.com/abstract_temp.php?id=V7-I1-P13

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning, 1*(1), 81—106. https://doi.org/10.1023/A:1022643204877

Quinlan, J. R. (1993). *C4.5: Programs for machine learning*. Morgan Kaufmann Publishers.

Sammut, C. (2011). Genetic and evolutionary algorithms. In C. Sammut & G. I. Webb (Eds.), *Encyclopedia of machine learning*. Springer Science & Business Media.

Samuel, A. L. (1959). Some studies in machine learning using the game of checkers. *IBM Journal, 3*(3), 210—229.

Schölkopf, B., & Smola, A. J. (2002). *Learning with kernels: support vector machines, regularization, optimization, and beyond.* MIT Press.

Sivanandam, D., & Deepa, S. (2008). *Introduction to genetic algorithms*. Springer.

Wasilewska, A. (2008). Classification lecture notes. https://www3.cs.stonybrook.edu/~cse634/lecture_notes/07classification.pdf

LIST OF TABLES AND FIGURES

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

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

media@iu.org
www.iu.org

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