TRANSPORTA UN SAKARU INSTITŪTS

ILYA JACKSON

NEIROEVOLŪCIJAS PIEEJA KRĀJUMU VADĪBAS SISTĒMU METAMODELĒŠANAI UN OPTIMIZĀCIJAI

Promocijas darbs

zinātniskā doktora grāda iegūšanai
būvniecības un transporta inženierzinātnēs

Apakšnozare – telemātika un loģistika

Zinātniskais vadītajs:
Dr.habil.sc.ing., profesors
Jurijs Toluevs

RĪGA – 2020

UDK 519.87:004

Transporta un sakaru institūts

I. Jackson

Neiroevolūcijas pieeja krājumu vadības sistēmu metamodelēšanai un optimizācijai. Promocijas darbs. Rīga, Transporta un sakaru institūts, 2020. 141 lpp.

TRANSPORT AND TELECOMMUNICATION INSTITUTE

ILYA JACKSON

NEUROEVOLUTIONARY APPROACH TO METAMODELING AND OPTIMIZATION OF INVENTORY CONTROL SYSTEMS

DOCTORAL THESIS

to obtain the scientific degree Doctor of Science in Engineering

Scientific area “Civil Engineering and Transport”
Scientific subarea “Telematics and Logistics”

Scientific supervisor:
Dr.habil.sc.ing., profesor
Jurijs Toluevs

RIGA – 2020

ANOTĀCIJA

Promocijas darbu “Neiroevolūcijas pieeja krājumu vadības sistēmu metamodelēšanai un optimizācijai” ir uzrakstījis Ilya Jackson, Dr.habil.sc.ing., profesora Jurija Tolujeva vadībā.

Dotajā disertācijā tiek pētīta mākslīgā neironu tīkla un ģenētiskā algoritma kombinācijas lietderība un efektivitāte krājumu vadības sistēmu automatizētai metamodelēšanai. Darba aktualitāte ir saistīta ar metamodelēšanas automatizācijas perspektīvām risinot uzdevumus, kas rodas rūpniecībā, un neiroevolūcijas pieeju nesenajiem panākumiem neiroarhitektūru meklēšanas nozarē un to hiperparametru optimizācijā.

Lai izklāstītu krājumu vadības pieeju attīstību un uzsvērtu optimizācijas, kas balstīta uz metamodeļiem ievērojamās priekšrocības, tika veikts krājumu vadības modeļu plašs pārskats. Turklāt, tā kā neeksistēja vispārēji atzīta un saskaņota krājumu vadības problēmu uzdevumu kataloga testēšanai, uzmanība tika pievērsta vairākiem īpašās intereses paplašinājumiem, proti, pārdošanu zaudējumiem, ātrbojībai un nestacionāriem pieprasījumiem. Tieši šie paplašinājumi tika izvēlēti, jo tie krājumu vadības modeļiem rada lielu nenoteiktību un nelinearitāti un arī ir sarežģīti no skaitļošanas viedokļa. Turklāt, tie ir cieši saistīti ar aktuālajiem rūpniecības jautājumiem mazumtirdzniecības un taupīgās ražošanas jomā. Balstoties uz šiem paplašinājumiem, tika izstrādāti divi krājumu vadības sistēmu diskrētu notikumu imitācijas modeļi un tie tika izmantoti kā pamatne ierosinātā ietvara priekš automatizētās metamodelēšanas pārbaudei.

Piedāvāts ietvars ir balstīts uz ievērojamākām mūsdienīgajām praksēm. Darbā detalizēti aprakstīts kā ģenētiskais algoritms vada neironu arhitektūras un hiperparametru evolūciju ņemot vērā meklēšanas telpu, meklēšanas stratēģiju un precizitātes novērtēšanas metriku. Ietvars demonstrē nozīmīgu skaitļošanas spēju klasiskajā metamodelēšanā, kas formulēta kā regresijas uzdevums. Turklāt tiek apskatīta iespēja izmantot ierosināto ietvaru lai iegūtu optimālos kontroles parametrus uz reālās situācijas izpētes piemēra.

Darba galvenie rezultāti ir prezentēti 7 starptautiskās zinātniskās konferencēs un publicēti 12 zinātniskajos rakstos. Promocijas darbs sastāv no 8 nodaļām un tajā ir 141 lpp., 44 attēli, 18 tabulas pamattekstā, 9 pielikumi un 275 atsauces bibliogrāfijā.

ABSTRACT

The thesis “Neuroevolutionary approach to metamodeling and optimization of inventory control systems” is written by Ilya Jackson under the supervision of Dr.habil.sc.ing., professor Jurijs Toluevs.

Taking into consideration the urgent industrial need in metamodeling automation and recent success of neuroevolutionary approaches in neural architecture search and hyperparameter optimization, this thesis examines feasibility and efficiency of the combination of artificial neural network and genetic algorithm for automated metamodeling of inventory control systems.

In order to outline evolution of approaches to inventory control and highlight the remarkable advantages of metamodel-based optimization, an extensive review on inventory control models was conducted. Additionally, since there is no congenitally agreed-upon test problem catalogue for inventory control problems, attention is drawn to several extensions of special interest, namely lost-sales, perishability and nonstationary demand. These extensions are chosen, because they insert high degree of uncertainty and non-linearity to inventory control models and are challenging from the computational point of view. Besides, they are closely associated with up-to-date industrial concerns in the fields of retail and lean-production. Based on these extensions, two discrete-event simulation model of inventory control systems are developed and used as the baseline to test the proposed framework for automated metamodeling.

The proposed framework is built upon the most prominent state-of-the-art practices. The thesis describes in detail, how genetic algorithm orchestrates evolutionary morphism of neural architecture and hyperparameters with regard to search space, search strategy and performance estimation metrics. The framework demonstrates solid computational capabilities in classical metamodeling formulated as a regression problem. Additionally, the possibility of using the proposed framework to derive optimal control parameters is discussed with regard to a real-world case study.

Main results of this thesis are presented at 7 international scientific conferences and published in 12 scientific papers. The thesis consists of 8 chapters and includes 141 pages, 44 figures, 18 tables in the main body, 9 appendixes and 275 references in the bibliography.

CONTENTS

ACKNOWLEDGMENTS

It was an extremely exciting journey that finally took me here. Since a work like this requires lots of assistance, my deepest gratitude goes to all the people and institutions that contributed, in whatever manner, to my eventual success.

As general as possible and not only in the context of doctoral studies, I want to thank my parents who never get tired of giving me care and love. I also wish to thank my scientific supervisor Jurijs Tolujevs for giving me the freedom of action along with encouraging guidance. Besides, he examined this thesis in a very meticulous way finding typos and formatting issues along with methodological and logical inconsistencies. Additionally, my special thanks to people, who guided me through the path of knowledge and armed me with all necessary methodology to conduct this research. In alphabetical order: Aleksandrs Grakovskis, Boris Misnev, Eftychia Nathanail, Enrique Onieva, Irina Jackiva, Igors Kabaskins, Dmitry Pavlyuk, Mihails Savrasovs, Nadezda Spiridovska, Neil Rubens.

Besides, I want to express my gratitude to the Latvian state in general, and to the Latvian state educational agency (VIAA) in particular for providing me with a generous scholarship twice. Thanks to this privilege, I was released from financial constraints and was able to completely immerse myself in research and studies.

Finally, I wish to highlight that several sections of this theses are based on the research conducted within the framework of the ALLIANCE project, which has received funding from the Horizon 2020. In this regard, I want to thank both the board of the project and the European Commission.

LIST OF ABBREVIATIONS AND ACRONYMS

AD – Anderson-Darling test

AF – activation function

ANN – artificial neural network

AutoML – automated machine learning

BP – backpropagation

CH – Calinski-Harabasz index

CI – confidence interval

CM – confusion matrix

CV – coefficient of variation

CPU – central processing unit

DBSCAN – density-based spatial clustering of applications with noise

DES – discrete-event simulation

DTE – days to expiration

DI – Dunn validity index

ELU – exponential linear unit

EOQ – economic order quantity

GA – genetic algorithm

GDP – gross domestic product

GPU – graphics processing unit

GQ – parametric Goldfeld-Quandt test

HPO – hyperparameter optimization

IC – inventory control

LOF – local outlier factor

MLP – multilayer perceptron

MSE – mean squared error

NAS – neural architecture search

NEAT – neuroevolution of augmenting topologies

NFLT – “no free lunch” theorem

NNI – nearest neighbor imputation

SELU – scaled exponential linear unit

SGD – stochastic gradient descent

SKU – stock keeping unit

SO – simulation-based optimization

PCA – principal component analysis

ReLU – rectified linear unit

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

1.1. Motivation

Mankind deals with inventory control (IC), since the day it started to mine and stockpile resources of the planet. For the last 80 years, the field of IC has gained massive attention in academic, scientific and business worlds (Jalali and Nieuwenhuyse, 2015). This is not surprising, since inventory-related issues constitute significant expenses for various businesses. According to IHL Group's report, a tremendous share of capital, namely $1.1 trillion in cash or equivalent to approximately 1.5% of the nominal world GDP are tied up in inventory (IHL and Buzek, 2015). Moreover, companies are losing $634.1 Billion each year due to out-of-stocks and $471.9 Billion due to excessive stocks, which accounts for 4.1% and 3.2% of total annual revenue for an average retailer (Dynamic Action and IHL, 2015).

All companies are challenged to match supply and demand, and the way they tackle this challenge has a tremendous impact on the profitability. Due to the fact that markets are rapidly evolving and becoming more complex, flexible, and information-intensive, notorious binging-and-purging approaches are unsuitable. Such approaches, in which product is, firstly, overpurchased or overproduced in order to prepare for expected demand spikes and then discarded by sharp decline in price (Altiok, 2012). However, the full story is more intricate. On the first hand, even if the stored product does not lose its valuable properties over time, it freezes the working capital, which leads to opportunity losses, thus, it is natural for an enterprise to keep inventory level as low as possible releasing the frozen capital as fast as possible. On the other hand, businesses are facing unceasingly growing pressure on assortment variety and are forced to increase the range of products they operate with in order to meet so-called “long tail” demands (Gallino *et al*., 2016). Long tail demand corresponds to a contemporary phenomenon that causes retailers to make the bulk of the profits from a wide spectrum of products with relatively low individual sales frequencies.

It is also worth to point out that IC problems arise in various industries, and each single real-world inventory is replete with non-standard factors and subtleties. In this regard, it is extremely unlikely that the same set of assumptions and considerations will be equally applicable to all real systems. A fortiori, disruptive technologies such as advanced robotics, autonomous intelligent systems and additive manufacturing are revolutionizing traditional ways of creating and

distributing value, thus, making modeling of IC systems even more difficult and laborious (Lehmacher *et al*., 2017). Additionally, besides classical IC in logistics, manufacturing and supply chains, there is a huge class of IC problems that do not match the standard patterns and, thus, require a special approach. This class includes humanitarian aid, military logistics and healthcare (Bonney and Jaber, 2011).

1.2. Scope, relevance and novelty

As it recognized by such researchers as Duan and Liao (2013) and Tsai and Zheng (2013), practical problems in stochastic IC are usually analytically intractable, because of their complexity. Due to such restrictions of analytical models, approaches based on simulation-optimization are becoming more and more popular tools for solving complicated business-driven problems (Jalali and Nieuwenhuyse, 2015). As it is mentioned by Merkuryev (2012) this statement is especially true for the analysis and planning of complex logistics systems that operate in a stochastic environment. In addition, this fact was also emphasized by Mahdavi and Wolfe-Adam in their whitepaper, which concludes with the statement that analytic approaches often fall short of real-world problems, when they involve dynamics, nonlinearity, and stochasticity. Specifically, for these situations, simulation modeling is recommended for use in order to capture and define the behavior of a system (Mahdavi and Wolfe-Adam, 2019). Furthermore, from a computational point of view, the vast majority of these problems are NP-hard by nature, which postulates that the number of possible solutions is increasing exponentially with the problem size and there presumably exist no algorithms that can solve them precisely in tolerable time (Hanne and Dornberger, 2017).

Unfortunately, simulations of IC systems, especially detailed, are hungry for computational resources and require the sheer amount of time and computer memory. Besides that, simulation-optimization algorithms do not “learn”, more specifically, whenever the input parameters change slightly, a metaheuristic search (or another optimization method) must be executed once again. Taking into account a scale and dimensionality of real-world IC problems, such a never-ending search becomes an unacceptable luxury for business. In the light of these facts, it may be more reasonable to use an alternative cheaper-to-compute model, which is specifically designed in order to approximate an original simulation with a sufficient degree of accuracy (Merkuryeva, 2004). Such a “model of the model” is conventionally called a metamodel (Blanning, 1975).

Over the last decade artificial neural networks (ANNs) have demonstrated remarkable performance on a variety of problems. More specifically, ANNs outperformed traditional approaches in such tasks as image recognition, machine translation, speech recognition, and fitness approximation (Goodfellow *et al.*, 2016). Moreover, with recent revolution in deep-learning, metamodeling of industrial systems based on ANN has sparked the surge of interest in simulation community. Nevertheless, ANNs are usually developed manually by data scientists and artificial intelligence developers, which is quite fault prone and requires the sheer amount of time. In light of this fact, an interest in automated neural architecture search methods is growing drastically in various domains (Elsken, 2019). This temptation did not bypass simulation community causing the surge of interest to metamodeling with artificial neural networks (Lechevalier *et al.*, 2015). Notwithstanding the astonishing performance of artificial neural networks, there is a serpent in this metamodeling Eden. Although neural networks are universal function approximators, it is still crucially important to pay attention to the structure. How many hidden layers should it contain? How many neurons should be in each layer? Which optimization algorithm should be applied? Currently data scientists, artificial intelligence developers and other human experts try to answer these persistent questions by being deeply involved into development, fine-tuning and customization of artificial neural network-based metamodels.

As it is brilliantly concluded by Whitehead: “Civilization advances by extending the number of important operations which it can perform without thinking of them.” (Whitehead, 2017). Since the metamodeling with artificial neural networks involves human labor and, thus, consumes lots of time and financial resources, the metamodeling automation is extremely tempting. Taking into consideration recent success of neuroevolutionary approaches in neural architecture search (Real *et al.*, 2019) and hyperparameter optimization (Elsken *et al.*, 2019), its application to metamodeling of IC systems become promising. In this regard, the novelty of this research lies in the exploration, examining and adaptation of the state-of-the-art neuroevolutionary methods for metamodeling and optimization of IC systems. As the result of this research, the proposed neuroevolutionary framework is distinguished by versatility, flexibility, complete automatism, parallelism, robustness and high chance of finding nearly-optimal solutions in a finite time. The term “nearly-optimal” is common in simulation community and refers to the fact that the vast majority of simulation-based optimization problems is NP-hard by nature (Juan *et al.*, 2015). The complexity of these problems makes it impossible to check every feasible candidate-solution in

tolerable computational time. In this regard, there is no guarantee that truly optimal solution can be found during the search. On the other hand, an efficient algorithm (including metaheuristic), most of the time, can produce solutions sufficient in the practical context. Such solutions are considered to be nearly-optimal (Yang, 2010).

Besides, it is not superfluous to emphasize that, since the framework proposed in this work is simulation-driven. Thus, it does not, by definition, exclude the model developer, who is responsible for development of the original simulation model. In this regard, metamodeling based on artificial neural networks can be considered as a way of transferring expert knowledge about the IC problem to the system equipped with artificial intelligence. This human-centric approach ensures that human vision of the process is a major factor and primary consideration that complies with the latest guidelines of the European Commission Expert Group on Artificial Intelligence (Floridi, 2019).

1.3. Problem statement and research objectives

Recapitulating the previous section, the problem can be stated as follows: despite the remarkable performance of artificial neural networks, its application to metamodeling requires participation of a human expert, which leads to fault proneness, expensiveness and time-consumption.

Taking into consideration the urgent industrial need in metamodeling automation and recent success of neuroevolutionary approaches, the following research questions can be stated:

- How feasible and efficient is the combination of artificial neural network and genetic algorithm for automated metamodeling of inventory control systems?
- Is it possible to derive optimal control parameters for complex inventory control problems based on the obtained metamodel?

In search of answers to these questions, this research puts forward the following research objectives:

- to examine efficiency of neuroevolutionary approaches for automated metamodeling of inventory control systems in terms of implementability and accuracy;

- to explore the possibility of deriving optimal control parameters for stochastic multiproduct inventory control problems based on the obtained metamodels.

In order to complete these objectives, the following tasks arise:

- to review inventory optimization methods highlighting their drawbacks and limitations;
- to consider inventory control models paying attention to the features, extensions and enhancements of particular interest to modern business;
- to review state-of-the-art neuroevolutionary techniques and discuss the possibility of their adaptation for metamodeling of inventory control systems;
- to develop two inventory control models that incorporate the features of particular interest highlighted during extensive literature review and distinguished by complexity and realism;
- to develop the neuroevolutionary framework for metamodeling of inventory control systems adopting the most efficient and reliable methods and practices, and test its accuracy on the developed inventory control models;
- to compare simulation-based optimization with metamodel-based optimization in terms of accuracy and discuss the possibility of scaling up the proposed method to optimize real-world multi-product inventory control problems.

Based on that inventory control systems are postulated as the object of this research. The subject, on the other hand, is narrowed down to metamodel-based optimization of inventory control systems.

The following statements are put forward for defense:

- Metamodeling of inventory control systems can be automated by combining artificial neural networks with genetic algorithm into a neuroevolutionary framework. Such that genetic algorithm searches for a structure in a feasible region that results the artificial neural network-based metamodel with the highest accuracy-related metric.
- Artificial neural networks obtained by the proposed neuroevolutionary framework are capable to learn and generalize complex nonlinear relations between simulation variables

and, thus, can be efficiently applied for metamodeling of complex real-world inventory control systems.

- If the dataset generated by the simulation of inventory control systems is labeled in accordance with certain criteria, for example, the profitability of a candidate solution, metamodeling can be alternatively considered as a binary classification problem.
- Metamodels obtained using the proposed neuroevolutionary framework can be used to derive optimal control parameters without significant loss of accuracy.

1.4. Methodology and methods

This research adopts methodology from such fields as inventory control theory, discrete-event simulation, machine learning and evolutionary computing. Besides, the methodological foundation of this research widely incorporates methods of mathematical statistics for validating models, normality testing, heteroskedasticity testing, clustering and dimensionality reduction. Models and algorithms described in this thesis are implemented in Python 3.7 using several libraries (Table 1.1.).

Table 1.1. Python libraries used in the research

№	Library name and version	Purpose of use
1	NumPy 1.17	Random number generation, manipulating with multi-dimensional arrays and matrices (Van Der Walt *et al*., 2011).
2	SciPy 1.3.1	Scientific computing and statistical functions (Blanco-Silva, 2013).
3	TensorFlow 1.13.1	Artificial neural networks development (Abadi *et al*., 2016).
4	Keras 2.3.0	High-level API operating on top of TensorFlow (Chollet, 2018).
5	Scikit-learn 0.21.3	Data preprocessing, clustering, dimensionality reduction, pipelines development (Pedregosa *et al*., 2011).
6	DEAP 1.3.0	Evolutionary computation and genetic algorithms (Fortin *et al*., 2012).
7	Pandas 0.25.1	Data manipulation (McKinney, 2011).
8	Statsmodels 0.10.1	Statistical modeling (Seabold and Perktold, 2010).
9	Matplotlib 3.1.1	Plotting and data visualization (Hunter, 2007).
10	Seaborn 0.9.0	Plotting and data visualization (Bisong, 2019).

All the code is open-source and available in GitHub repository (Jackson, 2020). In addition, it is also included to the appendices.

1.5. Dissemination and approbation

The preliminary results, inferences and findings of the research were presented at the following conferences in Latvia, Germany and Sweden:

- International Conference "Reliability and Statistics in Transportation and Communication", Riga, Latvia, 2018, 2019.
- Winter Simulation Conference, Gothenburg, Sweden, 2018.
- International Doctoral Student Workshop on Logistics, Magdeburg, Germany, 2018, 2019.
- International Conference on Information Modelling and Knowledge Bases, Riga, Latvia, 2018.
- PhD Seminar "Sci-Bi: Digitalization in Logistics and Transport", Riga, Latvia, 2018.
- Research and Academic Conference "Research and Technology – Step into the Future", Riga, Latvia, 2017, 2018, 2019.

The results of the research were published in the following journals and conference proceedings (5 indexed by Scopus, 3 by Web of Science, 2 in publishing):

1. Jackson, I. and Tolujevs, J. (2018) The Discrete-Event Approach to Simulate Stochastic Multi-Product (Q, r) Inventory Control Systems. In: *proceedings of the 28th International Conference on Information Modelling and Knowledge Bases (EJC-2018)*, June 4-8, 2018, Riga, Latvia. pp. 94-101 [indexed by Scopus].
2. Jackson, I., Tolujevs, J. and Reggelin, T. (2018) The Combination of Discrete-Event Simulation and Genetic Algorithm for Solving the Stochastic Multi-Product Inventory Optimization Problem. *Transport and Telecommunication Journal*, 19(3), pp. 233-243 [indexed by Scopus, indexed by Web of Science].
3. Jackson, I. and Tolujevs, J. (2018) A combination of simulation and genetic algorithm for solving a stochastic inventory optimization problem. In: *proceedings of 11th International Doctoral Student Workshop on Logistics*, June 19, 2018, Magdeburg, Germany, pp. 31-35.

4. Tolujevs, J., Yatskiv, I., Jackson, I. and Reggelin, T. (2018) Dynamic Model of the Passenger Flow on Rail Baltica. In: *proceedings of 2018 Winter Simulation Conference (WSC)*, IEEE, Gothenburg, Sweden. pp. 3096-3107 [indexed by Scopus, indexed by Web of Science].

5. Jackson, I. and Tolujevs, J. (2018) Simulation-driven artificial intelligence for solving stochastic combinatorial optimization problems in production and logistics. In: *proceedings of PhD seminar Sci-Bi: Digitalization in Logistics and Transport*, Riga, Latvia, pp. 30-36

6. Jackson, I., Avdeikins, A. and Tolujevs, J. (2018) Unsupervised Learning-Based Stock Keeping Units Segmentation. In: *proceedings of International Conference on Reliability and Statistics in Transportation and Communication.* Springer, Cham, pp. 603-612 [indexed by Scopus].

7. Jackson, I., Tolujevs, J., Lang, S. and Kegenbekov, Z. (2019) Metamodeling of Inventory-Control Simulations Based on a Multilayer Perceptron. *Transport and Telecommunication Journal*, 20(3), pp. 251-259 [indexed by Scopus, indexed by Web of Science].

8. Jackson, I. and Tolujevs, J. (2019) Metamodeling of Inventory-Control Systems Based on Artificial Neural Networks. In: *proceedings of 12th International Doctoral Student Workshop on Logistics*, June 6, 2019, Magdeburg, Germany, pp. 82-86.

9. Jackson, I. (2019) Simulation-Optimization Approach to Stochastic Inventory Control with Perishability. *Information Technology & Management Science*, 22, pp. 9-14.

10. Jackson, I. (2019) Combining Simulation with Genetic Algorithm for Solving Stochastic Multi-Product Inventory Optimization Problem. *The Journal of Economic Research & Business Administration*, 130(4), pp. 96-102.

11. Jackson, I. (2019) Neuroevolutionary Approach to Metamodeling of Production-Inventory Systems with Lost-Sales and Markovian Demand. In: *proceedings of International Conference on Reliability and Statistics in Transportation and Communication.* Springer, Cham [in publishing].

12. Jackson, I. and Grakovski, A. (2019) Combining LSTM Artificial Recurrent Neural Networks and Fractal Analysis for Inventory Dynamics Prediction. In: *proceedings of*

International Conference on Reliability and Statistics in Transportation and Communication. Springer, Cham [in publishing].

Besides, the developments and findings made within this research were used in pedagogical practice at the Transport and Telecommunication Institute in form of case studies incorporated into such subjects as “Intelligent systems”, “Modeling of logistic and supply chain systems” and “Risk management”.

1.6. Structure of the thesis

The thesis consists of 8 chapters including the introduction and conclusions, as well as 9 appendices. It contains 141 pages, 44 figures, and 18 tables. The list of references contains 275 titles.

The chapter 1 introduces a potential reader to the motivation behind the research. In addition, the scope is specified, and relevance is highlighted along with novelty. The main research question is stated and tasks are formulated. Besides, the papers published within the framework of the research are listed.

The chapter 2 provides an extensive review on IC models and methods for deriving optimal control parameters. The chapter sheds light on the evolution of approaches to IC from analytic closed-form expressions to simulation-based optimization and metamodeling. The crux of this process is related to the fact that the world is changing rapidly and IC problems are becoming more complex by several orders of magnitude giving a rise to new extensions and subtleties. Additionally, attention is drawn to several extensions of special interest, namely lost-sales, perishability and nonstationary demand. These extensions are chosen, because, firstly, they insert high degree of uncertainty and non-linearity to IC models and, thus, are challenging from the computational point of view, and, secondly, they are closely associated with up-to-date industrial concerns and challenges, especially in the fields of retail and lean-production.

The chapter 3 outlines the methodological framework behind the research. Considering recent success of approaches that combine neural networks with genetic algorithms (neuroevolutionary approaches) in neural architecture search and hyperparameter optimization, this chapter discusses its application to automated metamodeling of IC systems. The most prominent state-of-the-art practices in the field of neuroevolution are overviewed, and specific attention is paid to the core components (artificial neural networks and genetic algorithms) and possible ways to combine

them. Since this research uses several methods of mathematical statistics for validating models, normality testing, heteroskedasticity testing and clustering, they are also described in a concise form.

The chapter 4 describes discrete-event simulation of two IC models using formal notation. Since there is no congenitally agreed-upon test problem catalogue for IC problems, the described models encompass extensions of special interest specified in the chapter 2. The models are used in the following chapters as the baseline to test the neuroevolutionary framework. The first model is developed to simulate stochastic multiproduct IC system with perishability, lost-sales and quantity-discounts. The first model corresponds to inventory and warehousing operations in the "Trialto Latvia" LTD. The second model is developed to simulate stochastic multiproduct production-inventory system with lost-sales and nonstationary demand. Such that the demand process is associated with an underlying Markov chain, and environmental parameters are represented by demand states of a Markov process. Both models are distinguished by nonlinearity and stochasticity, which make them especially challenging from a computational point of view.

The chapter 5 contains a detailed description of both the neural and evolutionary components of the neuroevolutionary framework. The framework is implemented based on the state-of-the-art methods and practices outlined in the chapter 3. The chapter also describes, how genetic algorithm orchestrates evolutionary morphism of neural architecture and hyperparameters with regard to search space, search strategy and performance estimation metrics.

The chapter 6 demonstrates the computational capabilities of the neuroevolutionary framework proposed in the chapter 5 with regard to the models defined in the chapter 4. The framework performs classical metamodeling formulated as a regression problem. Besides, residuals are analyzed to determine the quality of the derived metamodels. In addition, it is demonstrated that, if a dataset generated by a simulation of IC systems is labeled in accordance with certain criteria, for example, the profitability of a candidate solution, metamodeling can be alternatively considered as a binary classification problem

The chapter 7 discusses how metamodels built by the neuroevolutionary framework can be used to derive optimal control parameters. Simulation-based optimization and optimization based on metamodeling are compared in terms of accuracy. Besides, the possibility of scaling up the proposed method to optimize real-world multi-product IC problems is discussed based on a case study of "Trialto Latvia" LTD.

The conclusions summarize the conducted research and highlight the most significant findings. Results are discussed with regard to practical implementation. Besides, open research questions are listed and recommendations for further research are made.

2. LITERATURE REVIEW ON INVENTORY CONTROL THEORY

Mankind has been engaged in IC practices, at least in some primitive form, since ancient times, when it started to gather and stockpile resources of the planet. Moreover, IC has been a major point of discussion in industrial engineering and operations research for over 100 years (Bushuev *et al*., 2015). Various advanced numerical methods can be used for solving IC problems, which makes it a highly multidisciplinary filed attracting researchers from different academic disciplines. Despite the fact that the fundamental questions in any IC are all about determining the proper timing and size of replenishment, each particular problem may be notably different from another. Moreover, real-world problems are diverse and replete with specialties and subtleties. On the first hand, it emphasizes importance of IC models, and sparks the high degree of interest to study them. On the other hand, these facts make it a daunting task to subsume and summarize the gargantuan spectrum of literature related to IC theory in one treatise. In light of this fact, this section is structured as follows. Firstly, the timeline of IC with respect to methodology behind deriving optimal control parameters is reviewed. Such methodology includes analytical approaches, control-theory, dynamic programming, simulation-based optimization and metamodeling. After that, attention is drawn to several extensions of special interest, namely lost-sales, products' perishability and Markovian demand (Figure 2.1.). These extensions are chosen for two pivotal reasons. Firstly, they insert high degree of uncertainty and non-linearity to IC models and, thus, are challenging from the computational point of view. Secondly, they are closely associated with up-to-date industrial concerns and challenges, especially in the fields of retail and lean-production, thus, are interesting from business point of view.

Despite the fact that the pivotal purpose of IC is quite straightforward, namely to keep inventory level in such a way that leads to effective satisfaction of arising demand, the effectiveness of the IC system can be measured in many different ways. The most commonly used approaches in operations research literature focus either on cost minimization or profit maximization (Beyer *et al*., 2010). However, there are many factors that should be considered. Among them, the most prominent and important ones include cost structure, demand, replenishment lead time (replenishment lag), review time, excess demand, shelf-life and constrains.

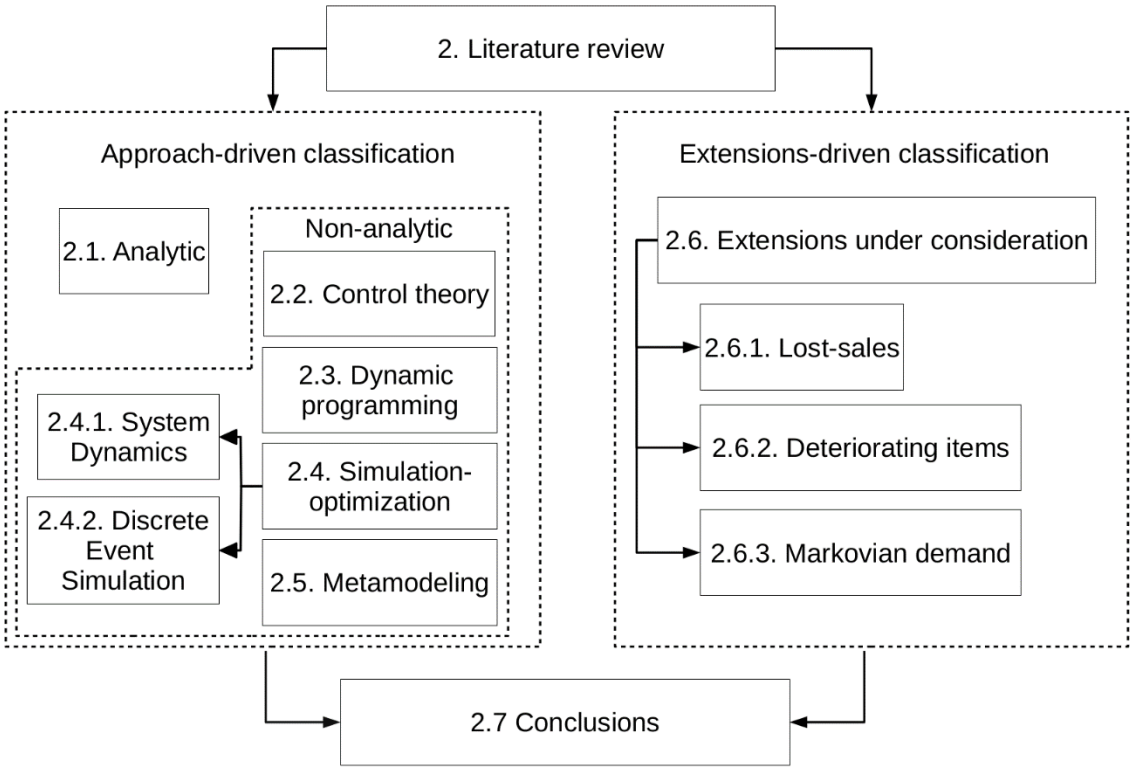

Figure 2.1. Outline of the chapter

Appropriate cost structure is the pivotal prerequisite for solving any IC problem. Cost structure, as a rule, consists of the following four core costs, which can be also extended and complemented depending on the purpose:

- Purchase cost – the cost of buying or producing products. It is important to note that in numerous models a quantity discount of some sort is introduced, if a number of purchased units is greater than a particular threshold;
- Setup cost – the cost associated with ordering and transportation of a batch of products. In production-inventory systems it can also include costs associated with setting up the machine;
- Holding cost – the cost associated with storing products in inventory over time. Besides the costs related to storage and handling, it can also include "frozen capital", which is a form of opportunity cost;
- Stock-out cost – the cost that reflects the financial consequences of unsatisfied demand (including loss of goodwill in advanced models).

Depending on the modeling purpose demand can be either constant or variable. Besides that, it can be deterministic or stochastic. Its stochasticity may be independent or depend on exogenous factors, such as the market conditions, decision maker's choice or simply the weather. Very much

like demand, replenishment lead time can be constant or random. Based on review time all IC models make up two groups. The first one includes models with continuous review, where the current inventory level is always known. The other one is periodic, where an IC system can get information on current inventory level at particular discrete points in time. Moreover, assumptions with regard to excess demand, excess supply and inventory deterioration affect specialty and complexity of an inventory model.

Among other things, for comfortable reading and navigation through this section, it is also crucially important to be familiar with the most common inventory policies that may be considered as an alphabet of IC. The models under consideration include such policies as:

- (*R, S*) policy assumes that the inventory level is reviewed every *R* period. Right after the review an order is placed bringing the inventory level to the predefined level *S* (this policy is also widely known as “base-stock");
- In (*R, s, S*) policy the inventory level is also reviewed every *R* periods and as soon as it passes a reorder point *s*, an order is placed bringing the inventory level to the predefined level *S*;
- (*r, Q*) policy is quite simple and straightforward, nevertheless, appears to be extremely efficient. According to this policy, the inventory level is reviewed continuously, and as soon as inventory level reaches the threshold *r*, an order of size *Q* is placed;
- (*s, S*) policy is absolutely the same as (*R, s, S*), besides the fact that the inventory level is reviewed continuously as in (*r, Q*) policy (Bartmann and Bach, 2012).

2.1. Analytic approach to inventory control

IC has occupied a steady niche among the most important topics in operations research. The history of mathematical theory of inventory and production may be traced back to Edgeworth (1888), who developed a variant of the news-vendor formula to model cash-flow in the bank. Nevertheless, the first formal model developed to aid managers in determining the proper size and timing of an inventory replenishment dates back to Harris (1913). The economic order quantity (EOQ) model is elementary, nevertheless, the most fundamental and celebrated:

$$EOQ^* = \sqrt{2\frac{KD}{h}} \tag{2.1}$$

Because of its algebraic representation, it is well-known as "square-root-formula". The classical EOQ model makes five strict assumptions:

- The demand rate *D* (units produced per unit time) is constant;
- Backlogs (shortages) are not allowed;
- The replenishment lead time equals to zero;
- Each replenished batch is subject to a setup cost *K*;
- The unit holding cost *h* is charged per unit of time.

Despite the simplicity, the equation elegantly describes the crucial relation in IC, namely the trade-off between replenishment and holding costs. This fact explains, why even nowadays several researchers still use EOQ model as a core complementing it with various extensions and superstructures (Lukinskiy and Lukinskiy, 2017). Important variations of this model include lost-sales, quantity discounts, shelf-life and non-zero replenishment lag. A detailed and comprehensive discussion on these extensions can be found in the study conducted by Hadley and Whitin (1963).

EOQ-based models were followed by the study of models that incorporate both uncertainty and dynamics (Arrow *et al.*, 1951). Models in which the demand flow is a random variable with a known probability distribution. Additionally, it is worth to note so-called dynamic lot-size model that in fact is a generalization of EOQ, which takes into account variability of demand for the product over time. The model was introduced by Wagner and Whitin (1958), who also proposed an algorithm for deriving the optimal control policy. These seminal papers were followed by the fundamental treatises in inventory theory (Arrow *et al.*, 1958; Scarf, 1960). These works gave a rise to (*R, s, S*) and (*s, S*) policies, most likely, the most celebrated policies in modeling of IC systems. The (*R, s, S*) policy has been extensively studied, and its optimality under assumptions of independent and stationary demand and deterministic replenishment lead time was proved by Karlin (1960) for single-product IC systems. For stochastic single-product IC models, on the other hand, it was demonstrated that the (*s, S*) policy is optimal under different conditions. For instance, the optimality is proved for a case of fixed ordering and setup costs by Veinott (1966). Ten years later, optimality for the case of lost-sales was proved by Shreve (1976). However, as it was emphasized by Zipkin (2008), this proof does not extend to the case when the replenishment lead lag is non-zero.

The last-mentioned fact explicitly exposes the major drawback of analytic models, namely they put forward a set of assumptions and considerations that frequently do not correspond to real-world IC problems and do not guarantee the optimal solution, if these assumptions are violated. Taking into account that IC problems arise in various industries, and each single business-driven problem is replete with non-standard factors and subtleties, it would be rash to believe that the same set of assumptions will be equally applicable to all IC systems. Besides such inflexibility, it is also recognized by such researchers as Duan and Liao (2013) and Tsai and Zheng (2013) that real-world IC problems are analytically intractable, due to their complexity and stochasticity. Such limitations of analytic approaches engendered new branches in studies of IC. Although, even nowadays classical analytic models are still developing and play an important role in inventory theory, a huge and diverse arsenal of non-analytic methods are applied, especially, in complicated business-driven problems.

2.2. Control theoretical approach

Currently optimal control theory is acknowledged as the completely developed branch of applied mathematics that uses differential equations to analyze, how physical systems behave in time (Ortega and Lin, 2004). Generally, differential equations are a classical way to model dynamics in the physical world. In this regard, its application to modeling of IC systems was considered a valid alternative. In the context of optimal control theory, a basic inventory dynamic can be expressed by the following differential equation:

$$\frac{dI}{dt} = Poduction(t - \tau) - Demand(t), \tag{2.2}$$

where the dynamics of the inventory *I(t)* is the result of the difference between the inbound flow *Production(t - τ)* and the demand rate *Demand(t)*. Such that τ is a replenishment lag (Axsater, 1985).

The utilization of classical optimal control methodology in IC can be traced back to Simon (1952), who managed to optimize inbound flow in an elementary single-product system. Sethi and Thompson (1981) attempted to apply stochastic differential equations to an industrial IC model. The considered model operated with a single product under stochastic and normally distributed demand. One year later Vickson (1982) also demonstrated a notable application of stochastic

optimal control. He managed to derive policies that leaded to average cost minimization for a periodic IC model with stochastic replenishment lead time.

A straightforward application of optimal control theory for inventory systems was shown by Wikner (1994), who considered an industrial inventory-control model on three fundamental levels, namely forecasting, lead times, and inventory replenishment strategies. Laplace-transformations and Z-transformations were calculated for several common forecasting techniques including rolling averages and exponential smoothing. Khmelnitsky and Gerchak (2002) paid specific attention on the fact that in some cases demand occasionally depends on the amount of available stock. This statement is especially true in cases of novelty or impulse purchases. The authors proposed optimal control theory-driven inventory model that incorporates inventory-level dependence of demand rate and lost-sales. The paper provides the explicit solution for special case of time-invariant demand and incapacitated case is solved numerically and analyzed in detail.

The seminal study conducted by Benjaafar *et al*. (2010) considers the IC system that deals with "impatient customers". Researches managed to show that in such settings, optimal policy can be achieved using the modified (R, s, S) policy. In a numerical study, the authors compared performance of the optimal policy against several other policies. More generalized model with probabilistic replenishment lags, allowed partial backordering and discounts is developed by Taleizadeh *et al*. (2017). This IC model is analyzed scrupulously and a closed-form solution is obtained for the optimum order size. Besides that, two special cases with the replenishment lags distributed uniformly and exponentially are studied using sensitivity analysis.

Summing up, the approaches based on control theory are, as a rule, proposed to a single-product IC system. As it was demonstrated such applications are both valid and efficient, when products are homogeneous enough to be treated as a flow.

2.3. Dynamic programming approach

Unfortunately, due to the "curse of dimensionality", at the scale of real-world business-driven problems both analytic and control theoretical models cannot be solved explicitly in reasonable computation time. In this regard, one must eventually resort to some sort of approximation (Sarimveis *et al*., 2008). On the other hand, as it was concluded by Domschke *et al*. (2015), IC problems can be naturally paraphrased as dynamic programs.

Dynamic programming belongs to the class of tree-like decision models. The pivotal philosophy behind dynamic programming suggest linear separation of a problem into several sub-problems determining the solution functions recursively for each of them. Such that the combination of the partial problems' optima corresponds to the optimum of the original problem (Bellman, 1957).

Among numerous applications of dynamic programming, Clark and Scarf (1960) were the first to proof that the optimal feedback rule for a multi-product IC system is the (*R, s, S*) policy for the cases with static demand. The considered IC problem can be characterized as discrete-time and finite-horizon. The optimal control parameters for the (*R, s, S*) policy were derived by splitting the original problem into a collection of single-echelon IC sub-problems with amended cost functions. Jammernegg (1981) meticulously studied a stochastic IC model with imperfect information on demand. Dynamic programming was applied to obtain optimal IC policies for two cases. Firstly, for a case of stochastic independent correlated demand. Secondly, for partially known and correlated demand. Dynamic programming was applied by Simchi-Levi and Zhao (2003) in order to analyze economic benefits from information sharing between a retailer and a producer in a multi-echelon supply chain. The researchers examined several scenarios with a different degree of mutual information sharing, and derive the optimal control policy for each scenario using dynamic programming. Based on computational experiments, the paper concludes with the statement that information sharing along supply chains can be extremely profitable, especially when the production facility has excessive production capacity.

Besides the mentioned applications, approximate dynamic programming is also very popular in IC. For instance, a dynamic programming framework for deriving approximate solutions for finite-horizon resource allocation and IC problems was developed by Powell (2006). However, it is worth to mention that this computational framework incorporated various methods beyond approximate dynamic programming, namely mathematical programming, stochastic approximation and simulation. Approximate dynamic programming also found its place in practical case-study (Simao and Powell, 2009). Solving the spare parts IC problem in aircraft industry, the paper presented a novel model and solution approach to determining the nearly-optimal inventory levels in large-scale distributed warehouse network. The proposed algorithm has been successfully validated and implemented. As it mentioned in recent study (Cimen and Kirkbride, 2017), as IC systems become more complex, approaches tailored for optimizing toy-

like single-product systems are incapable of deriving optimal policies. For instance, optimal policies cannot be readily derived for IC systems involving the interrelated product inventories, multidimensionality and non-decomposability. In light of this fact, Cimen and Kirkbride proposed the application of approximate dynamic programming augmented with sample backup simulation approach to overcome such a computational challenge. The paper demonstrates numerical results that signify the undeniable advantage of policies developed with the proposed algorithm in comparison to policies obtained using deterministic approximation.

Summarizing this section, dynamic programming is both popular and efficient procedure to obtain a nearly-optimal policy for stochastic IC problems. However, dealing with large-scale multi-product multi-echelon IC systems, the computational burden becomes immense, which compels to incorporate approximation techniques along with simulation into the algorithmic framework.

2.4. Simulation-based optimization

Computer simulation or just simulation is the digital reproduction of a physical asset that relies on a computer to calculate the outcomes of a mathematical model in numerically straightforward manner. Since simulations allow one to verify the reliability of mathematical models in very flexible way, they have proven to be a useful tool for modeling of various physical systems including inventory (Dubois, 2018). Moreover, recent advances in simulation modeling and increased availability of computational capacity encouraged engineering, scientific and business communities to take advantage on simulations for modeling real-world problems for which the objective function cannot be expressed analytically (Pasupathy and Ghosh, 2013). As the result of increased popularity of simulations, natural interest in manipulating with input parameters in order to maximize or minimize the value of the objective function has sparked. Simulations have gained particular popularity in applications related to production, logistics and supply chain management (Merkuryev *et al*., 2004).

Simulation-Optimization (SO), also known as simulation-based optimization is an umbrella term for techniques that treat computer simulation as a “black-box” looking for specific settings of the input parameters that lead to the optimal output (Amaran *et al*., 2016). The real-world IC is commonly characterized by the necessity for nearly-optimal solutions in feasible computing times. That is why SO techniques are used so widely to define an optimal inventory policy. Additionally,

such an approach provides a modeler with a tool to deal with real-world stochasticity in unconstrained way and assess alternative candidate-solutions by risk analysis.

In general terms, in SO problems an algorithm searches for such values of the decision variables that lead to the optimal output of the following objective function (Jalali and Nieuwenhuyse, 2015):

$$\max_{x \in \Phi} J(x) = E[Y(x)], \tag{2.3}$$

where x stands for the vector of input parameters, and Φ is the set of feasible solutions. $Y(x)$ is a stochastic output of the simulation. So, the value of the objective function $J(x)$ is estimated based on the average of η simulation runs (Koulouriotis *et al*., 2010).

$$J_\eta(x) = \frac{1}{\eta} \sum_{i=1}^{\eta} Y(x) \tag{2.4}$$

It is worth to emphasize that the number of runs (also known as replications) η used in the estimation of objective function is a core determinant of both the accuracy of estimation and computational cost for SO techniques, which make it a subject of classical trade-off (Banks *et al*., 2000).

Summarizing, SO aims to utilize a simulation instead of an objective function in traditional form applying an optimization algorithm to find such simulation adjustments that would lead to the optimal output (Figure 2.2.).

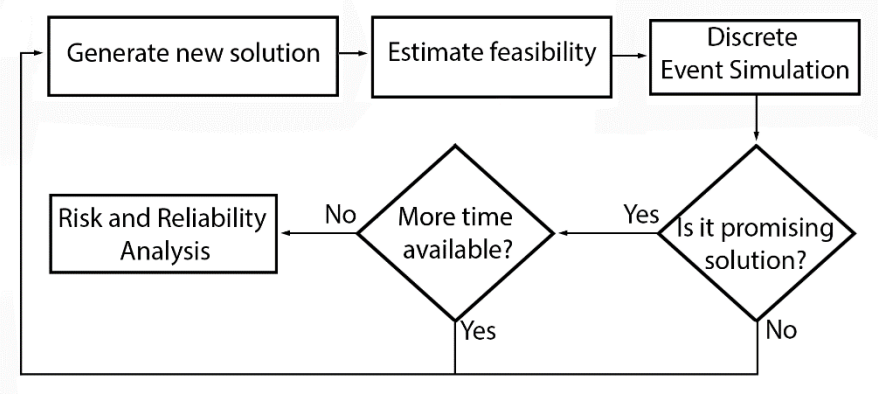

Figure 2.2. The logic behind SO in discrete-event case

In the demonstrated framework, the iterative searching process has to assess the quality of feasible solutions, highlighting the promising ones. The process continues until the search time

runs out. Immediately after this, a decision maker selects a final solution among promising with regard to a preferable risk policy. According to Pidd (1998), the simulation provides a natural way to introduce randomness of stochastic process. Furthermore, real-world stochasticity may be modeled throughout the best-fit probability distribution. The distribution may be either theoretical or empirical, without the need to be approximated to normal or exponential.

There is a plethora of SO methods, however, all of them may be divided by two major groups, namely metaheuristics and gradient-based. When one deals with extremely large or even infinite set of feasible solutions, it is recommended to apply metaheuristics (Hong and Nelson, 2009). Notable examples of metaheuristics include genetic and evolutionary algorithms, evolutionary strategies, particle swarm optimization, tabu search and simulated annealing (Olafsson, 2006). Although metaheuristics is mainly applied to combinatorial optimization problems, they also could be successfully used in a continuous case (Andradottir and Prudius, 2010). For cases of continuous decision variables and differentiable objective function, such gradient-based methods as sample path optimization and stochastic approximation are widely used. It is worth to mention that such techniques require the gradient estimation, which may be done using likelihood ratio, finite differences, simultaneous perturbations, or perturbation analysis (Fu and Hill, 1997).

2.4.1. System Dynamics

In the late 1950s a novel methodology aimed to study dynamic flows in industrial systems was introduced (Forrester, 1961). This methodology was initially named as “Industrial Dynamics”, however, the scope was greatly extended and the approach became eventually known as “System Dynamics” (Forrester, 1973). System Dynamics initially aimed to increasing the adoption of feed-back theory among managers and management scientists. At the beginning the method has been seriously criticized (Ansoff and Slevin 1968). However, nowadays System Dynamics is a widely used method to understand the dynamics of complex systems including IC. The pivotal philosophy of the method is based on the holistic approach to the modeling, namely System Dynamics recognizes that the structure of any complex system is as important in determining its behavior as the set of individual components themselves.

Despite the fact that System Dynamics is usually used exclusively as a simulation tool., the literature has several instances in which System Dynamics combined with optimization techniques was applied to identify optimal policies and parameters for IC systems (Schenk *et al*., 2010). For

example, the paper by Wolstenholme (1982) presented the use of System Dynamics in optimization of multi-echelon supply chains in the copper industry. Besides IC, such aspects as mining and production were also taken into account. Additionally, it is worth to mention the case-study described by Powell and Bradford (2000), who managed to apply qualitative systems dynamics to derive the optimal resource-management policy (including IC parameters) for an international defense company.

2.4.2. Discrete-Event Simulation

Nowadays discrete-event simulation (DES) is the most prominent simulation paradigm for SO frameworks (Gosavi, 2015). The DES paradigm is specialized in modeling systems at a low and medium level of abstraction and focuses on systems in which a sequence of operations is performed (Mahdavi and Wolfe-Adam, 2019). Following the introduction of linear congruential random-number generators (Lehmer, 1951), the origin of DES dates back to General Simulation Program, the first general-purpose simulator for industrial plant modeling (Tocher and Owen, 1960).

Nevertheless, the first simulation-based optimization of IC system was described only in the middle of 20th century by Fu (1994), who demonstrated the SO application to an IC system under (*s, S*) policy. The model assumed zero replenishment lag and periodic review. The cost function comprised holding, purchasing, transportation and backlogging. Fu managed to cover both the discrete parameter and the continuous parameter cases. In discrete parameter case, the author applied such methods as ranking-and-selection procedures and multiple-comparison. In the continuous parameter case such gradient-based methods as perturbation analysis were used. Applying derivative-free SO method Kapuscinski and Tayur (1998) derived the nearly-optimal control policy and analyzed its properties for a single-echelon IC model that operates with a single product under stochastic demand. Authors considered several cases including the finite-horizon and the discounted infinite-horizon. Even this early paper already concluded with an extremely significant statement that obtaining the exact analytical solutions is difficult for stochastic large-scale IC problems.

In the seminal paper Bollapragada and Rao (2006) examined a single-product, non-stationary IC problem with both supply and demand represented as random variables. Besides that, the model incorporated capacity limits on replenishment and service level requirements. A DES model for

finite-horizon problem is developed in order to determine nearly-optimal replenishment orders over the horizon. The authors proposed an unusual heuristic based on the first two moments of the random variables and a normal approximation. Numerical experiments confirmed that the proposed heuristic was distinguished by high performance of approximately 99.75% of real optima. The applied SO worked even when the output parameters were not normally distributed. Additionally, the paper contained sensitivity analyses of such parameters as shortage penalty costs, capacity limits, and demand variance. In the same year, the whole DES-based SO framework was proposed (Aras *et al*., 2006). The framework was originally developed to compare two alternative strategies in realistic settings. The first strategy used manufacturing as the primary mean of satisfying customer demand. On the other hand, the second one was focused on remanufacturing. SO equipped with non-monotonic search heuristic was tested on numerical experiments with different initial parameters.

The solemn paper by Ding *et al*. (2009) addressed the design of production-distribution networks. Besides IC, such networks include supply chain configuration, order splitting and transport allocation. During the research a flexible framework for simulation was developed in order to enable the automatic simulation with different IC strategies. The framework incorporated a multi-objective genetic algorithm for optimization of operation strategies and associated control parameters in production-distribution network. It is also worth to note that the proposed SO framework was applied in real-world automotive industry. The similar metaheuristics was utilized for optimization of stochastic multi-product IC system under the (*R, s, S*) policy (Arreola-Risa *et al*., 2011). Additionally, the proposed method incorporated regression analysis. The approach was developed and tested for an oil and gas company, which eventually decided to adopt it. In order to test the performance in general settings, 900 simulation experiment with different initial parameters were conducted.

Heuristics-driven SO approach was also used for determination of the appropriate safety-stock level for IC in clinical trial supply chain (Chen *et al*., 2012). Besides IC, demand forecasting was a subject of interest. The heuristics behind the framework took advantage on mixed integer linear programming for solving the resource-constrained scheduling problem). Online retail is also a popular scope of SO application. For example, Becerril-Arreola *et al*. (2013) considered a problem in online retail industry associated with two-step decision. In described settings, retailer firstly makes decisions on his profit margin and free-shipping threshold. After the optimal inventory level

for a planning horizon should be determined. In the presented case-study publicly available statistics is used to find the best-fitting distribution for consumers' order sizes and conversion rate. The simulation software "Arena" was used to simulate the IC system and in-built optimizer "OptQuest" was applied to derive the optimal control parameters that lead to maximum profit. Moreover, a sensitivity analysis was conducted and important managerial insights were obtained. Namely, the impact of the unit holding and unit shipping costs on the retailer's optimal decisions was discovered.

It is also worth to emphasize that some researches attempted to incorporate linear programming into SO framework. For instance, Zeballos *et al*. (2013) in their unusual research embedded simplex method in order to find the optimal working capital target and order size for a single-product single-echelon finite horizon IC system. This study especially focused on financial aspects of IC incorporating into simulation such features as working capital constraints, transaction delays and several separated sources of financing.

Peirleitner, *et al*. (2016) considered a stochastic supply chain management problem. In this paper each node along the multi-echelon multi-product supply chain manages stock according to the (*r; Q*) policy. The problem was represented as bi-objective optimization problem. Such that overall supply chain costs were subject to minimization, while service level was subject to maximization. Such optimal control parameters as reorder points and lot sizes were found using SO approach, which combined an evolutionary algorithm with DES simulation. In the same year discrete-rate simulation was used as a core to solve a single-product (*r, Q*) IC problem (Zvirgzdina and Tolujew, 2016). In this study the model was developed in ExtendSim using inbuilt genetic algorithm to derive optimal control parameters.

The recent research (Zahedi-Hosseini, 2018) is focused on the joint SO of operative maintenance and spare part IC for an industrial facility with various configurations. Firstly, spare part provision for a single-line conveyor-like system was considered. The simulation results indicate that a (*R, s, S*) was cost-optimal. Secondly, a parallel multi-line production system was modeled. It is found out that a policy inspired by just-in-time (*r, Q*) resulted the lowest costs. In both cases the long-run average cost per unit time was taken as an objective function to minimize. Several control policies were compared and optimal parameters were derived using inbuilt SimRunner's optimizer (ProModel, 2010).

2.5. Metamodel-based optimization

Unfortunately, simulation, especially detailed, is both time and memory consuming. Besides that, SO algorithms do not “learn”, more specifically, whenever the input parameters change slightly, a metaheuristic search (or another optimization method) must be executed once again. Taking into account a scale and dimensionality of real-world IC problems, such a never-ending search becomes an unacceptable luxury for business. In the light of these facts, it is usually more reasonable to use an alternative cheaper-to-compute model, which is specifically designed in order to approximate an original simulation with a sufficient degree of accuracy (Merkuryeva, 2004). Such a “model of the model” is conventionally called a metamodel (Blanning, 1975).

Metamodeling is a fairly old and well-known approach in simulation community (Law and Kelton, 2000) that also has not bypassed logistics and production (Tolujew *et al*., 1998). The grist of the idea is to find a model that approximates the “black-box” with accuracy sufficient in the context of the task (Figure 2.3). As the result, a computationally efficient metamodel captures relations between decision variables and simulation output (Kleijnen, 2008). After the metamodel is developed, it is possible to employ techniques specifically designed for deterministic optimization. Among various metamodeling techniques the most sought-after are response surface methodology (Angun *et al*., 2009), kriging models, radial basis functions (Biles *et al*., 2007) and artificial neural networks (Nezhad and Mahlooji, 2014).

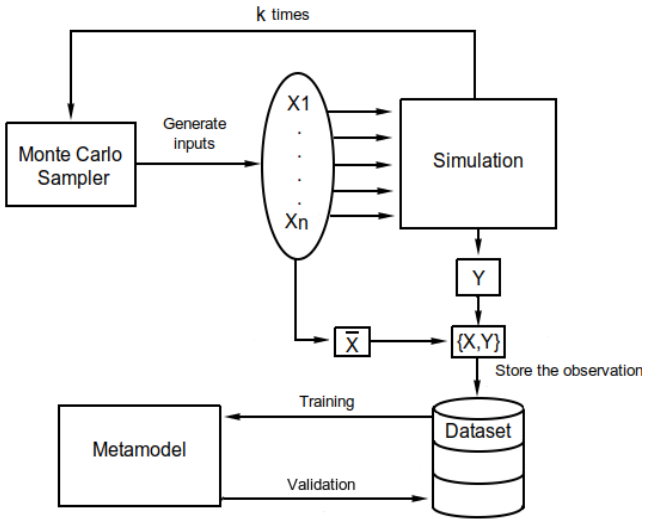

Figure 2.3. The logic behind metamodeling

Among applications related to inventory-control, it is, firstly, worth to mention the research conducted by Bendoly (2004). This research considered a simulation of substitute-inventory availability scenarios in transshipment IC system. The proposed methodology combined response surface methodology with non-linear optimization. The findings demonstrated that reductions in coordination costs and associated inventory availability decisions could provide limited benefits. Biles *et al*. (2007) explored the potential of the kriging methodology for constrained simulation optimization. The proposed technique is applied to an IC system under (*s, S*) policy. The experimental result demonstrated that kriging was a robust approach to solve constrained optimization problems in stochastic simulation of IC systems. Additionally, it is worth to mention the applied research (Lin and Tsai, 2009) that proposed an algorithm for defining an optimal inventory level for wafer fabrication processes. The research adopted a simulation model based on a real factory in order to generate data and demonstrates an optimization algorithm combining a multilayer feedforward neural network with quadratic programming. Combining second-order polynomial metamodel with response surface methodology the paper written by Angun (2011) demonstrated the solution to risk-averse SO problems. The novel robust response surface methodology was applied to IC problem with constrains.

Abruptly, recent pervasive and game-changing findings in deep-learning did not bypass metamodeling sparking the surge of interest to artificial neural networks (ANN) becoming a "hot-topic" in simulation community (Lechevalier *et al*., 2015). For example, Prestwich *et al*. (2012) managed to combine a single-layered ANN with an evolutionary algorithm to derive an optimal policy for a simulation-based stochastic multi-echelon inventory-control system. In the proposed neuroevolutionary approach ANN was trained by a simulation-driven evolutionary algorithm. Numerical experiments proved that this method is capable to derive high-quality policies in feasible computational time using networks of a simple architecture. In the same year the seminal research (Can and Heavey, 2012) presented another evidence of capability to combine genetic programming and ANNs for metamodeling of complex IC systems. This paper provided a comparative analysis of genetic programming and ANNs for metamodeling of DES models. Three stochastic industrial systems are empirically studied, namely an automated material handling system in semiconductor manufacturing, an (*s, S*) IC model and a serial production line. The results showed that genetic programming provides greater accuracy in validation tests, demonstrating a

better generalization capability than ANN with one hidden layer. On the other hand, genetic programming required more computational budget comparing to metamodel-based approach.

Nezhad and Mahlooji (2014) meticulously studied ANN-based metamodels for stochastic multi-dimensional SO problems with constraints. It is important to emphasize that the authors used a realistic (*s, S*) IC model to study the robustness and efficiency of the developed algorithm and compared the results with those of the OptQuest optimization package. This numerical study indicated that the newfound metamodel-based algorithm was competitive in terms of accuracy, but required fewer simulation replications. It is also worth mentioning that Jad and Owayjan (2017) presented the simulation of a whole enterprise resource planning system that utilized 30-layered ANN as an inventory-policy controller.

2.6. Extensions under consideration

In addition to various ways to derive optimal IC policy, extensions intended to relax restrictive assumptions or to incorporate special subtleties have also drawn attention of leading researchers. For instance, great efforts have been made to develop more general and realistic cost functions. Besides, extensions for IC models include incorporating features of real-world IC systems, such as capacity constraints, supply constraints, exogenous environment, lost-sales, product's perishability and so on. Such a diversity of IC problems can be illustrated by introducing the following classification scheme (Silver, 1981):

- Single-product and multi-product;
- Deterministic and stochastic;
- Single-period and multi-period;
- Stationary and non-stationary;
- Backorders and lost sales;
- Finite horizon and infinite horizon;
- Perishable and non-perishable.

Digressing from this multitude, attention is drawn to several extensions of special interest, namely lost-sales, products' perishability and Markovian demand. These extensions are chosen for

the following reasons. Modern markets are extremely competitive and industries are facing unceasingly growing pressure on both prices and quality. Moreover, companies are required to swiftly respond to stochastic market conditions, especially in the field of e-commerce and retail. It is also worth mentioning that improper IC policies lead not only to corporate losses, but also to overproduction, which is extremely harmful for humanity as a whole. In this regard sustainability concerns become supportive in advancing non-classical IC models to meet the challenges of the 21st century. Moreover, the mentioned extensions insert high degree of uncertainty and non-linearity to IC models and, thus, are quite challenging from the computational point of view.

2.6.1. Inventory control with lost-sales

Lost-sales or out-of-stock can occur along the entire supply chain, nevertheless, the most visible cases take place in the fast-moving and perishable goods industry (Gruen and Corsten, 2007). There are various culprits to blame for an out-of-stock, for example:

- A shortage of working capital;
- An unexpected spike in demand;
- Supply chain disruptions;
- Inaccurate inventory data;
- Human factor.

According to the study by Verhoef and Sloot (2006), the worldwide out-of-stock rate accounts for 7-8%. Nowadays companies are losing $634.1 Billion each year due to lost-sales, which makes up 4.1% of revenue for an average retailer (Dynamic Action and IHL, 2015). On the other hand, the customers' reaction to such an event appears to be extremely complex (Bijvank and Vis, 2011). For example, a study conducted by Gruen *et al*. (2002) revealed that only 15% of the customers that experienced a stock-out will wait for the product, whereas the remaining 85% will either buy a substitute product (45%), will visit a competitor (31%) or will decide to retain from buying (9%). Relatively similar proportions are found four years later by Verhoef and Sloot (2006), who concluded with the statement that only about a quarter of the customers will wait for the next replenishment. In any case, once a loyal customer will be tempted to try a product and service

offered by a competitor. Moreover, it is worth to note that if a substitution is made, the retailer also receives a collateral harm, because customers, as a rule, tend to switch to cheaper substitutes.

The mentioned facts demonstrate that unsatisfied demand is lost, at least partially, in practical settings. Nevertheless, as it can be seen from dozens of reviewed papers, the vast majority of IC models in academic papers assume that unsatisfied demand is backordered. By backordered it is literally meant that customers patiently and loyally wait as long as necessary for a new batch to arrive. Unfortunately, models of this kind are not fully representative for an immense share of real-world applications, especially in the field of retail.

The earliest work on lost-sales inventory models dates back to Hadley and Whitin (1963). The authors expressed the expected total cost in single-product (*r, Q*) IC model under the strict assumptions regarding the number of possible outstanding orders. The extensive study by Archibald (1981) analyzed IC systems under (*s, S*) policy and Poisson demand process. Archibald also proposed heuristic-based technique as an alternative to find an approximate solution. The (*r, Q*) model was modified to deal with stochastic replenishment lags by Ravichandran (1984) for exponential and Erlang distributions. Additionally, Johansen and Thorstenson (1996) extended the same model with Erlang distributed replenishment lead times incorporating discounts. Kalpakam and Arivarignan (1989) considered more general distributions of demand and replenishment lag.

It was also shown that the policy iteration algorithm can be applied to find optimal control parameters for more general replenishment lead time distributions, if orders do not overlap in time (Johansen and Thorstenson, 2004). Besides that, models in which the replenishment lag is a decision variable were developed by Ouyang *et al*. (2005). Hu *et al*. (2009) presented an alternative model, in which unsatisfied demand is partially backordered.

2.6.2. Inventory control with deteriorating items

In IC models, it is overwhelmingly assumed that the goods can be stored infinitely patiently waiting to be demanded. Nevertheless, such an assumption does not correspond to reality, since the vast majority of objects lose their valuable properties overtime. In IC theory such a property is defined as deterioration (Khanlarzade, 2014). For example, due to excessive humidity, evaporation, physical damage and natural rotting process, such products as food, medicine and donor blood are losing their quality over time horizon. Clothes and consumer electronics become outdated or out-

of-fashion, due to economic phenomenon of moral obsolescence. On its turn, such goods as alcohol and petrochemicals decay slowly and, thus, have an extremely long shelf-life (Figure 2.4).

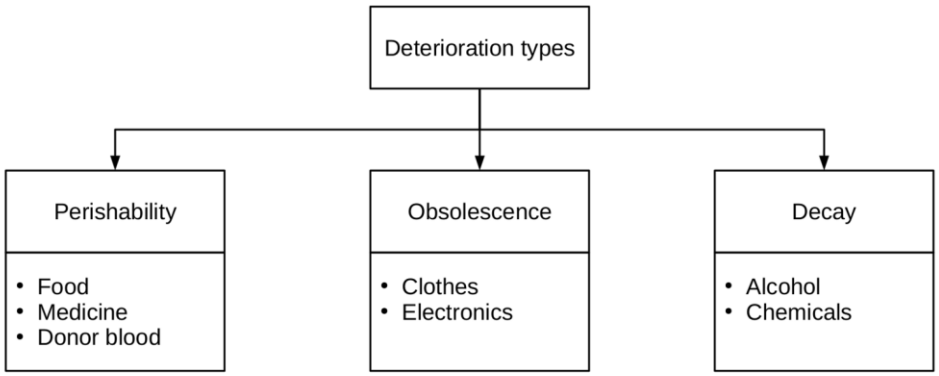

Figure 2.4. Deterioration types with examples

It is also should be pointed out that the current success of online grocery retail sparked interest to IC with perishability and gave researchers heart to study such models meticulously. According to the Statistics Portal (2018), in 2015 online grocery sales amounted to about $7 billion in the USA. Moreover, this figure is expected to rise up to $18 billion by 2020 and up to $26.87 billion by 2025 (Hexa Research, 2018). As it is concluded by Pitts *et al*. (2018), such impressive growth is attributed to technological advancements and aggressive service expansion along with increasing consumer acceptance. Thus, the future for online grocery retail looks even brighter.

The study on IC models with deterioration dates back to Whitin (1957). However, it was Ghare and Schrader (1963), who firstly attempted to derive optimal IC policies for perishable products. The first EOQ model that incorporated deterioration was described by Goyal (1985). The model allows one to obtain optimal order size, however, the set of assumptions is quite strict, namely, the demand rate is constant, shortages are prohibited and planning horizon is infinite. Considering lot size and capacity planning as simultaneous, the model was extended by Van Beek *et al*. (1985) to include obsolescence.

Inventory-dependent demand is quite a common and broadly studied phenomenon in retail. For example, Datta and Pal (1990) considered such an IC model and managed to find the optimal order quantity for it. Two years later Pakkala and Achary (1992) proposed two-echelon IC model for deteriorating products with finite replenishment rate, constant demand rate and allowed shortages. The paper clearly contributed to IC theory by comparing several cost structures. In the same year Xu and Wang (1992) described IC model for perishable products with linear time-

dependent demand. In this paper the optimal replenishment policy is derived applying extended Wagner-Whitin algorithm (Wagner and Whitin, 1958). Hariga (1993) demonstrated IC model for deteriorating products with infinite production rate, allowed backlogs and finite time horizon. Shortly after that, Wee (1993) extended EOQ model incorporating deterioration. In this model partial backlogging is allowed.

Assuming a constant perishability rate, the IC models in which both demand and supply are time-dependent are studied by Balkhi and Benkherouf (1996). One year later they also published an algorithm to find the optimal replenishment lead time (Balkhi and Benkherouf, 1997). It is worth to note that in the considered IC system incorporated time-varying deterioration rate and growing demand. The similar model for products with deterioration rate under Weibull distribution is presented by Chen (1998). Balkhi (1999) proposed an IC model for integrated manufacturing system. The author assumed that such parameters as supply rate, demand and deterioration depend on time.

Papachristos and Skouri (2000) presented a finite planning horizon IC model that incorporated constant deterioration rate, time-dependent demand along with partial backlogging. Wu (2002) used classical EOQ model complementing it with constant deterioration rate and stochastic demand under Weibull distribution. In this model, partial backlogging was allowed and backlogging rate depended on replenishment lead time. An inventory-production model for deteriorating products with constant finite production rate and time-dependent demand in finite planning horizon is considered by Sana *et al*. (2004). The "Box-Complex" algorithm-based solution is proposed and the optimal number of production cycles are eventually obtained.

An unusual (*r, Q*) IC model with permissible delay in payment for bigger order quantities was developed by Chang (2004). Besides deterioration, the model also incorporates inflation. Such optimal control parameters as order size and reorder point are obtained for a numerical example. One more IC model incorporating inflation is proposed by Chern *et al*. (2008). Besides, the model incorporaed inventory-dependent deterioration rate and price-dependent demand. Mahata (2012) discussed optimality of classical control policies for inventory models that deal with deteriorating products in single-echelon supply chain.

2.6.3. Markovian Demand Models

The crucial assumption in the vast classical IC models is that the demands in different periods are independent of environmental factors and identically distributed. However, in real life a plethora of stochastically changing environmental factors and uncertainties including fluctuating markets, stages of a product life cycle and even weather can have a tangible influence on demand. Moreover, these environmental states are distinguished by stochasticity – exogenous or controlled. For such cases, the approach based on Markov chains provides a natural and flexible alternative for modeling the environment dependent demand. In such an approach, environmental parameters are represented by demand states of a Markov process. So as demand is a random variable with a distribution function that depends on the demand state in the given period (Beyer *et al*., 2010).

The first IC model with Markov-modulated demand dates back to Karlin and Fabens (1960), who introduced an IC model with demand changing in accordance with the states of a corresponding Markov chain. Two years later they considered the problem with dependent probabilistic demand (Iglehart and Karlin, 1962). The authors studied an inventory model with zero setup cost and the demand process given by a discrete-time Markov chain. Such that, in each period, the current state of the Markov chain prescribes the probability distribution function for demand.

Sethi and Cheng (1997) studied a discrete-time version of the IC with general demand distributions and backlogging. After that Cheng and Sethi (1999) successfully extended these results incorporating lost-sales. However, for this case the optimal policy is derived only under quite unrealistic assumption of zero replenishment lead time. Another seminal work by Chen and Song (2001) studied an IC system with Markov-modulated demand integrated into multi-echelon supply chain. Bensoussan *et al*. (2008) presented so-called censored news vendor-models. Censored news-vendor models are in fact Markovian demand models with partially observed demands. Besides, it is worth to mention a recent study conducted by Avci *et al*. (2019), who proposed an approach for deriving optimal control parameters for infinite-horizon average-cost discrete-time IC problem with unbounded demand. The IC system was operation under belief-dependent base stock policy. Backlogs were allowed and replenishment lag was deterministic.

2.7. Conclusions

For over a century IC has been a major point of discussion in academic, scientific and business worlds. Analytical models still hold the huge lay of the land in IC theory, especially in academia. Unfortunately, they do not guarantee the optimal solution, if the initial assumptions and considerations are violated. Moreover, such assumptions frequently do not correspond to real-world IC problems. On the other hand, real-world business-driven IC is frequently distinguished by multidimensionality, non-decomposability and replete with numerous specialties and subtleties. Due to a high degree of complexity and analytically intractable of business-driven problems, IC theory adopts immense arsenal of methods from different scientific disciplines. For instance, the approaches based on control theory demonstrate efficiency in modeling of single-product IC systems, when products are homogeneous enough to be treated as a flow. On the other hand, the approaches based on dynamic programming are efficient in deriving nearly-optimal control parameters for stochastic IC problems. However, dealing with large-scale multi-product multi-echelon IC systems, the computational burden becomes immense, which compels to incorporate approximation techniques along with simulation into the algorithmic framework. This tendency to choose computationally efficient numerical approximations over explicit analytic solutions rises the popularity of SO techniques. A fortiori, simulation models may be exclusively tailored to industrial needs providing a modeler with a tool to deal with real-world uncertainty in unconstrained way. Unfortunately, simulation, especially detailed, is both time and memory consuming. In the light of these facts, it is considered be more reasonable to use an alternative cheaper-to-compute metamodel, which is specifically built in order to approximate a target simulation with a sufficient degree of accuracy. Thanks to recent advances and rapid development in deep-learning, metamodeling with ANN looks extremely promising. Nevertheless, ANNs are usually developed manually by data scientists and artificial intelligence developers, which is quite fault prone and requires the sheer amount of time. In light of this fact, an interest in automated neural architecture search methods is growing drastically, and ANN-based metamodels seem to be the next logical step in evolution of optimization of IC systems.

Nowadays lost-sales seriously damage retail industry, furthermore, stock-out in supply chains and production-inventory systems cause disruptions and idling. That is why, in realistic settings unsatisfied demand should be considered to be at least partially lost, which make IC with lost-sales an extension of great importance. Taking into account the fact that such products as food, medicine

and donor blood lose their valuable properties overtime, and current success of online grocery retail, IC with perishability receives rigorous attention in this treatise. Lastly, the Markov-modulated approach provides a natural and realistic way of modeling uncertainty in IC systems associated with demand. Namely, by linking the demand dynamics with an underlying Markov chain, one accesses an elegant way to represent the effect of environmental factors on demand.

3. METHODOLOGICAL BACKGROUND

Nowadays artificial neural networks have outperformed traditional approaches in such tasks as image recognition, machine translation, speech recognition, and fitness approximation (Goodfellow *et al*., 2016). Such a success of artificial neural networks did not bypass simulation community causing the surge of interest to metamodeling with artificial neural networks (Lechevalier *et al*., 2015).

Despite astonishing performance of artificial neural networks, there is a serpent in this metamodeling Eden. Although neural networks are universal function approximators, it is still important to pay attention to the structure. How many layers should it contain? How many neurons should be in each layer? Which optimization algorithm should be chosen? Currently data scientists, artificial intelligence developers and other human experts try to answer these persistent questions by being deeply involved into development, fine-tuning and customization of artificial neural network-based metamodels. Since the metamodeling with artificial neural networks involves human labor and, thus, consumes lots of time and financial resources, the metamodeling automation is extremely tempting.

Taking into consideration recent success of neuroevolutionary approaches (approaches that combine neural networks with genetic or evolutionary algorithms) in neural architecture search (Real *et al*., 2019) and hyperparameter optimization (Elsken *et al*., 2019), its application to metamodeling of IC systems becomes extremely promising.

3.1. Neuroevolution

The neuroevolution is an umbrella term that unites various approaches for optimizing neural network weights, architectures and hyperparameters using evolutionary computation. Neuroevolutionary approaches are inspired by the undeniable success of the evolution of biological nervous systems. The same nervous systems that ultimately allowed us to think, feel, create and possess other cognitive properties that humanity is very proud of. Thus, neuroevolution can be considered as a biologically-inspired computational intelligence (Rawal *et al*., 2010).

Neuroevolution is guided by gradual increase in metrics of overall performance, which makes it resembling Darwinian natural selection driven by reproductive success. More specifically, neuroevolutionary approaches take advantage on abstractions of evolution, for example, genetic

algorithm, in order to construct abstractions of biological nervous system, for example, artificial neural networks (Risi *et al*., 2012). Despite the fact that an overall objective is extremely ambitious, namely, to evolve complex artificial neural networks capable of human-level intelligent behavior, industrial applications are promising even today.

Applied to such tasks as Neural Architecture Search (NAS) and Hyperparameter Optimization (HPO) neuroevolution can be seen as subfield of automated machine learning (Hutter *et al*., 2019). It is also worth to mention that since automated machine learning is new and rapidly developing field, terms and definitions within its framework are quite vague and have significant overlaps. In order to prevent confusion, these concepts and terms are clearly defined. Automated machine learning (AutoML) can be defined as the process of full or partial automation of applying machine learning to a real-world problem. NAS, on the other hand, is an approach for automating topology tuning of ANNs (Zoph and Le, 2016). Since values of hyperparameters guide the whole process of learning, HPO can be defined as the problem of searching for such a set of hyperparameters that lead to the machine learning model with the highest performance (Feurer and Hutter, 2019). Relations between AutoML, NAS, HPO and neuroevolution are demonstrated in the Figure 3.1.

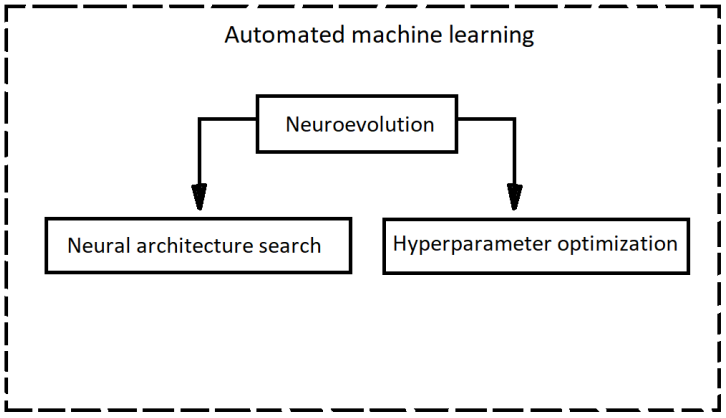

Figure 3.1. Relations between AutoML, NAS, HPO and neuroevolution

As a rule, neuroevolutionary techniques follow select-modify-evaluate loop typical for genetic algorithms. Such that there is a vector of encoded parameters (a chromosome) that results an ANN of particular architecture (an individual). So, the analogy with biological evolution becomes visible to the naked eye. Namely, every single ANN is an amalgam of characteristics determined by the genes in its chromosomes. This ANN is feed with training data, and its performance is measured,

according to the chosen metrics. This metrics is then treated as a fitness value of the corresponding chromosome. After all the individuals in the population have been evaluated, the fittest of them are mutated and crossed over with each other in order to produce potentially even fitter offspring, that will pass to the next generation. Such that this search for the best chromosome continues until an ANN with a sufficiently high fitness is found or stop conditions are met.

Historically, evolutionary algorithms were widely used to evolve neural topologies as well as synaptic weights. Such applications date back to Angeline *et al*. (1994), and it eventually gave a rise to a method known as neuroevolution of augmenting topologies (NEAT) (Stanley and Miikkulainen, 2002). NEAT is a canonical example of neuroevolution that affects both architecture and weights. It is quite popular and viable approach that takes advantage on genetic algorithm to optimize neural architecture as well as its weights (Stanley *et al*., 2009). However, real-world problems often require ANNs with millions of weights that is why gradient-based methods for weight optimization currently outperform evolutionary. That is why, state-of-the-art neuroevolutionary approaches therefore again use gradient-based methods for optimizing weights and only use genetic algorithms for NAS and HPO (Xie *et al*., 2019).

Neuroevolutionary techniques also differ in how they populate, select, mutate and mate chromosomes. For example, Real *et al*. (2019), and Liu *et al*. (2018) apply tournament selection to select parents and initialized initial population randomly. Elsken *et al*. (2019) applied Lamarckian inheritance using network morphisms. Additionally, Real *et al*. (2017) also encoded the learning rate into the chromosome, thus allowing it to be varied. Besides that, it is important to mention that in their seminal study Real *et al*. (2019) compared neuroevolution with reinforcement learning. It was concluded that reinforcement learning and neuroevolution performed equally well in terms of accuracy, however, neuroevolution resulted more compact and, therefore, cheaper-to-compute models.

3.2. Artificial neural networks

In every nook and cranny, there exist problems that cannot be explicitly expressed in algorithmic form. Such problems can involve extremely high dimensionality, noisy data or various subtle factors that are difficult to consider. For instance, the human ability to approximate real estate price is a widely praised example in the literature on artificial intelligence (Rossini, 2000). However, non-trivial cognitive functions are not unique to humans, and bats navigation using sonar

is a good example of such a function. Highly-complex computation inside the brain the size of an apricot process information of echolocation signals extracting the knowledge on velocities, distances, sizes, densities and other features of the surrounding world. Another astonishing fact is that these sophisticated computations are performed in fractions of a second, much faster, than the fastest digital computers of the same size (not to mention power consumption) can handle. Since learning is the central concept behind these capabilities, an artificial neural network can be defined as a computational model that mimics the way the biological central nervous system performs a particular function.

The history of ANNs starts with a breakthrough. As soon as the first models of neural networks based on threshold switchers are introduced, it was demonstrated that even such rough representations are able to approximate any arithmetic function and reproduce almost any binary logic (McCulloch and Pitts, 1943). Fifteen years later, inspired by the retina Rosenblatt came up with several versions of the single-layer perceptron (Rosenblatt, 1958). After perceptron models were deeply analyzed, it was concluded that due to linear inseparability many problems cannot be approached with single-layer perceptron models (Minsky and Papert, 1969). Ultimately, the issue was resolved, and the single-layer perceptron restrictions were removed with the development of backpropagation of error (Werbos, 1974). Unfortunately, this work was not widespread and backpropagation of error was rediscovered later as a generalization of the delta rule (Rumelhart *et al*., 1986). It was later formally proved that a feedforward ANNs with at least one hidden layer equipped with sigmoid functions are capable to approximate any continuous functions (Cybenko, 1992). The theorem was extended for other activation functions to become the universal approximation theorem (Hornik, 1993). Besides that, in the context of this research it is worth to mention that ANNs equipped with a rectified linear unit (ReLU) can also approximate continuous functions (Hanin, 2018).

An ANN can be defined as a triple (N, V, w), where N is the set of neuron nodes, V a set of connections between neurons i and neuron j, such that $\{(i, j) \mid i, j \in \mathrm{N}\}$. In this context, it is convenient to view w as a function $w{:}V{\rightarrow}\mathrm{R}$ that assigns the value of weights between a pair of neurons (i, j), which is direct analogy to the way the information is transferred between neurons via synaptic connections. Considering a single neuron j, it can be pointed out that in the practical architectures there are lots of neurons connected to it. That is why, the input of the following layer is the result of propagation function applied to the previous one. In applications a weighted sum is

a conventional choice $net_j = \sum o_i w_{i,j}$, where net_j is an input of a neuron j and o_i is an output of a neuron i. The reactions of neurons to the input values depend on the activation function, which can be formalized for the neuron j as $act_j(t) = f_{act}(net_j(t), act_j(t-1), \theta_j)$, where t is a time step during the learning procedure, so as the activation function $f_{act}(.)$ maps network input and previous activation state $act_j(t-1)$ into a new one $act_j(t)$ with regard to the threshold value θ_j (Kriesel, 2007).

Relying on the universal approximation theorem and its relevance to metamodeling, this research focuses only on a specific ANN subclass, namely on feedforward fully connected ANNs. In such networks all neurons belong to the layers of three types, namely input, hidden and output. Feedforward fully connected ANNs are distinguished by the property that each single neuron in one layer is connected to all the neurons of the following layer towards the output layer and never vice-versa (Figure 3.2.).

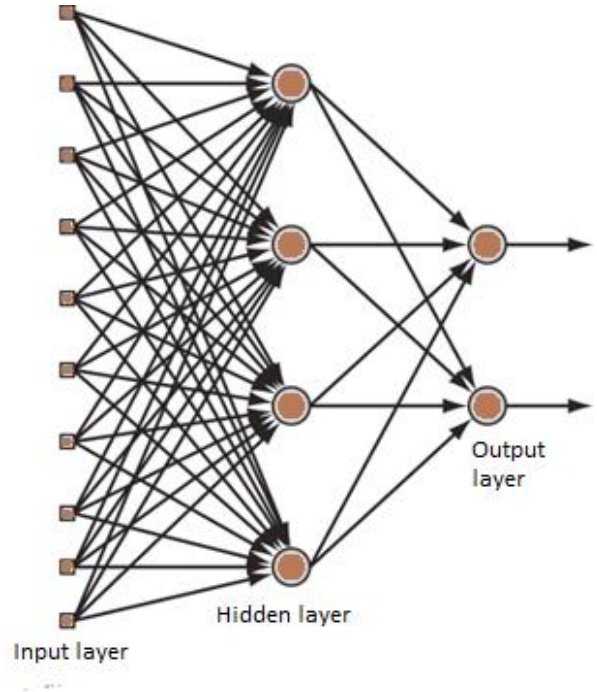

Figure 3.2. The example of the feedforward fully connected ANN (Kriesel, 2007)

Supervise learning in general and its application to metamodeling can be conveniently seen as an optimization problem, because it pursues a goal to derive a function that is capable to map a set of inputs to the corresponding correct output with a sufficient degree of accuracy. The most intuitive way to perform such an optimization is by applying procedures based on gradient descent.

Backpropagation of error (BP) is an umbrella term that includes quite a diverse family of algorithms that calculate the loss function with respect to the weights of ANN, namely $\Delta w_{ij} = -\rho \frac{\partial Err}{\partial w_{ij}}$, where Δw_{ij} is added to the old weight, such that the product of the gradient of error *Err* (which depends on the loss function) with respect to the weight w_{ij} and learning rate ρ (Krose *et al*., 1993). Multiplication by -1 in this case guarantees that the weight will change in a way that decreases the error. The complete derivation of BP is brilliantly described by Rojas (1996).

3.3. Genetic algorithms

Since the days of Charles Darwin such organs of unfathomable complexity as the eye enraptured scientists of different fields (Darwin, 1859). It was a matter of time before evolution would one day be seen as an optimization process (Mayr, 1988). Despite the fact that biological evolution does not result perfection, nevertheless, for every living organism evolution was able to discover highly efficient engineering solution to burning problems posed by hostile environment. In this regard, evolution provides strong impulse of inspiration for solving problems that were previously considered to be intractable. It is extremely tempting, therefore, to describe evolution in the form of an algorithm in such a way that complex optimization problems, previously appeared intractable, can be solved by iterative selections and mutations (Holland, 1992).

Genetic algorithms along with evolution strategies, and evolutionary programming belong to the broad class of evolutionary algorithms. Despite significant differences, all the evolutionary approaches have fundamental commonalities. All of them are generally inspired by natural phenomena of evolution, survival of the fittest and Darwinian struggle to survive. Since evolutionary algorithms are gleaned from biological evolution of organic form, they grasp its crucial aspects, namely reproduction, variation, competition, and selection of the fittest one among the contending individuals in a population. In is worth to emphasize that the concept of selection contains implicitly nested within it the concept of fitness. Namely, fitness can be viewed as the probability that an amalgam of genes will propagate its genes through generations. The pivotal steps behind GA can be described in the following pseudocode (Luke, 2015):

```
N ← desired population size
P ← {}
for N times do
  P ← P ∪ {new random individual}
```

```
Best ← □
end for
repeat
  for each individual Pi ∈ P do
    AssesFitness(Pi)
    if Best = □ or Fitness(Pi) > Fitness(Best) do
      Best ← Pi
    end if
  Q ← {}
  end for
  for N/2 times do
    Parent Pa ← SelectWithReplacement(P)
    Parent Pb ← SelectWithReplacement(P)
    Offspring Ca, Cb ← Crossover(Copy(Pa), Copy(Pb))
    Q ← Q ∪ {Mutate(Ca), Mutate(Cb)}
  end for
  P ← Q
until time runs out
return Best
```

The most essential ideas and concepts behind GAs can be traced back to Holland (Holland, 1962), who significantly developed them shaping the general theory of adaptivity (Holland, 1967) and the schemata theory (Holland, 1969). Eventually, his solemn work on examining different reproductive plans gave a rise to genetic algorithms (Holland, 1971). Already in these early works, the main features underlying every single genetic algorithm were highlighted. These features include chromosome representation (encoding), selection, crossover and mutation. It is also important to mention that, since the paradigm of evolutionary strategies adopted crossover, the distinguishing operator of GAs, the boarder between various evolutionary algorithms have become completely vague and blurred and, thus, have meaning rather in historical context (Back *et al*, 2018).

3.3.1. Chromosome encoding

Although chromosome is typically encoded as a string of bits, GA is not restricted to binary representations. For example, Davis (1991) and Wright (1991) represented candidate solutions as lists of real numbers. Grefenstette (1986) encoded chromosomes in the form of permutations and Antonisse and Keller (1987) as directed graphs.

Besides that, there is variability even among different binary schemes. In this regard, it is worth to mention that empirical studies conducted by Caruana and Schaffer (1988) highlight that reflected binary scheme, also known as Gray code (Frank, 1953), is significantly superior to the standard binary coding. Gray code confers an advantage, because exactly one bit has to be flipped in order to reach the nearest value. As the result, the problem pertaining to large Hamming distances evaporates, and gradual search in the continuous search spaces can be conducted without let or hindrance (Deb and Agrawal, 1995).

3.3.2. Mutation and crossover

Mutation is the most simple and straightforward, nevertheless, the most important operator for inserting variations into the population. In case of all binary representations, mutation is achieved by randomly flipping bits along the chromosome. Such an approach grasps the pivotal analogy with evolution, where an allele of a gene is replaced by an alternative one. It is worth to note that in many optimization problems mutation operator is successfully applied alone without crossover, thus, its role should not be neglected (Mathias and Whitley, 1994).

Nevertheless, in classical applications of genetic algorithms mutation is used rather as a background operator for diversifying chromosomes that can be affected by crossover after that (Goldberg, 1989). Thus, it can be stated that crossover is the most substantial genetic operator that cannot be disparaged in practical applications. The grist behind crossover operators is easy to grasp. Let's assume that there are two individuals of the same species with high fitness, however the features that make these individuals highly fit are different. In this case, these features are combined begetting a new individual, who possess even greater fitness. For instance, mother with a keen ear and father with good fine motor skills will have a chance of giving birth to a great pianist. Indeed, due to the fact that we, unfortunately, have no clue which features determine high fitness, the best strategy is to recombine features randomly. Inevitably, however, time to time such a crossover will combine the negative features throwing away the positive, which result an unfit individual. Fortunately, as in nature, such a creature does not last long due to selection.

More formally, crossover is considered as a form of convergence-controlled variation (Radcliffe, 1991). If a particular string of bits carries the highest fitness, then all the population will ultimately converge to this string. Such that, from iteration to iteration exploration narrows down the focus to smaller intervals of the search space. Besides that, the efficiency behind GA can

be explained within the framework of the building block hypothesis, which states that by combining features of fit parents, crossover will, most likely, result offspring with even higher fitness (Holland, 1975). In this regard, the contrast between mutation and crossover becomes clear, namely mutation increases variation, whereas crossover converges the population to the optimum.

At the dawn of evolutionary computing, researchers applied multi-point crossovers (De Jong, 1975). However, this approach has an undesirable property. Such crossovers tend to split up the fit building blocks. In light of this fact, uniform crossover has been introduced in order to compensate this drawback (Ackley, 1987). In this case, the number of crossover points is not constant. Instead, the decision to flip is made individually for every single bit in chromosome.

3.3.3. Selection

Selection is a driving force of evolution, whether natural or simulated. Without selection operator, exploration of the search space would be no better than random-walk. Every time selection operator is applied, unfit individuals are replaced with fitter ones. As the result, the average fitness of the population increases and bad genetic material is dumped. This effect is known as the loss of genetic diversity (Razali and Geraghty, 2011).

Classical selection proportional to fitness is very sensitive to the distribution of fitness in the population, and, thus, suffers from loss of diversity (Hancock, 1994). Seeking the way to overcome these disadvantages researchers come up with the method of ranked selection (Whitley 1989). Ranked selection ranks the individuals in the parent population, such that the chance of selection depends linearly on these ranks. Nevertheless, contemporary approaches, especially in the field of neuroevolution, use tournament selection, because it is robust approach for working with noisy fitness functions (Goldberg and Deb, 1991). The method runs several “tournaments” among *t_size* individual solutions randomly driven from the population, such that a winner is determined depending on his fitness. After that this individual is inserted into the next generation. Among other benefits, tournament selection intensiveness can be adjusted by varying the parameter *t_size* (Miller and Goldberg, 1995).

3.4. Supplementary methods

Beside the core neuroevolutionary components (ANN and GA), this research uses several methods of mathematical statistics for validating models, normality testing, heteroskedasticity testing and clustering.

3.4.1. K-fold cross-validation

This research also incorporates k-fold cross-validation for model assessment and selection. The method estimates the expected generalization error by splitting a dataset apart into *k* approximately equal parts using the larger fraction of data to train the model and smaller one to validate (Figure 3.3.).

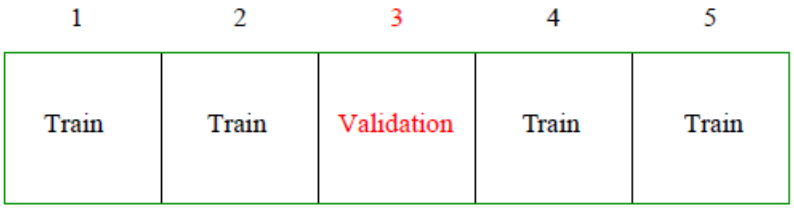

Figure 3.3. 5-fold cross-validation (Hastie *et al.*, 2005)

Such that the model is trained using *k*-1 parts of data and the loss or prediction error is computed based on the remaining part. This procedure is conducted for $i = 1, 2, \ldots, k$ and prediction errors are averaged after that.

3.4.2. Anderson–Darling test

The Anderson-Darling (AD) statistical test is used to test the null hypothesis that a sample is taken from a normally distributed population. After the observations are sorted $\{x_1 < x_2 < \ldots < x_n\}$ the statistic *A* can be calculated according to the following formula:

$$A^2 = -n - \sum_{i-1}^{n} \frac{2i-1}{n} \left[\ln\left(CDF(x_i)\right) + \ln\left(1 - CDF(x_i)\right)\right], \quad (3.1)$$

where *n* is a sample size and *CDF(.)* stands for the cumulative distribution function (Anderson and Darling, 1954).

If the calculated statistic is smaller than the critical value, the null hypothesis that the data follows the considered distribution can be accepted for the corresponding significance level. As it is concluded by Razali and Wah (2011), AD test is extremely robust and reliable, especially for samples with more than 100 observations.

3.4.3. Goldfeld–Quandt test

The parametric Goldfeld–Quandt test (GQ) is a simple to implement method to diagnose heteroskedasticity of errors in regression models. GQ compares the variance of errors from two discrete subgroups. These subgroups do not have to be the same size, however, the test assumes that the errors follow a normal distribution. The resulting test statistic corresponds to the F-test of equality of variances, namely, it is the ratio of mean square residual errors for the independent variables from the two subgroups (Goldfeld and Quandt, 1965).

3.4.4. Clustering validation

Since in practical settings it is computationally hard to consider each item individually, chapter 7 heavily relies on clustering in order to demonstrate how similar stock keeping units can be grouped together and be operated under a common IC policy. It is important to note that the methodology described in this section was previously published in the author's paper (Jackson *et al*., 2018b). Due to the fact that the ground truth is not known, clustering algorithms are compared using internal cluster validation tests. Namely, silhouette index, Calinski-Harabasz index (CH) and Dunn index (DI) were calculated in order to compare clustering results based on such properties of clusters as density, shape and separability.

Silhouette validity index refers to a method of internal clustering consistency validation. The index reflects cohesion and separation of clustered data and may be defined as follows:

$$s(i) = \frac{b(i)-a(i)}{\max\{a(i),b(i)\}}, \tag{3.2}$$

where *a(i)* is the average distance between an observation *i* and all the other observations, *b(i)* is the lowest average distance between *i* and observations in clusters of which *i* does not belong to (Rousseeuw, 1987). The values of index are bounded in range $-1 \leq s(i) \leq 1$. The score is generally higher for dense and well separated clusters, which corresponds to a concept of a cluster.

CH, also known as pseudo F-statistics, is a traditional and computationally fast method for internal clustering validation. For *k* clusters the CH-index *s(k)* is defined as the ratio of the between-clusters dispersion mean and the within-cluster dispersion:

$$s(k) = \frac{tr(B_k)}{tr(W_k)} \frac{N-k}{k-1}, \tag{3.3}$$

where $tr(B_k)$ is the trace of the between-clusters dispersion matrix and $tr(W_k)$ is the trace of within-cluster dispersion matrix (Calinski and Harabasz, 1974). As a side note, the value of CH-index is generally higher for convex clusters which make it biased towards DBSCAN and other density-based clustering techniques.

DI measures a compactness and well-separation of clusters. Namely, the score is higher if clustering produces a small variance between observations within one cluster keeping mean values of different clusters sufficiently far apart:

$$DI = \min_{i=1,\dots,n_c} \left\{ \min_{j=i+1,\dots,n_c} \left(\frac{d(c_i, c_j)}{\max_{l=1,\dots,n_c} diam(c_k)} \right) \right\}, \tag{3.4}$$

where $d(c_i, c_j) = \min_{x \in c_i, y \in c_j} d(x, y)$ stands for the intercluster distance between clusters c_i and c_j, $d(x, y)$ is the Euclidean distance between observations *x* and *y*, $diam(c) = \max_{x,y \in c} d(x, y)$ is the diameter of a cluster (Yatskiv and Gusarova, 2005). The DI values lie in the interval from 0 to infinity such that the higher values correspond to better clustering.

4. INVENTORY CONTROL MODELS UNDER CONSIDERATION

The models are designed for further algorithmic implementation in the form of a discrete-event simulation (DES), and described using Hurlimann's indexed notation (Hurlimann, 2007) as a base complemented with set theory notation (Winskel, 2010) and Iverson's brackets (Iverson, 1962). The DES implementation of both models in Python 3.7 is conducted based on the events list algorithm (Henriksen, 1977). These models incorporate extensions of special interest specified in the chapter 2 and are used in the following chapters as the baseline IC models to test the neuroevolutionary framework. Besides, nonlinearity and self-similarity of the inventory dynamics in the described IC models are studied in the related research (Jackson and Grakovski, 2019).

4.1. Stochastic multiproduct inventory control system with perishability

The first DES model under consideration is referred to in the following chapters simply as the model 1. It is mainly based on the recent work (Jackson and Tolujevs, 2018; Jackson, *et al*., 2018a; Jackson, 2019a). The model corresponds to inventory and warehousing operations in the "Trialto Latvia" LTD. In addition, the model adopts several new features from related papers. Namely, the model incorporates perishability mechanics quite similar to one used by Duan and Liao (2013). Besides that, the model operates with multiple products under the constrained total inventory in a similar way as the transhipment inventory simulation described by Hochmuth and Kochel (2012). As the result, the model can be characterized as multiproduct and stochastic with lost-sales, discounts and perishability. Appendix 1 contains a Python 3.7 implementation of the DES model.

4.1.1. Material flow

The IC system under consideration operates with a sequence of products $P = (p_1, p_2, \dots, p_n)_{n\in\mathbb{N}^+}$ under the limited total storage capacity I_{max} (Figure 4.1.). For the sake of computational efficiency only those moments of time, in which the system parameters change are considered. Timings of such events constitute a sequence $T = (t_1, t_2, \dots, t_n)_{n\in\mathbb{N}^+}$. Thus, the value of $t_n - t_1$ can be seen as the planning horizon. Due to the fact that the IC system operates with perishable products, the storage is denoted as a sequence of independent lots replenished at different moments of time $S_t^p = (s_1^{p,t}, s_2^{p,t}, \dots, s_n^{p,t})_{n\in\mathbb{N}^+}$. Such that for each S_t^p there is a corresponding sequence of days to expiration (DTE) $E_t^p = (e_1^{\mathrm{p,t}}, e_2^{\mathrm{p,t}}, \dots, e_n^{\mathrm{p,t}})_{n\in\mathbb{N}^+}$. These

sequences apparently have the same cardinalities $\left|S_t^p\right| = \left|E_t^p\right| \forall p \in P, \forall t \in T$. Therefore, for every single product at each instance of time, there is an inventory level $I_t^p = \sum_{i=1}^{n} S_t^p$. In order to model how the DTE decreases over time, the following function is defined:

$$E_{t+1}^{p} = \varepsilon(E_t^p, t_i, t_{i+1}) = (e_0^{p,t} - (t_{i+1} - t_i), e_1^{p,t} - (t_{i+1} - t_i), \ldots, e_n^{p,t} - (t_{i+1} - t_i)) \qquad (4.1)$$

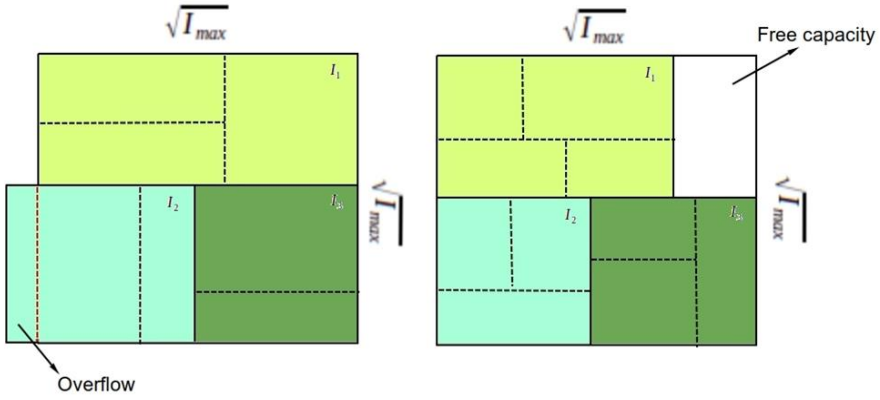

Figure 4.1. The limited storage capacity can be illustrated as the square with side length equal to $\sqrt{I_{max}}$

In each discrete time lag $\Delta t = t_i - t_{i-1}$, all empty and expired lots are removed. Formally, $\forall e_i^{p,t} \leqslant 0,\ S_t^p \leftarrow S_t^p / s_i^{p,t}, E_t^p \leftarrow E_t^p / e_i^{p,t}$ and $\forall s_i^{p,t} = 0,\ S_t^p \leftarrow S_t^p / s_i^{p,t}, E_t^p \leftarrow E_t^p / e_i^{p,t}$ (Figure 4.2.). It is done for an apparent reason, expired goods are unfit for consumption, or sale. In addition, it is crucially important to count the number of expired products for later total expenses calculation:

$$Expired_t^p = \sum_{i=1}^{n} s_i^{p,t}[\Phi], \qquad (4.2)$$

where [Φ] is the Iverson bracket $[\Phi] = \begin{cases} 1\ if\ e_i^{p,t} \leqslant 0 \\ else\ 0 \end{cases}$ (Iverson, 1962).

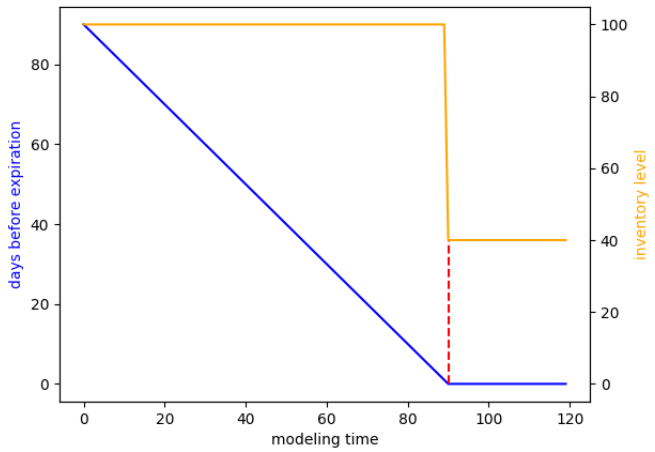

Figure 4.2. Illustration of the perishability mechanics

Assuming that $T^p_{demands} = (\hat{t}_1,\ \hat{t}_2, \dots, \hat{t}_n)$ includes only such time moments, in which a new demand d^p_t arises. Demand size is a random variable D under a known distribution $F_d = Prob(d \leqslant D)$. Thus, it is convenient to denote demand interarrival time as $a_i = \hat{t}_i - \hat{t}_{i-1}$, which is also a value of a random variable A under a known continuous distribution $F_a = Prob(a \leqslant A)$. Besides, a recursive function $f^{i=1}(.)$ that satisfies incoming demands depending on the available inventory capacity is defined:

$$f^{i=1}(s_i^{p,t}, d_t^p) = \begin{cases} s_i^{p,t} \leftarrow s_i^{p,t} - d_t^p \ if\ s_i^{p,t} \geq d_t^p \\ else\ s_i^{p,t} \leftarrow 0, f^{i+1}(s_{i+1}^{p,t}, d_t^p - s_i^{p,t}) \end{cases}, \tag{4.3}$$

where i stands for an index of the lot, from which demand is fulfilled.

In addition, it is important to count satisfied demands $Sales_t^p = \begin{cases} d_t^p\ if\ I_t^p \geq d_t^p \\ else\ I_t^p \end{cases}$ for later net profit calculation.

Besides, for all products in IC system there is a pair of control parameters (Q^p, r^p) that define an IC policy. It is important to note that control parameters are constant and do not vary over time. As soon as the current inventory level reaches the reorder point r^p, namely, $I_t^p \leq r^p$, the IC system places an order of size Q^p. In addition, a Boolean variable $status^p \in$ {True, False} is introduced in order to know, whether the batch is already ordered and on the way:

$$o_t^p = \begin{cases} Q^p,\ status^p \leftarrow True\ if\ I_t^p \leq r^p \text{and}\ status^p is\ False \\ else\ 0 \end{cases} \tag{4.4}$$

If an order is placed, such that $o_t^p > 0$, the inventory level will not be replenished immediately. Instead, there is a lag between the time, when the order has been placed and the time, when the order is delivered. Therefore, replenishment lead time is introduced to the models as a random variable L under a known distribution $F_l = Prob(L \leqslant l)$. This literally means that the batch of size o_{t-l}^p ordered at the moment of time $t_i \in T$ will be appended to the storage sequence $S_t^p \cup \{o_{t-l}^p\}, E_t^p \cup \{e^p\}$ at the moment of time $t_j \in T$, such that $t_j - t_i = l$. For this reason, a supply function *g(.)* is declared:

$$(S_{t+1}^p, E_{t+1}^p) = g(S_t^p, E_t^p, Q^p) = \begin{cases} S_t^p,\ E_t^p\ if\ o_{t-l}^p = 0 \\ else\ S_t^p \cup \{o_{t-l}^p\}, E_t^p \cup \{e^p\},\ status^p \leftarrow False \end{cases} \quad (4.5)$$

It is important to emphasize that either a backorder-event $d_t^p > I_t^p$ or an overflow-event $\sum_{t=1}^n S_t^p > I_{\max}$ can happen. Therefore, these events are counted for later cost function calculation:

$$Backorders_t^p = \begin{cases} 0\ if\ d_t^p \leq I_t^p \\ else\ d_t^p - I_t^p \end{cases};\ Overflow_t^p = \begin{cases} 0\ if \sum_{t=1}^n S_t^p \leq I_{\max} \\ else \sum_{t=1}^n S_t^p - I_{\max} \end{cases} \quad (4.6)$$

Obeying the following order of operations: 1) check expiration dates 2) replenish products 3) satisfy the demand, the physical can be simulated as follows:

$$(S_{t+1}^p, E_{t+1}^p) = f(g(S_t^p, \varepsilon(E_t^p), Q^p)) \quad (4.7)$$

Taking into consideration that $I_t^p = \sum_{t=1}^n S_t^p$, The overall inventory dynamics can be expressed in the following way:

$$I_{t+1}^p = min(max(I_t^p + o_t^p - d_t^p - Expired_t^p, 0), I_{max}) \quad (4.8)$$

With the last equation it is also emphasized that the inventory level cannot be negative or higher than I_{max}.

4.1.2. Monetary flow

At the level of financial flows, a mild assumption is made. Namely, monetary transaction is received immediately after a product is sold. Five composite costs are considered: ordering costs, inventory costs, backorder costs, overflow fee and recycle fee.

Ordering cost includes both purchase price and transportation cost and costs for associated logistics (loading, packaging, unloading). The cut-off point quantity discount is adopted. Such that discount is given only to the extent that the order exceeds a cut-off point:

$$c^p(Q^p) = \begin{cases} c^p for\ 0 < Q^p \leq \beta_1 \\ k_2 c^p for\ \beta_1 < Q^p \leq \beta_2 \\ k_n c^p for\ \beta_{\text{n-1}} < Q^p \leq \beta_n \end{cases}, \tag{4.9}$$

where ordering costs for a unit (e.g. box, container, pallet) $c^p(Q^p)$ is a function of an order quantity, such that $B^p = (\beta_1^{\text{p}}, \beta_2^{\text{p}}, \ldots, \beta_n^{\text{p}})$ is a series of cut-off points and $K^p = (1, k_2^p, \ldots, k_n^p)$, $\forall k_i^p \in [0,1]$ is a series of corresponding discount factors.

From a business point of view, a quantity discount is a stimulus offered to a customer that results in a decreased cost per unit, when a commodity is purchased in a greater amount. That is a common practice to motivate customers to purchase in greater amounts. As the result, a win-win situation appears, namely, a seller can increase turnover, and a customer receives more favorable prices. That is why, the cut-off point quantity discount introduces a quite realistic return-to-scale mechanics to the model. Depending on the real-world context, unit inventory cost h^pcan consist of handling costs and opportunity loss. Opportunity loss in this context is the possibility of using the capital for other purposes.

Inventory cost is constant and set for a unit of modelling time. Such that, $Inventorycost^p = h^p \sum_{i=1}^{n} I_i^p \Delta t_i$. Every single out-of-stock is considered to be associated with a loss of business reputation. When a product is backordered, a customer is directly stimulated to search for a substitute. This fact provides a possibility that once loyal customer can potentially switch, which leads to eventual loss of a market share. In this regard, for every product in the IC system out-of-stock is related to a constant fee b^p. In addition, overflows are penalized by a constant fee ω^p. In real world, such expenses are related to the sudden need for reverse logistics. A fortiori, when a lot is perished, quite similar penalty κ^p related to reverse logistics of expired goods arises. Namely, such a batch must be sent for recycling. Based on the introduced variables total costs related to a product *p* may be calculated as follows:

$$TC^p = c^p \sum o_t^p + h^p \sum I_t^p \Delta t + b^p \sum Backorders_t^p + \omega^p \sum Overflow_t^p + \kappa^p \sum Expired_t^p \tag{4.10}$$

Taking into account that each unit of product p is sold by a constant $price^p$, the net profit associated with this product is $Profit^p = price^p \sum sales^p - TC^p$. Thus, total net profit of the IC system is $\sum_{p=1}^{n} Profit^p$, which is treated as the simulation output. The Figure 4.3. demonstrates the dynamics of physical and monetary flows.

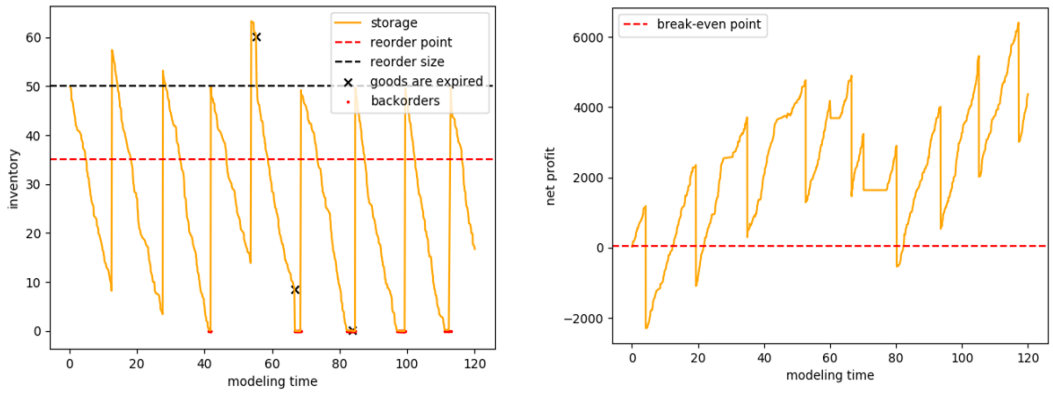

Figure 4.3. The dynamics of physical and monetary flows

4.1.3. Algorithmic implementation

Despite such a verbose formal description, the DES model may be executed by a relatively simple iterative algorithm consisting of three functions called one by one (Figure 4.4.).

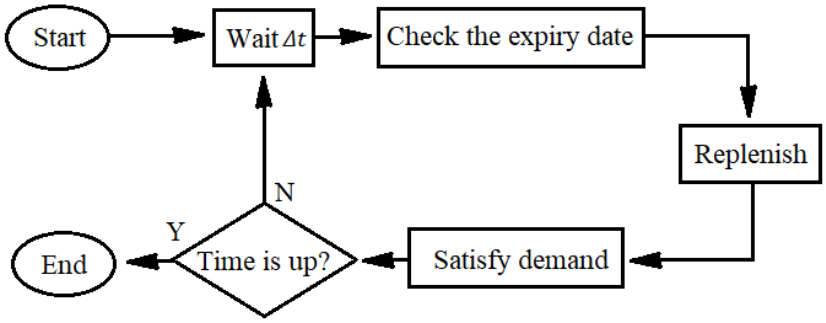

Figure 4.4. The order of operations

Firstly, the expiry date is checked and perished products are removed:

```
for all elements in E_t^p do
    if e_i^p ≤ 0 do
        E_t^p.pop(i)
    end if
end for
```

```
if r^p>l_t^p and status = False do
    status ← True
    o_t^p ← Q^p
end if
```

Secondly, replenished products are added to the stock:

```
if status = True and o_{t−l}^p ≠ 0 do
     S_t^p.append(Q^p)
     E_t^p.append(e^p)
     status ← False
end if
```

After that, demand is satisfied:

```
if I_t^p ≥ d_t^p do
    i ← 1
    while d_t^p > 0 do
        tmp ← s_i
        s_i ← s_i − d_t^p
        d_t^p ← d_t^p − tmp
        if s_i <0 do
            s_i ← 0
        end if
        i ← i+1
    end while
else do
     Backorder_t^p = d_t^p − I_t^p
     S_t^p ← ()
end if
if r^p>I_t^p and status = False do
     status ← True
     o_t^p ← Q^p
end if
```

4.2. Stochastic multiproduct production-inventory system with lost-sales and Markov-modulated demand

The second DES model under consideration is referred to in the following chapters simply as the model 2. It also encompasses the recent work (Jackson, 2019b) adopting features from related papers. Notably, the model adopts Markov-modulated demand described by Beyer *et al*. (2010), in which the demand process is associated with an underlying Markov chain. Such that

environmental parameters are represented by demand states of a Markov process. As the result, the production-inventory model can be characterized as multiproduct and stochastic with lost-sales and Markov-modulated demand. Appendix 2 contains a Python 3.7 implementation of the DES model.

4.2.1. Material flow

The production-inventory system under consideration operates with a sequence of products $P = (1, \dots, n)_{n \in N^+}$ and has a limited total storage capacity I_{max}. The model considers only those moments of time, in which the system parameters change (discrete events of particular interest happen). Timings of such events are given as a sequence $T = (t_1, \dots, t_n)_{n \in N^+}$. In this regard, the value of $t_n - t_1$ is the planning horizon. For each product inventory on hand at time t is given by a corresponding sequence $S_t = \left(S_1, \dots, S_p\right)_{p \in P}$. Each product is produced on its own machine (or by independent production facility in more general case) with production rates $R_t = \left(R_1, \dots, R_p\right)_{p \in P}$measured in items per unit time. Each R_p is a random variable under a known distribution $F_R = Prob\left(r_p \leqslant R_p\right)$. Demand is composite and includes all available products $D_t = \left(D_1, \dots, D_p\right)_{p \in P}$. Such that demand for each particular product is also a random variable under the distribution that depends on the environmental state. Considering a set of demand states $\Pi = (1, \dots, q)_{q \in N^+}$, the demand state i_t is a Markov chain over Π with the corresponding transition matrix $M_p=\{m_{ij}\}$. Such that $0 \leqslant m_{ij} \leqslant 1$, and $\sum m_{ij} = 1 \; \forall i, j \in \Pi$. Under such assumptions demand D_t depends only on instance of time t and the demand state i_t. The special case with three states environmental states is shown in the Figure 4.5.

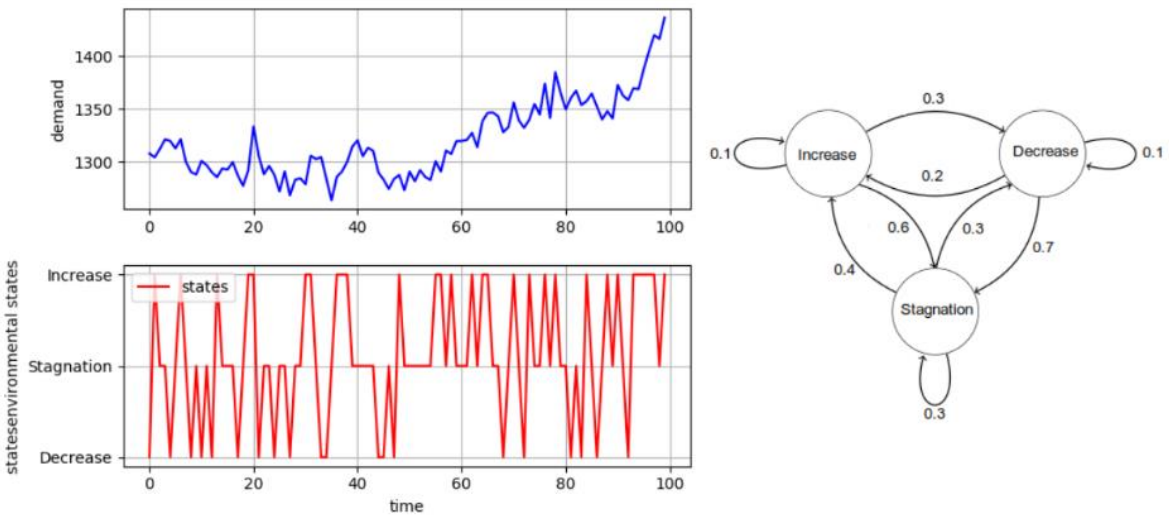

Figure. 4.5. The Markovian demand for market with 3 environmental states (increase, stagnation, decrease)

We denote demand inter-arrival time as $a_i = t_i - t_{i-1} = \Delta t$, which is a value of a random variable A under a specified continuous distribution $F_a = Prob(a \leqslant A)$. The demand function *f(.)*is defined in order to fulfil arising demands depending on the available inventory capacity (4.1) and supply function *g(.)* that replenishes the inventory with newly produced goods (4.12). Besides that, for each product *p* variables Pro_p, $Sales_p$, $Lostsales_p$ and $Over_p$ are introduced in order to keep track on production, sales, lost-sales and overflows for later calculations of net-profit.

$$f(S_t, D_t) = \begin{cases} S_t \leftarrow (S_t - D_t)\ \&\ Sales_p \leftarrow Sales_p + D_{t,p}, if\, D_{t,p} \leqslant S_{t,p} \\ else, Lostsales_p \leftarrow Lostsales_p + D_{t,p} \forall p \in P \end{cases} \quad (4.11)$$

$$g(S_t, R_t) = \begin{cases} S_t \leftarrow (S_t + \Delta t R_t)\ \&\ Pro_p \leftarrow Pro_p + \Delta t R_{t,p}, if\ S_t + \Delta t R_t \leqslant S_{max} \\ else, S_t \leftarrow S_{max}\ \&\ Over_p \leftarrow Over_p + S_{max} - S_t + \Delta t R_t\ \forall p \in P \end{cases} \quad (4.12)$$

It is worth to note that *f(.)* works according to "all or nothing" principle, namely, either the composite demand is fulfilled completely or considered completely lost. Such settings are quite common for consolidation problems in retail.

The system operates according to quite a straightforward control rule, namely there is a sequence of production statuses $U_t = (U_1, \ldots, U_p)_{p \in P}$, so as $U_{t,p} \in \{True, False\}\ \ \forall p \in P$. Besides that, for each product there is a pair of control parameters (G^p, H^p), which prescribe the

inventory level that must be reached to stop and to resume production process. Each resumption of production is associated with setup costs and, for this reason, a variable Set_p is declared to keep track of it. In order to switch the statuses and count the corresponding setups, the following function is denoted:

$$u(S_t, U_t) = \begin{cases} U_{t,p} \leftarrow False, if\ U_{t,p} = True\ and\ S_{t,p} > G^p \\ U_{t,p} \leftarrow True,\ Set_p \leftarrow Set_p + 1, if\ U_{t,p} = False\ and\ S_{t,p} < H^p \end{cases} \quad (4.13)$$

The Figure 4.6. illustrates the inventory dynamics under such control rules.

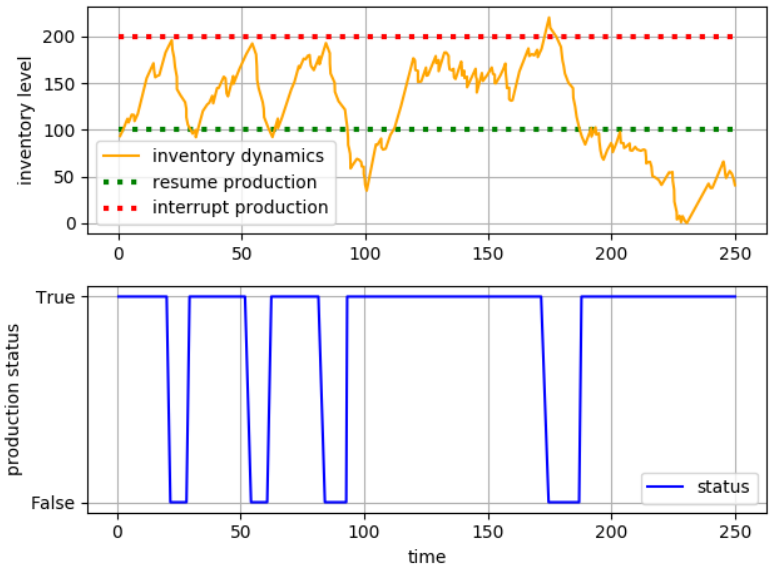

Figure 4.6. The logic behind control

Obeying the following order of operations: 1) produce 2) fulfil the composite demand 3) update production status, the equation $(S_{t+1}, U_{t+1}) = u(f(g(S_t, R_t), D_t), U_t)$ simulates the dynamics of the physical flow.

4.2.2. Monetary flow

For each unit of product p there are corresponding production cost c^p, handling cost h^p, setup cost o^p and fees associated with lost-sales b^p and overflows ω^p. As the result total cost can be calculated as follows:

$$TC^p = c^p \sum Pro_t^p + h^p \sum S_t^p \Delta t + b^p \sum Lostsales_t^p + \omega^p \sum Over_t^p + o^p \sum Set_t^p \quad (4.14)$$

Considering that each unit of product p is sold by a constant $price^p$, the net profit associated with this product is $Profit^p = price^p \sum Sales^p - TC^p$. Thus, total net profit of the production-inventory system is $\sum_{p=1}^{n} Profit^p$, which is the simulation output of special interest.

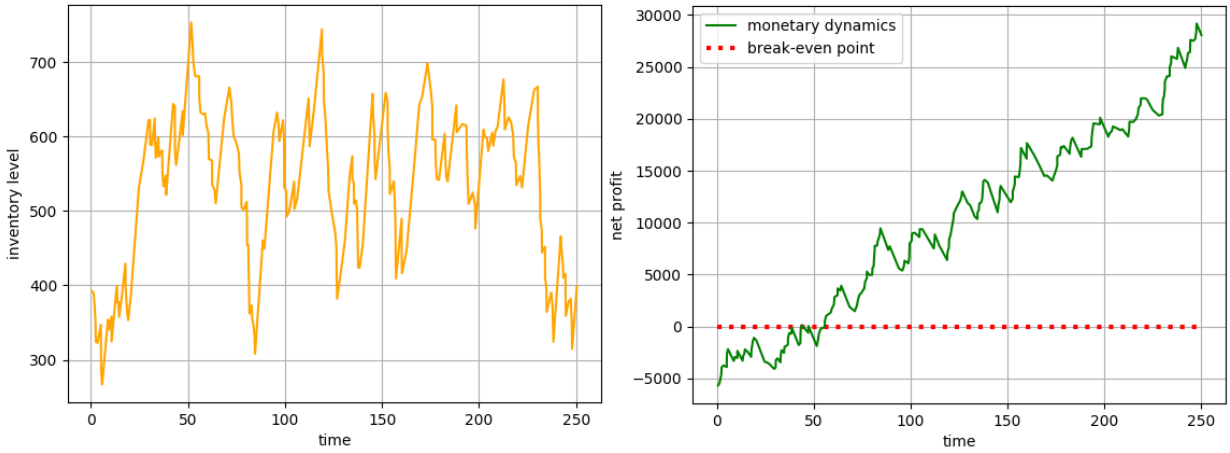

Figure 4.7. The dynamics of physical and monetary flows

4.2.3. Algorithmic implementation

Precisely like in the case of the model 1, the DES model may be executed by an iterative algorithm consisting of three functions executed sequentially (Figure 4.8.).

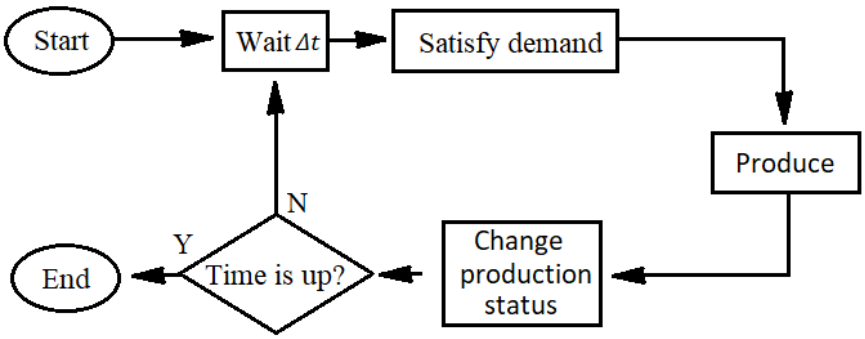

Figure 4.8. The order of operations

Firstly, the composite demand is satisfied:

```
if all elements in S_t ≥ all elements in D_t do
    S_t ←(S_t - D_t)and Sales_p ← D_t,p  ∀p ∈ P
else do
    Lostsales_p ← Lostsales_p + D_t,p  ∀p ∈ P
end if
```

Secondly, inventory level is replenished by the produced products:

```
if S_t + ΔtR_t ≤ S_max do
    S_t ← (S_t + ΔtR_t) and Pro_p ← Pro_p + ΔtR_t,p  ∀p ∈ P
else do
    S_t ← S_max and Over_p ← Over_p + (S_max − S_t,p + ΔtR_t,p)  ∀p ∈ P
end if
```

Lastly, control parameters are changed, according to the inventory level:

```
for all p in P do
    if U_t,p = True and S_t,p > G^p do
        U_t,p ← False
    elseif U_t,p = False and S_t,p < H^p do
        U_t,p ← False and Set_p ← Set_p + 1
    end if
end for
```

4.3. Defining the number of replications

Due to the fact that several input variables of both models under consideration are random, namely, in the model 1 $D\sim N(\mu, \sigma^2)$, $L\sim N(\mu, \sigma^2)$ and $A\sim Exp(\lambda)$, and in the model 2 $D\sim N(\mu, \sigma^2)$, $R\sim N(\mu, \sigma^2)$ and $A\sim Exp(\lambda)$, the output (net profit) is also a random variable under some distribution. Thus, in order to decide, how many replications are enough, the method based on confidence intervals (CI) proposed by Byrne (2013) is applied:

$$\eta = \left(\frac{z_{\alpha/2}}{\zeta} CV\right)^2, \tag{4.15}$$

where η is a minimum number of replications to achieve desired confidence interval width ζ (expressed as a multiplier by the mean) for model with a coefficient of variation CV.

A script runs every DES model 10000 times with random input parameters generated in feasible range. Such that each replication has the same inputs and covers exactly 120 modelling days (a season). After that the generated sample is used to calculate a coefficient of variation and test normality applying the AD test (Table 4.1). Appendix 3 contains the source-code of the described procedure.

Table 4.1. Anderson–Darling normality test

Model	Statistic	Critical value	Significance level
Model 1	0.716	0.787	0.05
Model 2	0.648	0.787	0.05

With this approach the null hypothesis that a sample is taken from a normally distributed population is tested. Since the calculated statistic is smaller than the critical value for both samples, the null hypothesis that the data is drawn from normal distribution can be accepted for the corresponding significance level. The empirical distribution and box-plot of the simulation output from different replications with the same input parameters is demonstrated for the model 1 (Figure 4.9) and for the model 2 (Figure 4.10).

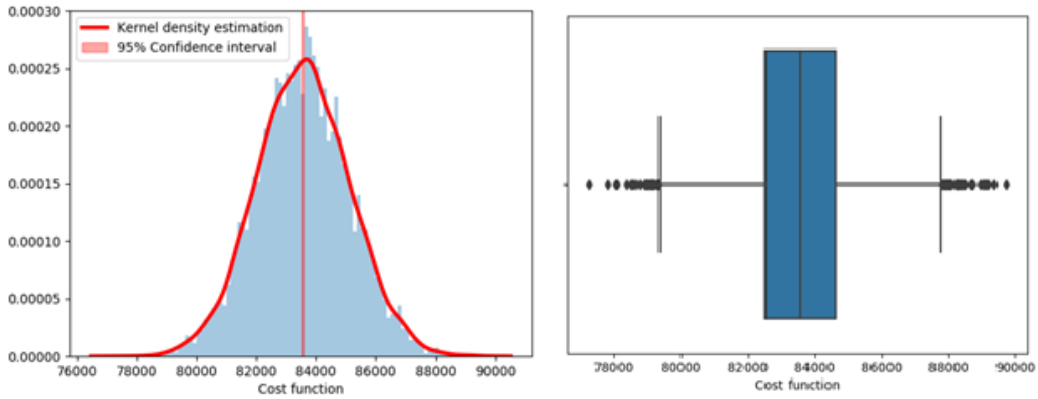

Figure 4.9. The empirical distribution and box-plot of the output variable (model 1)

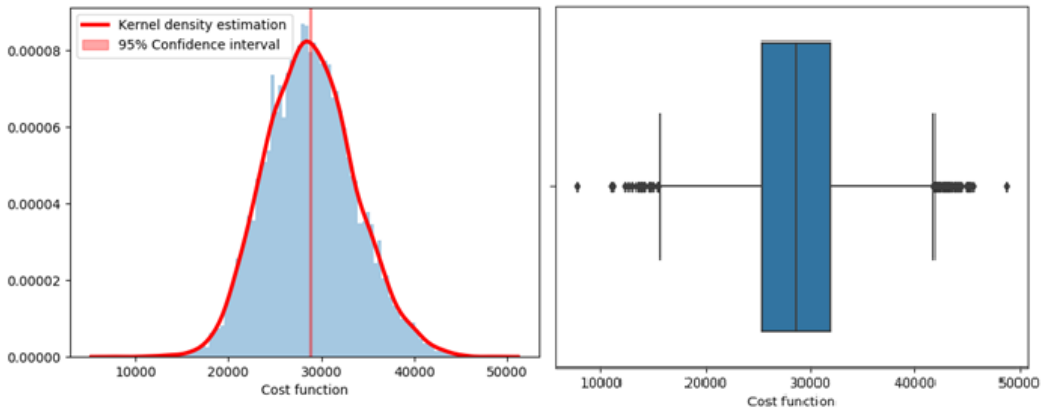

Figure 4.10. The empirical distribution and box-plot of the output variable (model 2)

Assuming the confidence level of 95% and corresponding $z_{\alpha/2} = 1.96$. In the case of model 1 the coefficient of variation can be computed using sample mean and variance *CV* = 83721.8/558147.2 = 0.15. It is decided to work with the CI of length 4186.1 (5% of the mean) running each simulation 35 times to take the average output. For the model 2 all the calculations can be done in the same manner. Namely, *CV* = 29357.3/172688.4 = 0.17, so for the CI of length 1467.9 (5% of the mean) each simulation must be replicated 44 times. The number of replications and the values used for intermediate calculations are shown in the Table 4.2.

Table 4.2. Defining the number of replications

Model	$z_{\alpha/2}$	Confidence level	Mean	Variance	CI	CV	Number of replications
Model 1	1.96	95%	83721.8	558147.2	4186.1	0.15	35
Model 2	1.96	95%	29357.3	172688.4	1467.9	0.17	44

5. NEUROEVOLUTIONARY FRAMEWORK FOR AUTOMATED METAMODELING

As it is highlighted in the chapter 3, neuroevolutionary framework comprises two vital components, namely neural and evolutionary. According to the state-of-the-art practices and considerations, in the proposed framework GA is chosen as the evolutionary component and multilayer perceptron (MLP) as the neural one. Such that GA orchestrates evolutionary morphism of neural architecture and hyperparameters in order to maximize a performance evaluation metrics. In this regard, the capability of the neuroevolutionary framework is determined by three factors:

- Search space;
- Search strategy;
- Performance estimation strategy.

The search space isolates the manifold of attainable neural architectures that can be derived in principle. It should be pointed out that the search space is defined by the researcher in accordance with prior knowledge about the problem and properties of particular architectures and, thus, it remains on his conscience. On the first hand, smaller search space can significantly accelerate and simplify the search. On the other hand, a bias inserted by the human-expert can prevent finding of the optimal architecture and hyperparameters that lay beyond the allowed search space. Search strategy is associated with the neural component and, in fact, defines from what “spare parts” a candidate metamodel can be assembled.

The search strategy postulates the exploration of the search space. Due to the fact that the search space is often extremely large, the classical “exploration versus exploitation” dilemma arises. On the first hand, it is highly optative to find a nearly optimal metamodel quickly. On the other hand, excessively fast search enhances the chances of premature convergence to local optima or suboptimal region, which is strongly undesirable and must be avoided. It is worth to note that the search strategy is related to the evolutionary component, and in GA case, exploration-exploitation trade-off can be adjusted by varying mutation and crossover probability as well as the value of the tournament size.

Performance estimation strategy refers to the approach behind estimation of metamodel predictive capabilities. It is a joint part shared by neural and evolutionary components. Namely, for neural component performance estimation strategy corresponds to the metric associated with accuracy on the testing subset, and for the evolutionary one it corresponds to the fitness function.

5.1. Neural component

The framework for automated metamodeling of IC systems heavily relies on the universal approximation theorem proved by Cybenko (1989) and later extended by Hornik (1991) and Hanin (2018), which states that any feedforward fully connected ANN with at least one hidden layer containing a finite number of neurons and non-linear activation function can approximate simulation model distinguished for nonlinear relation between input and output variables and presence of stochastic noise.

Since MLP is the most general feed-forward ANN, and other feedforward networks can be considered as its specific case, MLP is used as a baseline for a metamodel. MLP apparently derives its computational efficiency, flexibility and robustness from extremely distributed structure and ability to generalize. Generalization refers to the ability of producing correct output for inputs that have not been encountered during the learning phase. As the result, these information processing capabilities allow MLP to derive nearly-optimal approximate solutions to complex multidimensional problems in tolerable elapsed time. Thus, MLP-based metamodels of IC systems will be distinguished by the following properties:

- Nonlinearity. MLP that contains nonlinear activation functions is itself nonlinear. It is crucially important, if a model of IC system is distinguished by nonlinearity;
- Input-output mapping. Metamodeling can be considered as a form of supervised learning paradigm. Supervised learning involves weights modification in accordance with a dataset of labeled observations. Such that, each single observation consists of input vector and desired output. The weights are modified to minimize the difference (based on performance-evaluation metric) between the desired output and the actual one. The training phase is repeated for all observations in the dataset, until the MLP reaches the plateau, where there are no further notable changes in the weights. In this regard, it should be pointed out that the MLP learns from the training data by mimicking an input-output mapping of an original function (Geman *et al*., 1992);
- Adaptivity. MLP has natural capability to adapt the weights in accordance with changes in a nonstationary environment. Therefore, MLP-based metamodel tailored to reproduce input-output map of a particular IC system can be effortlessly retrained to deal with minor

changes in the input parameters (Grossberg, 1988). This property seriously enhances both flexibility and robustness of the neuroevolutionary framework;

- Fault tolerance. Thanks to the fact that information is massively distributed across the MLP network, small or moderate damage of hardware components underneath the MLP will cause graceful decrease in performance rather than catastrophic degradation (Kerlirzin and Vallet, 1993).

In the neuroevolutionary framework the search space defines the plethora of attainable structures, thus, it should be defined within the neural component. Since this research is rather focus on the proof of concept and not on the development of the all-embracing and catchall framework, the search space includes only standard and the most vital components of MLP, namely:

- Depth. How many hidden layers does the MLP include;
- Width. How many neurons are in each single layer;
- AF. Which nonlinear activation function is implemented;
- Optimizer;
- Learning rate. How fast and accurate is the learning procedure.

Due to the fact that in any practical context a total number of neurons and weights in the MLP is limited, the choice between deep and wide learning can be hard to make. Wide architectures are good at memorization of desirable input-output maps, however, they perform poorly at generalization. Thus, excessively wide, but shallow MLP-based metamodel can eventually memorize the corresponding output value for every observation in the training dataset. Such a metamodel would be of little use, because it will not be able to demonstrate desirable performance at predicting output values for inputs that have not been encountered during the learning phase. That is where the pivotal advantage of having multiple hidden layers arises. Namely, the deep MLP-based metamodel can firstly construct complex intermediate features and use them as a base to predict the output (Pandey and Dukkipati, 2014). For this reason, the neuroevolutionary framework searches for the proper balance between depth and width and between memorization and generalization respectively.

Learning with MLP in general and its metamodeling manifestation in particular can be viewed as minimization of the prediction error. Thus, learning is the search for the global minimum on the

loss surface. The distinguishing feature of such optimization is non-convexity and ill-conditioning, such that the loss surface contains saddle points, flat areas and has multiple optima. These facts make the optimizer an important component of the MLP-based metamodel. The Table 5.1. contains state-of-the-art optimizers incorporated into the search space and their concise description.

Table 5.1. Optimizers available in the search space

№	Abbreviation	Concise description
1	SGD	Classical stochastic gradient descent optimizer (Bottou, 2010).
2	Adagrad	Sub-gradient optimizer with adaptive learning rate (Duchi *et al.*, 2011).
3	Adadelta	Robust extension of Adagard that adjusts learning rate in accordance with a moving window of gradient updates (Zeiler, 2012).
4	Adam	Extension of SGD to compute individual adaptive learning rates based on moments' estimates of the gradient (Kingma and Ba, 2014).
5	Adamax	Extended Adam that uses the infinity norm (Zeng *et al.*, 2016).
6	Nadam	Adam optimizer equipped with Nesterov momentum (Dozat, 2016).

It is worth to emphasize that adaptivity of the weight is proportional to the learning rate. The learning rate affects both speed and accuracy of a learning procedure. Such that large values lead to rapid and uncontrolled wandering across the error surface with a significant chance to simply bypass the optimum. On the other hand, small values of the learning rate can require unaffordable amount of time or computational resources. In this regard, the learning rate should be included into the set of hyperparameters adjustable by the neuroevolutionary framework.

AF is also an important component that converts an input signal of a node into an output, which is treated as an input in the following layer of the MLP. Different activation functions possess different properties and significantly affect the learning outcome. The Table 5.2. contains nonlinear activation functions within the search space and their concise description.

Table 5.2. Activation functions available in the search space

№	Abbreviation	Full name and concise description
1	ELU	Exponential linear unit can have negative values, which shifts the mean of the activation functions close to zero. As the result, the faster learning and convergence can be achieved (Clevert *et al*., 2015).
2	SELU	Scaled exponential linear unit is distinguished by self-normalization, which allows to successfully treat the vanishing gradient problem (Zhang and Li, 2018).
3	ReLU	Rectified linear unit is both efficient and simple to compute, since it is linear for values greater than zero and equal to zero otherwise (Hara *et al*., 2015).
4	Hard_Sigmoid	Approximation of sigmoid activation function that is significantly faster to compute than original one (Chen *et al*., 2006).
5	Softplus	Unlike ReLU, Softplus activation function is smooth and differentiable near zero (Zheng *et al*., 2015).
6	Tanh	Hyperbolic tangent activation function is a sigmoidal function that outputs values in the range from -1 to 1. Therefore, strongly negative inputs are mapped to negative outputs (Zamanlooy and Mirhassani, 2013).

Additionally, in order to test the capability of the MLP-based metamodel to predict output of the IC simulation for such inputs that have not been encountered during the training phase, the framework incorporates k-fold cross-validation. As it is mentioned by Cawley and Talbot (2010), it is a common approach in machine learning to spot and prevent such frequent problems as overfitting.

5.2. Evolutionary component

GA defines the search strategy of the neuroevolutionary framework. Mimicking Darwinian natural selection and “survival of the fittest” principle, GA appears to be the driving force behind the neuroevolution, namely, it samples, crosses over, mutates and resamples the population of highly fit MLP-based metamodels in order to form a new even fitter population. As the result, instead searching for nearly optimal topologies by exhaustively trying every feasible combination, GA improves the solution step by step from the partially best topologies from the previous samplings.

The implemented GA is distinguished by Gray-code chromosome representation, uniform crossover and tournament selection. This recipe is already justified in the chapter 3, nevertheless, the most crucial advantages behind its components will be highlighted once again in the context of neuroevolution. In order to implement GA in accordance with this specification, the following initial parameters are required (Goldberg and Holland, 1988):

- Population size (N_{pop}) – the number of individuals in each generation;
- Crossover rate (P_c) – the probability of executing a crossover operator;
- Mixing ratio (P_u) – the probability for each attribute to be exchanged;
- Mutation rate (P_m) – the probability of executing a mutation operator;
- Tournament size (t_size).

Since the choice of an appropriate chromosome representation for the particular problem domain can accelerate and simplify the search drastically, the role of chromosome encoding cannot be disparaged. Because of space-efficiency, binary representation is an apparent choice, however, chromosome encodings vary noticeably even among different binary schemes. Relying on the study conducted by Caruana and Schaffer (1988), the choice falls on the reflected Gray code for its guarantee of gradualism, for the reason that exactly one bit has to be flipped in order to reach the nearest value, which helps to overcome problems related to large Hamming clefts. Figure 5.1. demonstrates the implemented encoding mechanism.

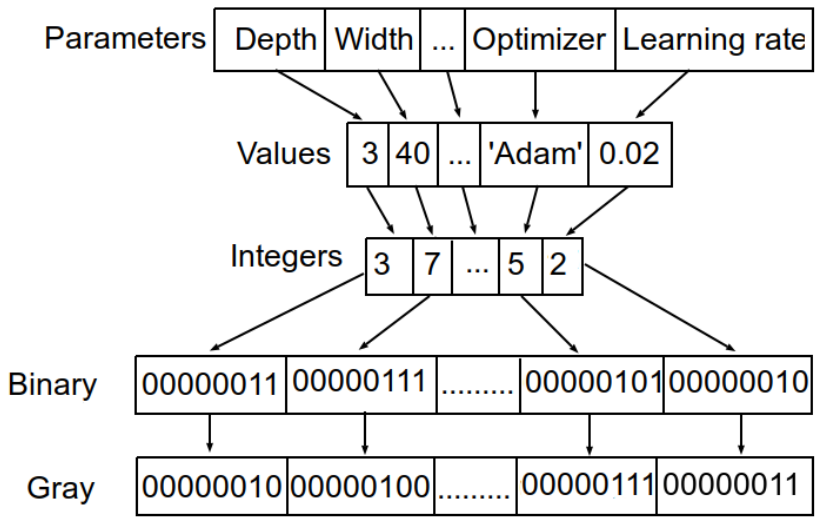

Figure 5.1. Components of the MLP are encoded into the Gray-code representation

It is tremendously important to take into account that both crossover and mutation operators must take into consideration the chosen chromosome encoding. By incorporating the uniform crossover into the framework, a serious drawback of more traditional crossovers can be compensated. Namely, since the number of crossover points is not constant, the decision of flipping is made independently for every single bit in chromosome. Thus, the crossover operator will not tend to split up the fit building blocks. Besides that, the uniform crossover is an efficient way to avoid the premature convergence (Michalewicz, 1996). Lastly, according to an empirical study, the uniform crossover is exploratory-shifted and provides more complete search by maintaining the exchange of valuable information (Williams and Crossley, 1998). In the uniform crossover bits in the chromosome are compared between two parents and swapped with the fixed mixing ratio P_u, according to the following algorithm:

```
Pu ← probability of swapping values
v⃗ ← first vector ⟨v1, v2, …, vn⟩ to be crossed over
w⃗ ← second vector ⟨w1, w2, …, wn⟩ to be crossed over
for i in range from 1 to the length of the vector do
   if Pu ≥ random number in range (0.0, 1.0) do
     swap the values of vi and wi
   end if
end for
return v⃗ and w⃗
```

In addition, GA requires a mutation operator to perform the optimization, otherwise a premature convergence of a population is inevitable. Thus, in order to diversify the population of MLP-based metamodels some bits should be flipped with the probability P_m along the chromosome:

```
P_m ← probability of flipping the bit
v⃗ ← vector
for i in range from 1 to length of v⃗ do
  if P_m ≥ random number in range (0.0, 1.0) do
    v_i ← flipped bit
  end if
end for
return v⃗
```

As it is emphasized in the chapter 3, in contemporary neuroevolutionary approaches tournament selection is a standard choice, because it has several significant benefits over alternative selection techniques. More specifically, it is both simple and efficient to implement. Besides that, as it is concluded by Miller and Goldberg (1995), tournament selection is a robust mechanism for working with noisy fitness functions. Besides, it works with parallel architectures and can be fine-tuned by varying the parameter *t_size*.

Tournament selection runs several "tournaments" among *t_size* individuals randomly selected from the population. The fittest individual in each conducted tournament is taken for the following mutation and crossover. The logic behind tournament selection can be implemented as follows:

```
P ← population
t_size ← tournament size, t_size ≥ 2
Best ← randomly picked individual from P with replacement
for i in range from 2 to t_size do
   Next ← randomly picked individual from P
   if Fitness(Next) > Fitness(Best) do
      Best ← Next
   end if
end for
return Best
```

5.3. Metamodeling via neuroevolution

Finally, armed with MLPs and GA automated metamodeling within the neuroevolutionary framework can be formalized. For this purpose, let's consider a nonlinear input-output map represented by the function $y = \varphi(x)$, where x stands for the vector of input parameters and y is the output. The simulation $\varphi(x)$ is a "black-box", however, the set of observations (experiments with this simulation) $\tau = \{x_i, y_i\}_{i=1}^{N}$ is obtained through Monte Carlo sampling. Since MLP-based metamodel can be built from such building blocks as layers, AFs, optimizers and respective hyperparameters, this information is encoded into one single chromosome $a \in A$ in form of Gray's binary code. After that GA orchestrates the morphism of MLP-based metamodels in an attempt to find such $a^* \in A$ that allows MLP-based metamodel $\Phi(a^*, x)$ to approximate the original simulation of an IC system close in Euclidean sense $\|\Phi(a, x) - \varphi(x)\| < \varepsilon$, where ε is a positive value, small enough in the context of the problem under consideration (Figure 5.2).

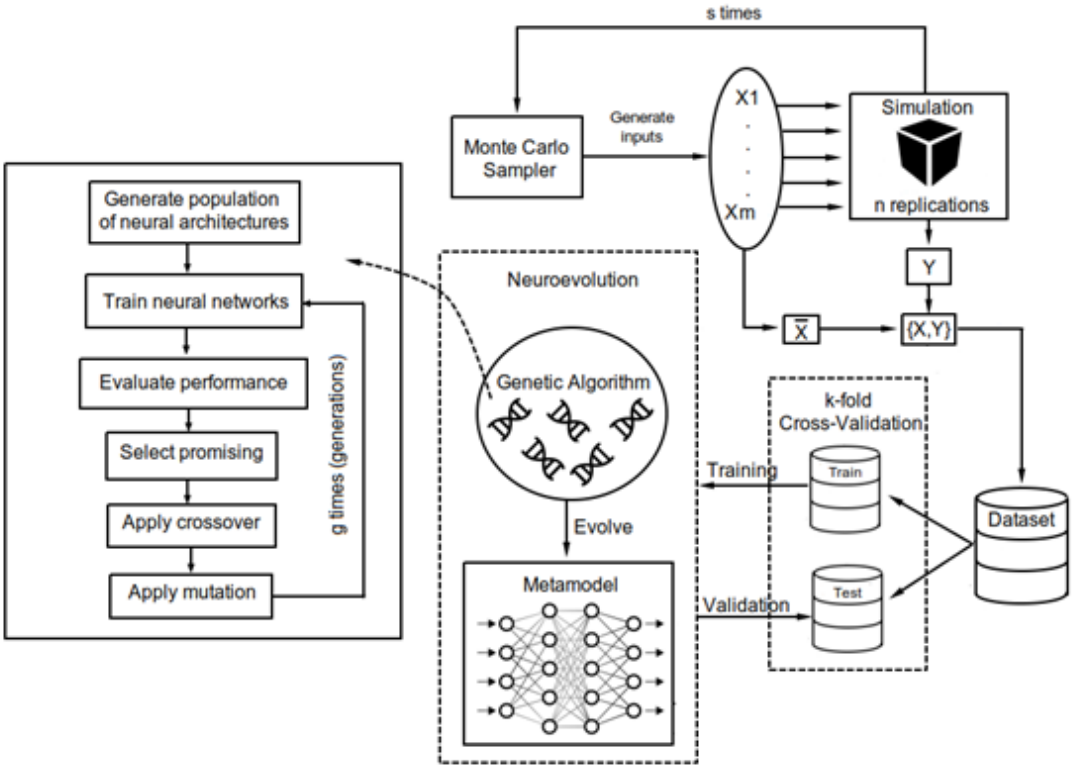

Figure. 5.2 The neuroevolutionary framework for automated metamodeling

The choice regarding the search space predominantly determines both difficulty and completeness of the NAS and HPO problems. In the conducted numerical studies, the search space includes the number of hidden layers from 0 to 8 (depth) and the number of neurons in each layer from 10 to 200 (width) with the discretization step of 1. Besides that, the search space encompasses learning rates from 0.01 to 0.3 with the discretization step of 0.01 and all the activation functions and optimizers mentioned in the section 5.1. So that GA is searching for such a structure in feasible region that results the MLP-based metamodel with the highest performance evaluation metric. Performance evaluation metric, in its turn, depends on, whether metamodeling is formulated as a regression or classification problem. In the classical case of metamodeling as a regression problem, the neuroevolutionary framework will seek to maximize the coefficient of determination (R^2). In the case of metamodeling as a binary classification problem, the harmonic mean of precision and recall, also known as F_1-*score* must be maximized (Van Rijsbergen, 2004). Besides that, the total number of trainable parameters (neurons and weights) is restricted to 20000, due to the limited computational budget. Such that, if a candidate MLP-based metamodel violates the constraints, the corresponding fitness function will take negative values. During the optimization search orchestrated by GA, such candidate solutions will have quite insignificant chance to pass to the next generation. Appendix 4 contains the source-code of the Monte Carlo sampler and Appendix 5 includes the source-code of the described neuroevolutionary framework.

5.4. No-free-lunch theorem

It is worth mentioning that both components within the neuroevolutionary framework are strongly associated with "No Free Lunch" theorem (NFLT). According to NFLT, there cannot exist an algorithm for solving all possible problems that is in average case superior to any competitive one. This statement is true for both optimization (Wolpert and Macready, 1997) and supervised learning (Wolpert, 1996). Therefore, in general case the question of whether the proposed neuroevolutionary approach is inferior or superior to any alternative has no sense. Besides that, unfortunately, there is no congenitally agreed-upon test problem catalogue of IC problems to test approaches in a concise way. In this regard, it is worth to emphasize unequivocally that the main advantage of the proposed framework is its versatility, flexibility, complete automatism, parallelism and high chance of finding nearly-optimal solutions in a finite time.

6. AUTOMATED METAMODELING VIA NEUROEVOLUTION

In this chapter the neuroevolutionary framework proposed in the chapter 5 is tested based on the model 1 and model 2 defined in the chapter 4. Firstly, the framework performs classical metamodeling formulated as a regression problem. Secondly, residuals are analyzed to determine the quality of MLP-based metamodels. Lastly, it is demonstrated, how metamodeling can be alternatively formulated as a binary classification problem.

6.1. Metamodeling as a regression problem

The procedure begins with generation of random inputs in the feasible range (Monte Carlo sampling). Simulation model of the IC system is replicated η consecutive times with these inputs, and output value is averaged. Storing both the vector of inputs and the average output the dataset τ is generated. As soon as τ is obtained, metamodeling can be naturally formulated as multivariate nonlinear regression, which can be performed using ANN in general and MLP in particular by considering simulation inputs as independent variables and simulation output (net profit) as dependent one (Landi *et al*., 2010). Thus, the learning process is reduced to minimization of the mean squared error (MSE) between actual simulation output *y* and one predicted by the MLP-based metamodel $\Phi(a, x)$ with the structure *a*:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(\Phi(a, x_i) - y_i)^2 \tag{6.1}$$

So, GA searches for such a structure $a^* \in A$ that result the MLP-based metamodel with the highest value of the coefficient of determination (R^2), which is a commonly used statistic for validation of a metamodel (Ryberg, 2013). The coefficient of determination can be interpreted as the share of the variance in the dependent variable that is predictable from the independent variables:

$$R^2 = 1 - \frac{\sum_i(\Phi(a,\ x_i) - y_i)^2}{\sum_i(\bar{y} - y_i)^2}, \tag{6.2}$$

where $\bar{y}$ is the mean of the observed data $\bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i$.

In the following numerical examples, the data is standardized prior to training using Z-score. AutoML procedure is driven by GA with *t_size* of 5, crossover probability of 0.35 and mutation

probability of 0.04. The evolution lasts 10 generations, and each generation is populated with 40 MLP-based metamodels with randomly initialized weights. It is worth to note that these hyperparameters are chosen according to the suggestions (Back *et al*., 2018) and with regard to the limited computational budget, however, their variation in the sane range has not notably affected either the performance of the best MLP or the convergence speed. Besides that, with regard to the limited computational resources and realism, the total number of trainable parameters (neurons and weights) is constrained to 20000. Such that, the candidate solution violating the constraint simply gets negative fitness and nearly zero chance to pass into the next generation.

Firstly, the 10-product case of the model 1 with 151 independent variables respectively is considered. The model 1 is replicated 35 consecutive times with randomly generated inputs, and the corresponding output is averaged. Storing both the vector of inputs and the average output as an observation, the dataset τ of 600 observations is generated. The dataset is split by training and test subsets using 10-fold cross validation. Such that every MLP-based metamodel is expected to predict the dependent variable for such vector of independent variables that have not been encountered during the training phase. In 10 generations the MLP-based metamodel with the average coefficient of determination of 0.81 has been evolved (Figure 6.1.).

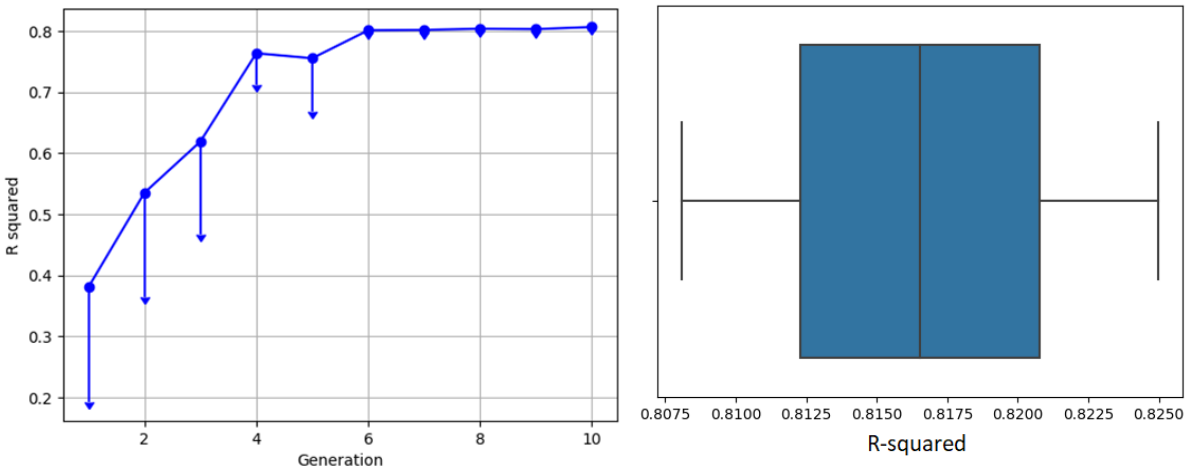

Figure. 6.1. The convergence path (vertical arrows represent the standard deviation of the fitness function within the generation) and box-plot of coefficients of determination calculated in 10-fold cross-validation (model 1)

The variance of performance estimation metric calculated based on all the cross-validation slices is insignificant, which means that the MLP-based metamodel was capable to generalize nonlinear relationships between the inputs and outputs.

Surprisingly, the fittest MLP-based metamodel is distinguished by relatively shallow architecture of 3 hidden layers with 60 neurons in each layer. Such that the model totally contains 16472 trainable parameters. The fittest metamodel incorporates ELU activation function and Adamax optimizer with the relatively low learning rate of 0.02**.** According to Kingma and Ba (2014), the infinite-order norm makes the Adamax algorithm both stable and fast to converge. Besides that, since ELU AFs cover negative values, the mean of the activations is shifted closer to zero, which also allows to achieve the faster learning and convergence. Figure 6.2. demonstrates the learning path.

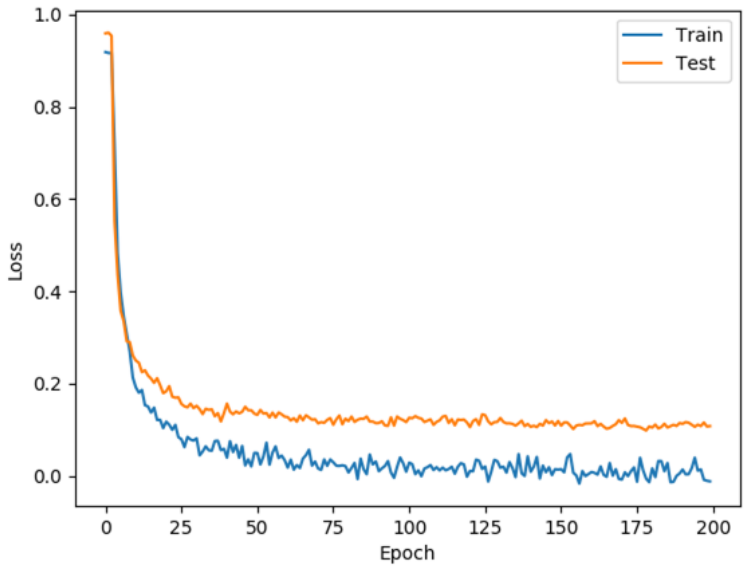

Figure 6.2. The learning process of the fittest metamodel (model 1)

Moreover, box-plot and violin plot clearly illustrate that the original sample generated by the simulation and the sample recreated using the trained metamodel belong to the same population (Figure 6.3.).

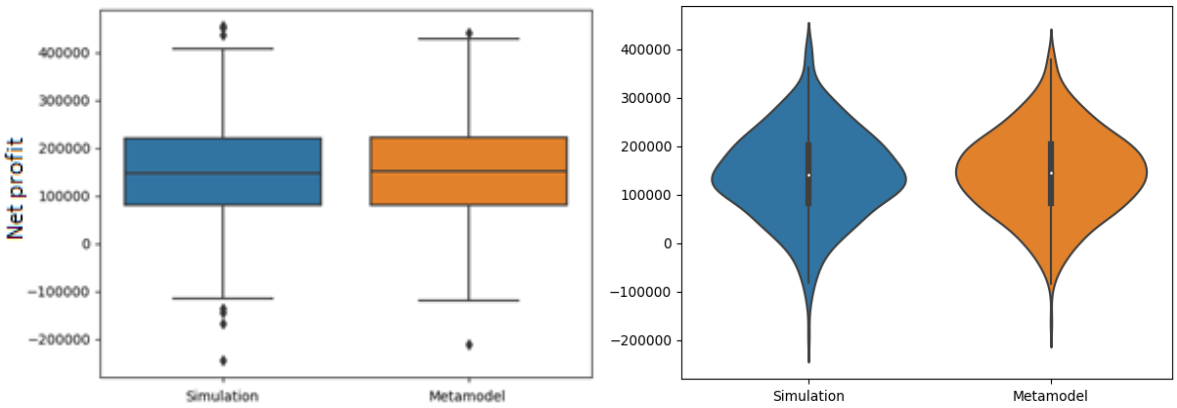

Figure 6.3. Boxplots and Violin plots of the simulated net profit and predicted by the metamodel (model 1)

Secondly, the 4-product case of the model 2 with 3 market states and 81 independent variables respectively is considered. The model 2 is replicated 44 times with randomly generated inputs, and the corresponding output is averaged. The vector of inputs and the corresponding output are stored making up the dataset of 600 observations. 10-fold cross validation is applied in training-test cycle in the same way as in the case of model 1. In 10 generations the MLP-based metamodel with the average coefficient of determination of 0.82 has been derived (Figure 6.4.).

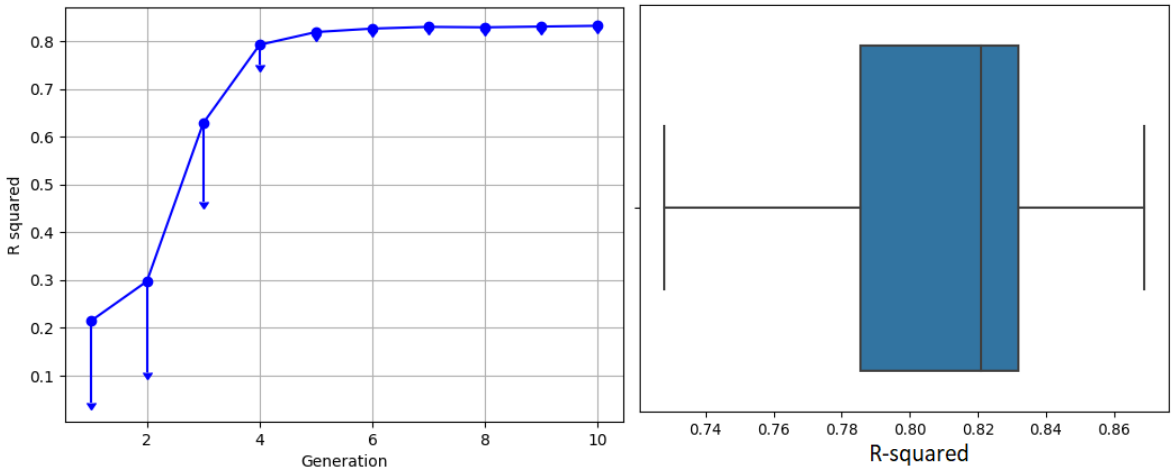

Figure. 6.4. The convergence path (vertical arrows represent the standard deviation of the fitness function within the generation) and box-plot of coefficients of determination calculated in 10-fold cross-validation (model 2)

In this case the variance of performance estimation metric calculated based on all the cross-validation slices is notably higher, nevertheless, even the lowest value of 0.73 can be sufficient depending on the context of the task.

As well as in the previous case the fittest solution is notable for its shallowness, namely, 3 hidden layers with 55 neurons in each layer and 10561 trainable parameters. The fittest MLP-based is built using ReLU AF and Nadam optimizer with learning rate of 0.07. The ReLU is characterized by one-sidedness and biological plausibility, which leads to good gradient propagation and, thus, reduced risk of vanishing gradient descent (Schmidt-Hieber, 2017). Additionally, it is worth to point out that this AF contains only such operations as addition, multiplication and comparison, which make it extremely efficient in terms of computations. Nadam optimizer takes advantage on Nesterov momentum, which is often useful, because even if the gradient descents in the right direction, the momentum term may overshoot (Sutskever *et al.*, 2013). The learning process of the fittest MLP-based metamodel with regard to training and test data is demonstrated (Figure 6.5.).

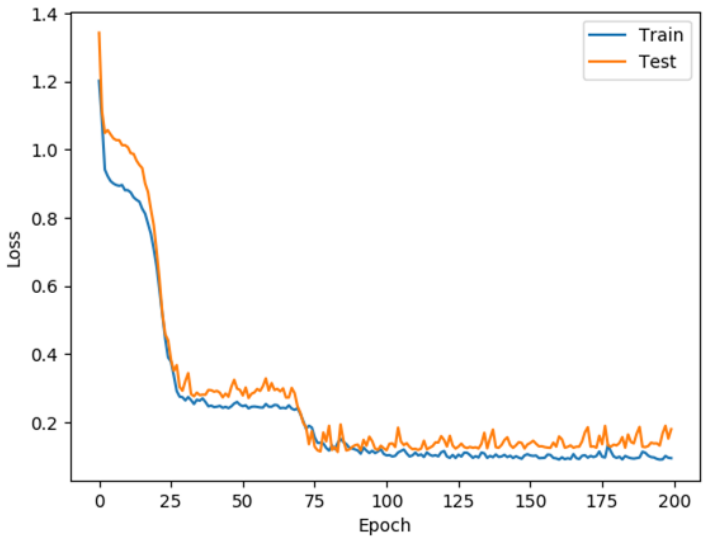

Figure 6.5. The learning process of the fittest metamodel (model 2)

The sample generated by the experiments with the simulation and the sample obtained using the trained MLP-based metamodel can be compared using box and violin plots (Figure 6.6.).

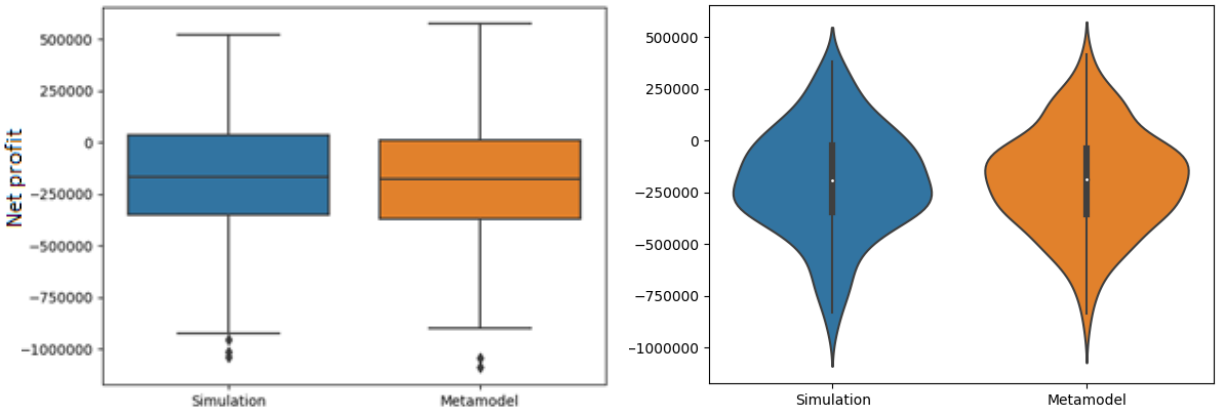

Figure 6.6. Boxplots and violin plots of the simulated net profit and predicted by the metamodel (model 2)

Several accuracy-related metrics regarding both metamodeling experiments are shown in the Table 6.1. Additionally, ordinary least squares linear regression is applied in order to demonstrate the presence of nonlinearity.

Table 6.1. Accuracy of the fittest MLPs compared to ordinary least squares

Models	Training sample			Test sample			Ordinary least squares		
	MSE	R^2	Adj. R^2	MSE	R^2	Adj. R^2	MSE	R^2	Adj. R^2
1	0.07	0.88	0.86	0.20	0.81	0.80	0.42	0.55	0.52
2	0.09	0.86	0.84	0.18	0.82	0.81	0.67	0.39	0.38

6.2. Residual analysis

Residual analysis provides a general approach to indicate the quality of the MLP-based metamodels and diagnose possible problems. In addition, if residuals contain a pattern, it indicates that the model is qualitatively inconsistent, incapable to capture some explanatory information and explain relations within the data (Pagan and Hall, 1983). The expected value of residuals for both metamodels is close to 0, namely -0.011 with regard to the model 1 and 0.013 for the model 2. It implies that both metamodels are not systematically biased toward over-prediction or under-prediction. Right after the distributions of residuals are plotted against theoretical standard normal (quantile-quantile plot), the possible issue related to heavy tails immediately catches the eye (Figure 6.7.).

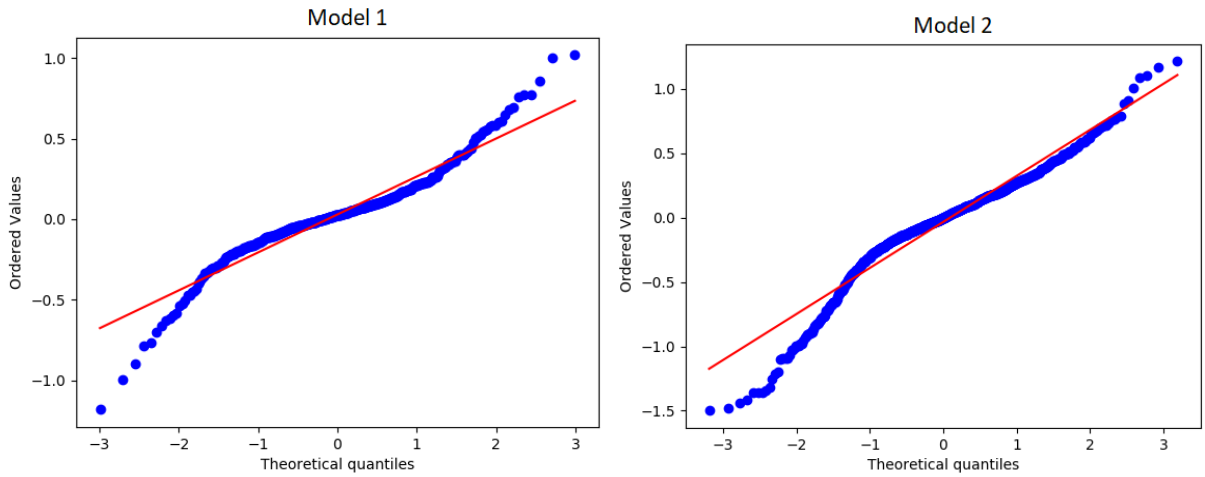

Figure 6.7. Quantile-quantile plot of residuals. Heavy tails are clearly visible.

Nevertheless, AD normality test is confidently passed in both cases (Table 6.2.). With AD approach the null hypothesis that a sample is taken from a normally distributed population is tested. Since the calculated statistic is smaller than the critical value for both samples, the null hypothesis that the data is drawn from normal distribution can be accepted for the significance level 0.05. The Figure 6.8. also demonstrates the empirical distribution of residuals.

Table 6.2. AD normality test for residuals

Model	Statistic	Critical value	Significance level
Model 1	0.523	0.787	0.05
Model 2	0.619	0.787	0.05

It is also worth to test, whether residuals are homoscedastic (their variance does not change over time). In practical sense, if residuals are heteroscedastic, this implies that the predictive power of the model is different for different sections of the data and, perhaps, it is worth thinking about dividing the dataset into two (or more) subsets in order to train two (or more) models, such that each model is specializing on the corresponding subset.

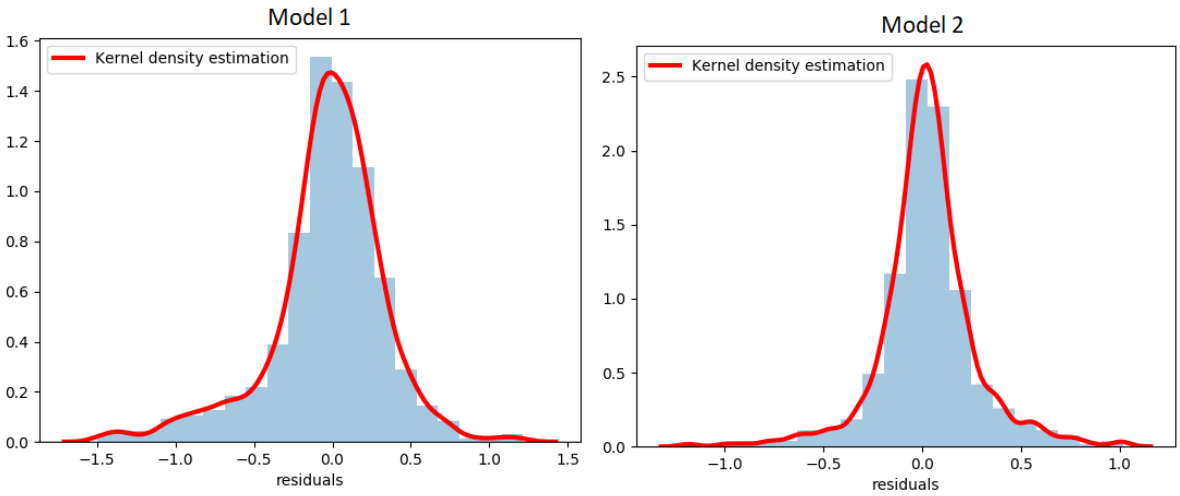

Figure 6.8. Empirical distributions of residuals

Fortunately, applying GQ test for heteroscedasticity, the null hypothesis that the two subsamples of the dataset have the same variance is not rejected for residuals of both metamodels (Table 6.3.).

Table 6.3. The parametric Goldfeld–Quandt test for heteroscedasticity

Model	GQ Statistic	Critical F-value	p-value	Significance level
Model 1	1.1	1.23	0.38	0.05
Model 2	0.97	1.30	0.43	0.05

6.3. Metamodeling as a classification problem

Alternatively, metamodeling can be formulated as a binary classification problem. Because of the universality of ANNs, the MLP-based metamodel with slight modifications can become a classifier. Classification is an instance of pattern recognition problem that has to do with developing of algorithm capable of grouping together data by particular criteria. Formally, in the context of metamodeling of IC systems, a classifier maps a vector of input parameters x_i to a class $label_i$ labeled based on the certain criteria.

Operating within the same neuroevolutionary framework on the same dataset τ that was used in the previous sections for regression, the problem can be almost effortlessly rephrased into a classification problem by labeling the data based some specific criteria of interest and, accordingly, changing the loss function. Indeed, taking into account the fact that the simulation output corresponds to the net profit, it is reasonable to take eventual profitability of IC system as this criterion of interest. Thus, the dataset τ can be labeled in the following way:

$$label_i = \begin{cases} 1, if\ y_i > 0 \\ else\ 0 \end{cases} \quad (6.3)$$

Such that all the inputs that make the IC system profitable are labeled as 1 and the rest as 0. As the result a new labeled dataset $\tilde{\tau} = \{x_i, label_i\}_{i=1}^{N}$ is derived and can be further used for binary classification. In these settings MLP can classify the input parameters as leading to profit or not by minimizing the binary cross-entropy function:

$$CE = -\frac{1}{N}\sum_{i=1}^{N} label_i \log_2\big(Prob(label_i)\big) + (1 - label_i)\log_2(1 - Prob(label_i)), \quad (6.4)$$

where $label_i$ is the label of i^{th} observation in the dataset (1 for profitable solutions and 0 for non-profitable). In this regard, *Prob(label)* is the predicted probability of the solution being profitable for all *N* observations in the dataset (Murphy, 2012).

In the context of the considered classification problems the confusion matrix (CM) takes the following form (Table 6.4.).

Table 6.4. The confusion matrix in the context of binary classification

Confusion matrix		Actual class	
		Profitable (1)	Non-profitable (0)
Predicted class	Profitable (1)	True positive (*tp*)	False positive (*fp*)
	Non-profitable (0)	False negative (*fn*)	True negative (*tn*)

CM is a specific table layout derived for visualization of the algorithmic performance in a supervised learning problem. Each row of the CM represents the instances in a predicted class, whereas columns represent the instances in an actual class. Calculating true positive (*tp*), false positive (*fp*), false negative (*fn*), true negative (*tn*) instances after classification, such pivotal performance estimation metrics as accuracy, precision and recall can be calculated:

$$accuracy = \frac{tp+tn}{tp+tn+fp+fn} \quad (6.5)$$

$$precision = \frac{tp}{tp+fp} \quad (6.6)$$

$$recall = \frac{tp}{tp+fn} \quad (6.7)$$

Under new assumptions GA within the neuroevolutionary framework orchestrates the network morphism and hyperparameter tuning searching for such a structure $a^* \in A$ that result the MLP-based classifier with the highest value of the harmonic mean of the precision and recall, also well-known as *F_1-score*:

$$F_1 = \frac{2*precision*recall}{precision+recall} \tag{6.8}$$

In the context of binary classification, the *F_1-score* measures the accuracy of the MLP-based classifier considering both the precision and recall.

Firstly, the dataset that contains 600 observations of 10-product case of the model 1 is labeled. As well as in the regression case, the dataset is split by training and test subsets using 10-fold cross validation. Such that every MLP-based classifier is expected to predict the label for such vector of features that have not been encountered during the training phase. In 10 generations of neuroevolutionary morphism the MLP-based classifier with the average *F_1-score* of 0.92 has been derived (Figure 6.9.).

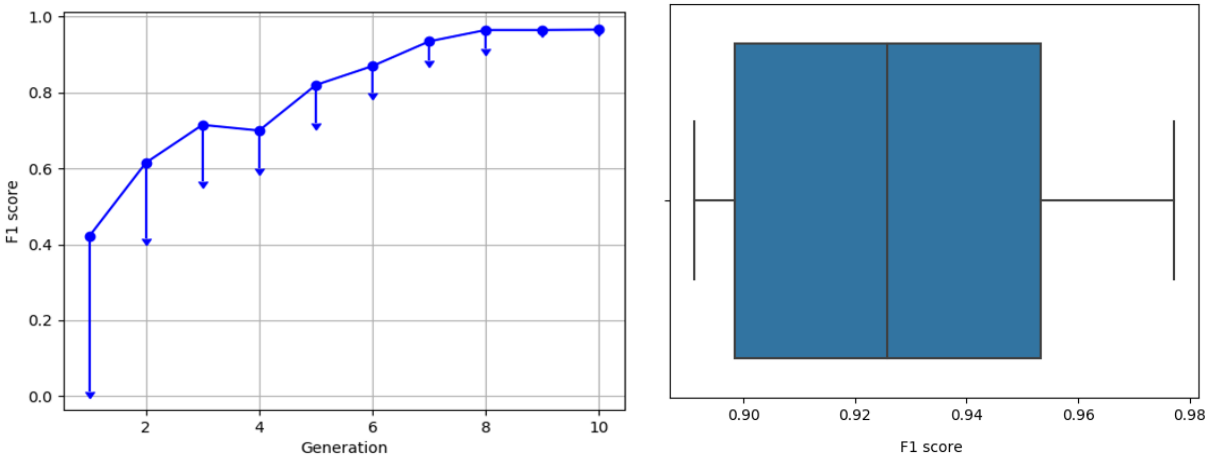

Figure 6.9. The convergence path (vertical arrows represent the standard deviation of the fitness function within the generation) and box-plot of *F_1-score* calculated using 10-fold cross-validation (model 1)

The variance of *F_1-score* calculated based on all the cross-validation slices is minor (0.89 in the worst case), which means that the MLP-based classifier was capable to generalize nonlinear relationships between the vector of features and the corresponding label. It is also worth to note

that, since, according to several backstage experiments, the MLPs cope significantly better with classification task, than with regression, it was decided to reduce the number of epochs to 60, which, indeed, does not affect the grist of the learning process (Figure 6.10).

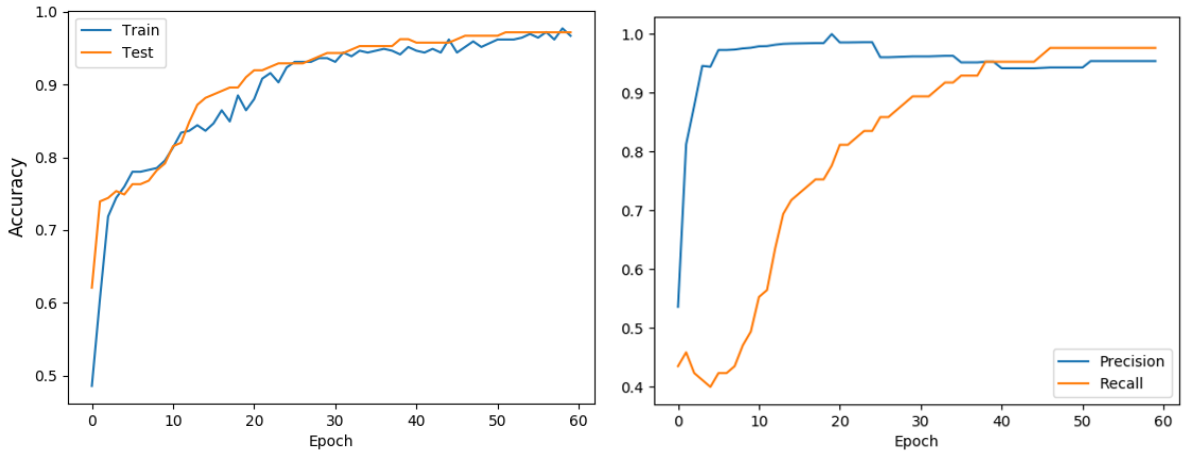

Figure 6.10. Accuracy improves during the learning process (model 1)

The fittest MLP-based classifier, all of a sudden is even more compact, than the corresponding metamodel trained to solve regression problem. Namely, the architecture includes 2 hidden layers with 60 neurons in each layer. As the result, the model totally has 12872 trainable parameters. Table 6.5. shows the obtained CM.

Table 6.5. The confusion matrix of the model 1

Confusion matrix (Model 1)		Actual class	
		Profitable	Non-profitable
Predicted class	Profitable	341	14
	Non-profitable	6	239

The fittest classifier uses SELU as AF and Adamax optimizer with the learning rate of 0.04**.** Distinguishing advantages of Adamax optimizer related to the infinity norm are already mentioned. On the other hand, evolutionary success of SELU, as it is mentioned by Zhang and Li (2018), can be associated with its excellent quality of self-normalization, which extirpates vanishing gradients. Figure 6.11. demonstrates the learning path.

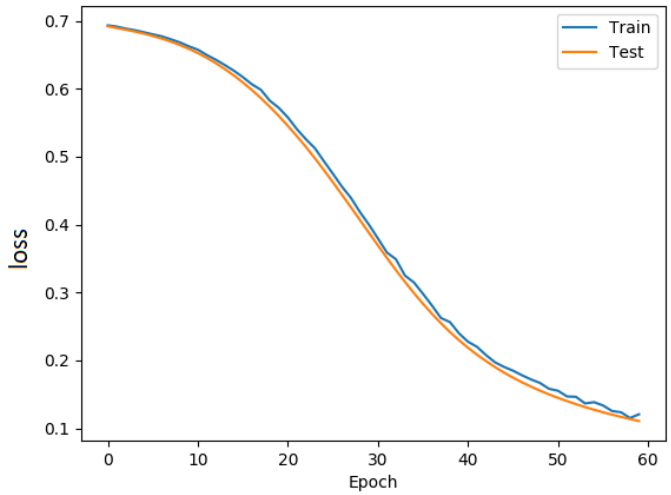

Figure 6.11. Learning path. Binary cross-entropy is used as the loss function (model 1)

Secondly, the dataset with 600 observations of the 4-product case of the model 2 with 3 market states and 81 independent variables respectively is labeled. Analogically to the previous problems 10-fold cross validation is involved to split the dataset by training and test subsets. In 10 generations of neuroevolutionary morphism the MLP-based classifier with the average *F_1-score* of 0.98 has been derived (Figure 6.12.).

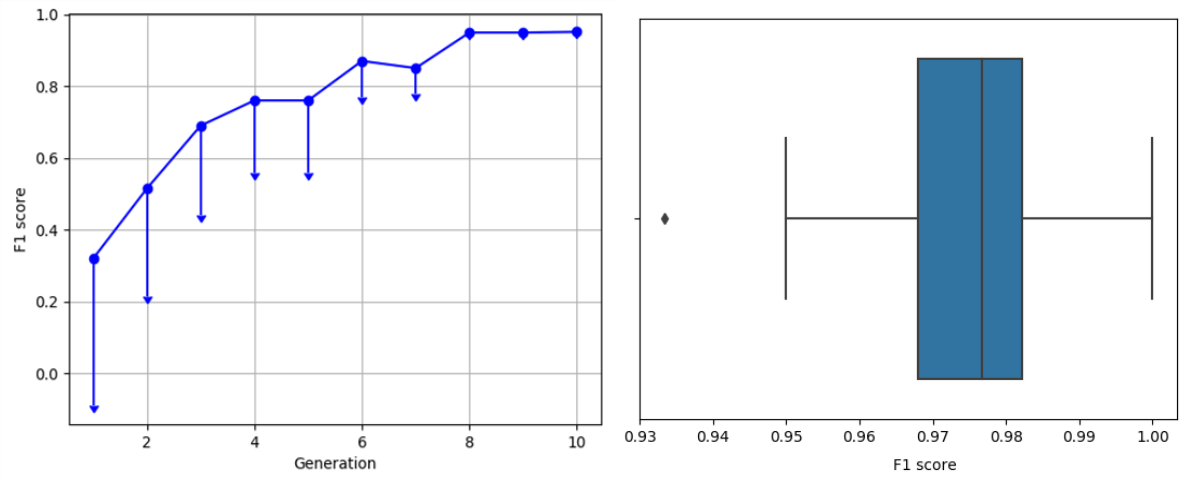

Figure 6.12. The convergence path (vertical arrows represent the standard deviation of the fitness function within the generation) and box-plot of *F_1-score* calculated using 10-fold cross-validation (model 2)

As well as in the previous numerical example, the variance of *F_1-score* calculated based on all the cross-validation slices is not significant, which emphasizes the capability of the MLP-based classifier to learn and generalize nonlinear relationships. In addition, it is important to notice that even the perfect *F_1-score* was achieved for one of ten iterations of the 10-fold cross-validation, which corresponds to absolutely perfect accuracy. The Figure 6.13 demonstrates, how accuracy, precision and recall are improving during the learning process.

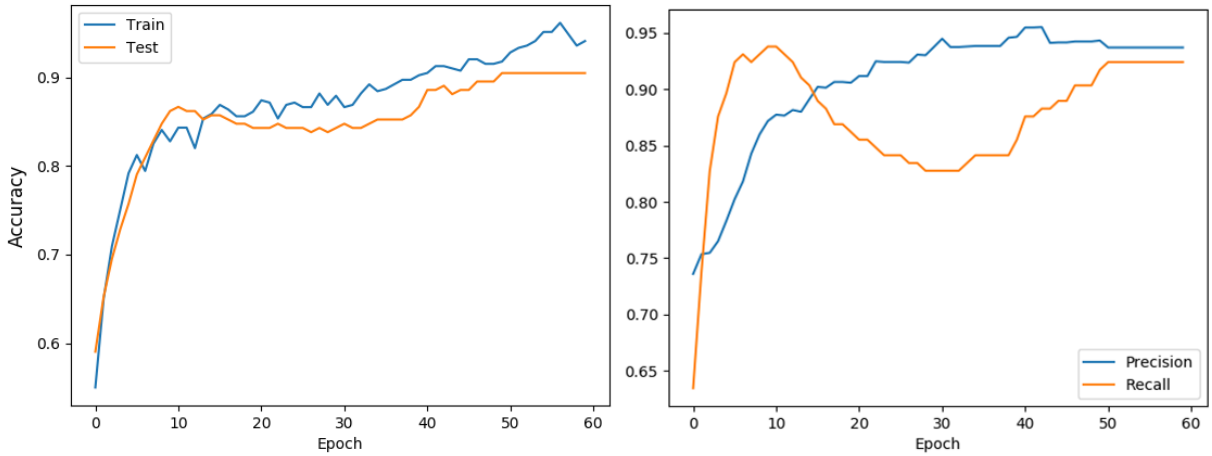

Figure 6.13. Accuracy improves during the learning process (model 2)

The fittest MLP-based classifier has a compact 2-layered architecture with 70 neurons in each layer (15541 trainable parameters totally). The Table 6.6. shows the derived CM.

Table 6.6. The confusion matrix of the model 2

Confusion matrix (Model 2)		Actual class	
		Profitable	Non-profitable
Predicted class	Profitable	142	15
	Non-profitable	18	425

The fittest classifier uses SELU as AF and Nadam optimizer with the relatively low learning rate of 0.02**.** The Figure 6.14. shows the learning path.

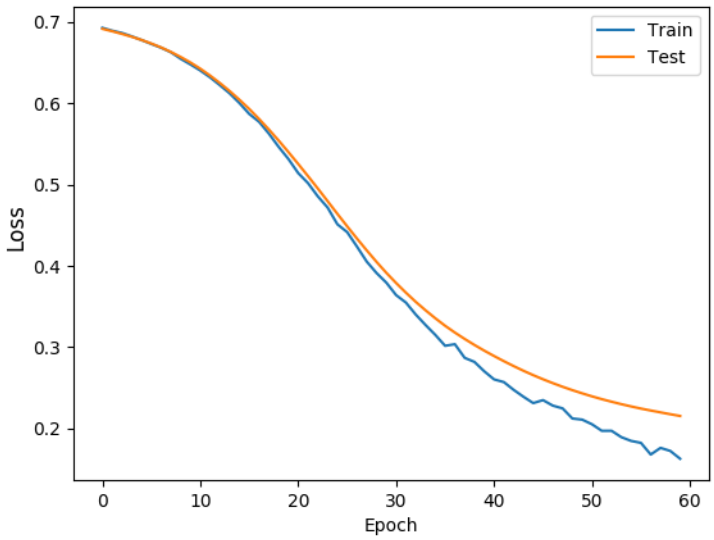

Figure 6.14. Learning path. Binary cross-entropy is used as the loss function (model 2)

The most important measures related to classifiers' accuracy are derived from CMs and demonstrated in the Table 6.7.

Table 6.7. Accuracy of both classifiers

Models	Metrics related to classification accuracy					
	Loss	Accuracy	Precision	Recall	F_1	Cohen's kappa
Model 1	0.11	0.96	0.94	0.98	0.96	0.93
Model 2	0.21	0.94	0.97	0.96	0.96	0.86

Besides, it is important to emphasize that multiclass classification can be done in the same way by, for example, combining several MLP-based binary classifiers.

6.4. Conclusions

In conclusion MLP-based metamodels are capable to learn and generalize complex nonlinear relations between simulation variables and, thus, may be efficiently applied for metamodeling of complex real-world IC systems. Despite the fact that ANN is generally robust to stochastic noise in a training sample, the MLP-based metamodel will inevitably have some sort of upper-bound for accuracy. However, the amount of noise can be controlled by increasing the number of simulation's replications with regard to computational budget, which leads to a classical trade-off between accuracy and computational efficiency. Moreover, both numerical and categorical

variables can be used as independent variables. Besides, it was demonstrated that, if the dataset is labeled in accordance with certain criteria, metamodeling can be alternatively viewed as a binary classification problem. The Table 6.8. contains the specification of architecture and hyperparameters of the fittest MLP-based metamodels and classifiers derived during the neuroevolutionary procedure.

Table 6.8. Architecture of the fittest MLP-based metamodels and classifiers

Model under consideration	Associated problem	Depth (layers)	Width (neurons in the layer)	Optimizer	Learning rate	AF
Model 1	Regression	3	60	Adamax	0.02	ELU
Model 2	Regression	3	55	Nadam	0.07	ReLU
Model 1	Classification	2	60	Adamax	0.04	SELU
Model 2	Classification	2	70	Nadam	0.02	SELU

Paying attention on the components underneath the fittest MLP it should be pointed out that all the derived networks have crucially important things in common. Firstly, all the MLPs are distinguished by relatively compact and shallow architectures of 2-3 hidden layers. Secondly, MLPs incorporate optimizers that are built upon Adam, which highlights the importance of momentum in the learning process. It is also worth to emphasize that the learning rates are relatively low for all cases, because it provides a smoother optimization reducing the chance of skipping the potential optimum.

All the MLPs take advantage on AF from linear unit family, which leads to good gradient propagation and prevents problems associated with the vanishing gradient descent. Additionally, it is worth to point out that, by definition, these AFs need much less computational time and memory than the corresponding simulation models, which makes it extremely useful for metamodel-based optimization, which is discussed in the next chapter.

Concerning the computational efficiency, first and foremost it is important to point out that the simulation models of IC systems and respective MLP-based metamodels were executed using different hardware components, namely the metamodels are implemented on graphics processing

unit (GPU) and simulations on central processing unit (CPU) in parallel utilizing all 4 processor cores (Table 6.9). In this regard, a direct comparison of computational efficiency would be incorrect. On the other hand, since this research focuses on metamodeling automation, the fact that ANN in general and MLP in particular can be naturally implemented in parallel and run using GPU is a significant advantage. Namely, state-of-the-art libraries and frameworks allow one to train and use ANN on the GPU by default without additional investment of human labor. However, implementation of the simulation model in parallel is a separate, rather labor-intense and time-consuming technical task that requires the participation of a human-expert.

Table 6.9. Computational efficiency

Computational resource		Computational time				
CPU & RAM	GPU	Models	Simulation run [s]	η runs[s]	Trained meatmodel to predict one observation [s]	Neuroevolution [h]
Intel Core i-7 6700HQ 4-core 2.6 GHz RAM-8GB	NVIDIA GeForce GTX 960M 4GB	Model 1	2.7	94.5 (35 runs)	0.073	4.1
		Model 2	3.1	136.4 (44 runs)	0.051	3.3

In the context of computational efficiency, it is worth to emphasize that the output of the MLP-based metamodel is deterministic, which is an additional advantage of the MLP-based metamodels over simulations that require dozens runs to obtain an average value. Besides, the total number of trainable parameters (neurons and weights) can be adjusted with regard to the computational budget. In other words, it is up to a potential user to decide how computationally fast the metamodel should be.

7. METAMODEL-BASED OPTIMIZATION

The detailed IC simulations can be computationally expensive to run, which is a challenge when optimization procedure requires many evaluations. Fortunately, an accurate MLP-based metamodel allows one to ease the computational burden. Adequate MLP-based metamodel approximates the detailed simulation model of IC system and, by definition, requires less computational effort (Ryberg, 2013). Namely, MLP-based metamodels can be trained once based on a dataset derived from consecutive replications of a simulation runs as it was demonstrated in the chapter 6. After that the optimization can be conducted iteratively using the trained MLP-based metamodel. There are dozens of optimization algorithms suitable for solving this problem, but once again GA will be chosen for demonstration for the following reasons. Firstly, the framework is already equipped with GA and, because of universality of the algorithm, it is convenient to use it for both neuroevolution and metamodel-based optimization. Secondly, GAs are praised by simulation community for robustness in searching through complex spaces (Carson and Anu, 1997).

Optimization using MLP-based metamodels has several distinct advantages over the classical SO approaches. Firstly, the output of the MLP-based metamodel is deterministic. Secondly, MLP-based metamodel is, by definition, are inexpensive to compute and require less computer memory (Barton and Meckesheimer, 2006). This fact is especially true for relatively shallow MLP-based metamodels equipped by AFs from the linear unit family.

7.1. Implementation

In the case of the model 1, GA searches for a pair of control parameters (Q, r). Such that $Q = (Q_1, Q_2, \ldots Q_n)$ and $r = (r_1, r_2, \ldots r_n)$ that result the highest output (net profit). In the case of the model 2, GA searches for a pair of control parameters (G, H), such that $G = (G_1, G_2, \ldots, G_n)$ and $H = (H_1, H_2, \ldots, H_n)$, which define the inventory level that must be reached to stop and resume production process.

The chromosome can be firstly represented as a list of integers $\vec{v} = (Q_1, r_1, \ldots Q_n, r_n)$ for the case of the model 1. Such that odd elements represent order sizes and even elements stand for reorder points. For the case of the model 2 the chromosome takes the following form $\vec{v} = (G_1, H_1, \ldots G_n, H_n)$. Such that odd elements stand for stoppage criteria and even elements for resumption

criteria. After that the chromosome is encoded into the Gray-code binary representation (Figure 7.1). Once again, such a coding scheme can be justified by space-efficiency and guarantee of gradualism associated with smaller Hamming clefts (Caruana and Schaffer, 1988).

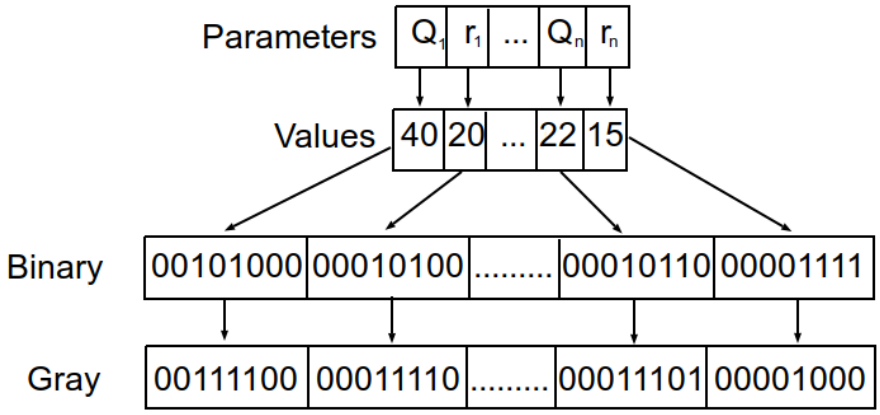

Figure 7.1. Control parameters of the model 1 are encoded into the Gray-code representation

In order to demonstrate the universality of the method, optimization procedure is driven by the same GA that is already incorporated into the framework. Hyperparameters are also the same, namely, *t_size* of 5, crossover probability of 0.35 and mutation probability of 0.04. However, it is worth to note that hyperparameter variation in the sane range has not notably affected convergence speed. Figure 7.2 demonstrates the uniform crossover operator with regard to the chromosome representation for the model 1.

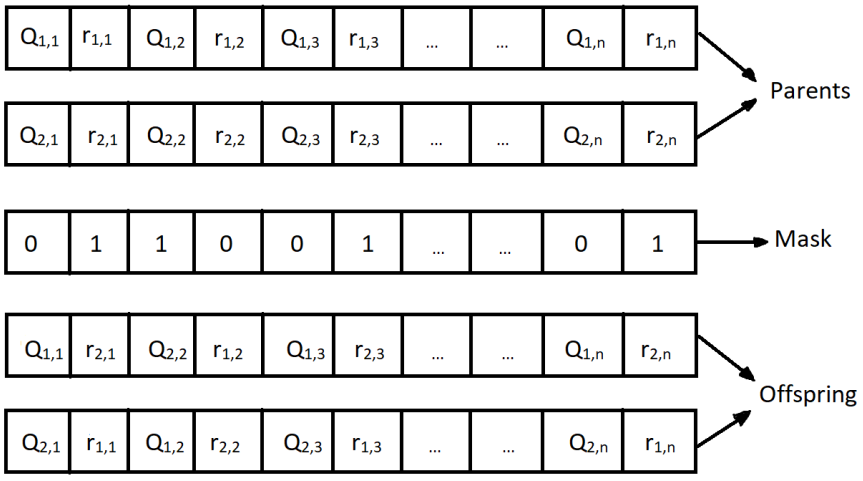

Figure 7.2. Uniform crossover in the context of the model 1

Fitness of every single candidate solution corresponds to the net profit approximation by the MLP-based metamodels derived in the chapter 6. Besides, a candidate solution must satisfy the following constraints:

$$\sum_{i=1}^{n} Q_i \leq I_{max} \; and \; r_i < Q_i \tag{7.1}$$

$$\sum_{i=1}^{n} H_i \leq I_{max} \; and \; H_i < G_i \tag{7.2}$$

If constraints are violated, fitness will take extremely low values, due to infeasibility of such a solution.

The logic behind GA for obtaining optimal control parameters is the same as for the neuroevolutionary network morphism. Namely, by applying such genetic operators as crossover and mutation to the parental solutions, a set of new candidate solutions (the offspring) is obtained. After that, the offspring compete with the parents for a place in the next generation based on their fitness. This process is repeated based on several iterations of the principal evolution cycle until stoppage criteria is met (Figure 7.3). The Python 3.7 implementation is provided in Appendix 8.

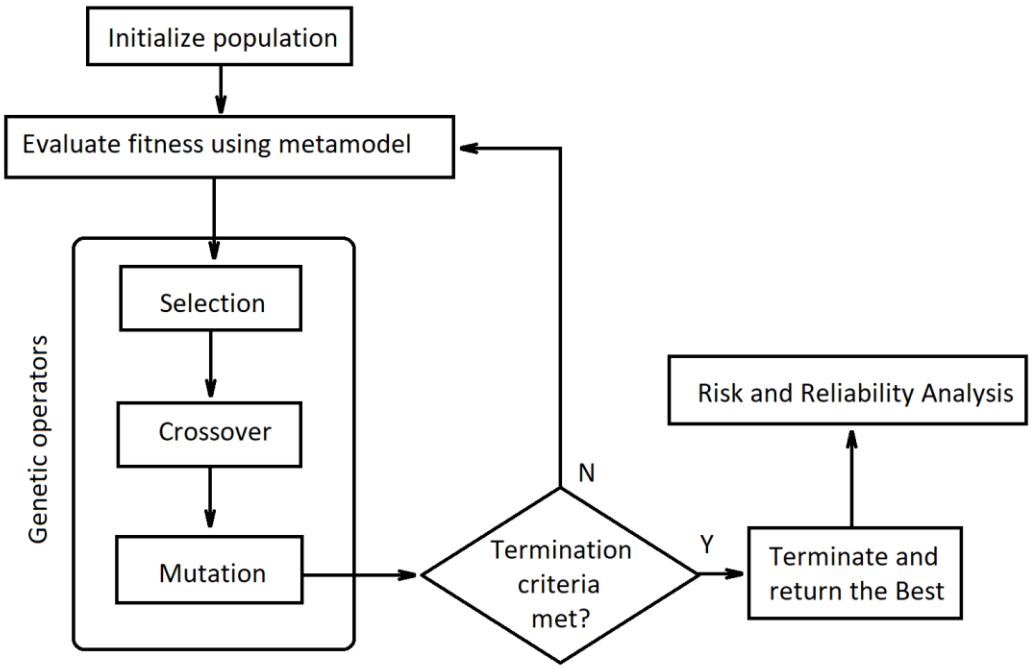

Figure 7.3. The logic behind GA-driven metamodel-based optimization

7.2. Accuracy of the method

The proposed metamodel-based optimization approach is applied to the MLP-based metamodels of model 1 and model 2 built in the chapter 6. Namely, 10-product version of the model 1 and 4-product version of the model 2 are considered. All the inputs except control parameters are randomly generated 30 times in feasible range. As the result 30 test problems are obtained. For each test problem, the output (net profit) to the control parameters obtained by the metamodel-based optimization is compared against the output to the control parameters obtained by classical SO driven by the same GA. Both the metamodel-based optimization and SO last 31 generations, and each generation is populated with 100 candidate solutions. Percentage error is used as a metric to compare the accuracy (Figure 7.4).

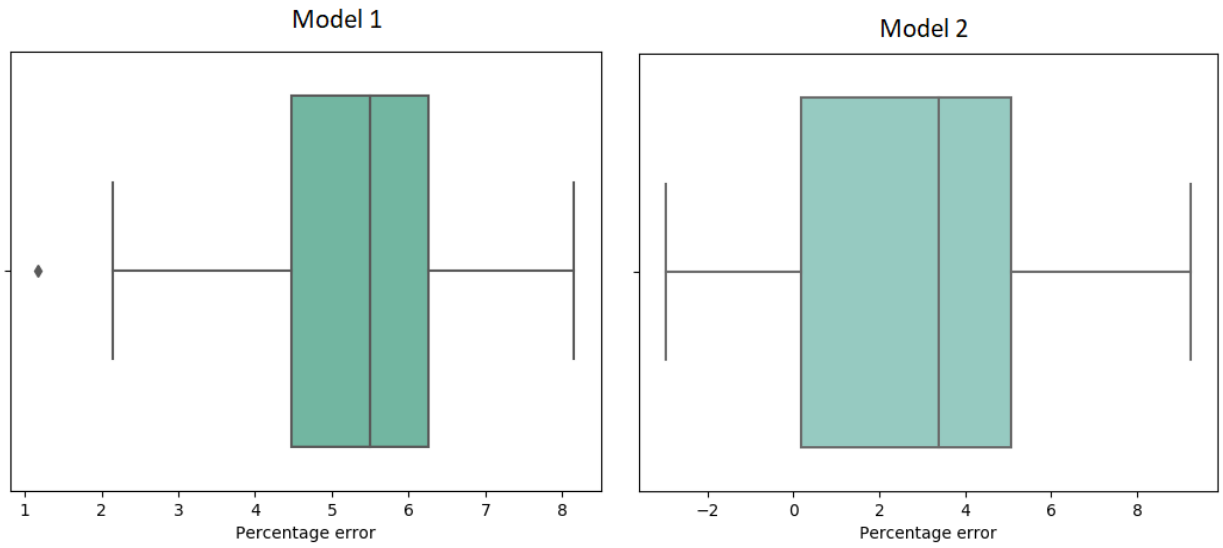

Figure 7.4. Model vs. Metamodel (model 1, 2)

Mean percentage error of 5.21 for model 1 and 2.93 for model 2 can be interpreted as insignificant taking into account complexity, stochastic nature of a problem and limited computational budget. Given the fact that MLP-based metamodels are, by definition, approximations of the original simulation models, an error is simply inevitable. It is worth to note that these experiments are conducted to proof the concept, however, the accuracy of the MLP-based metamodel can be improved further by increasing the number of simulation's replication and performing more exhaustive neuroevolutionary and optimization searches. Nevertheless, these

measures are associated with an increase in the computational budget, which leads to the classic trade-off between accuracy and computational efficiency. Detailed comparison of the accuracy is presented in the Table 7.1.

Table 7.1. Comparison of SO and metamodel-based optimization based on percentage error for 30 test problems

Model	Number of experiments	Mean percentage error	Standard deviation of percentage error	Minimal percentage error	Maximum percentage error
Model 1	30	5.21	1.58	1.16	8.15
Model 2	30	2.93	3.12	-2.98	9.26

The Figure 7.5 demonstrates one example of optimization procedure for each model.

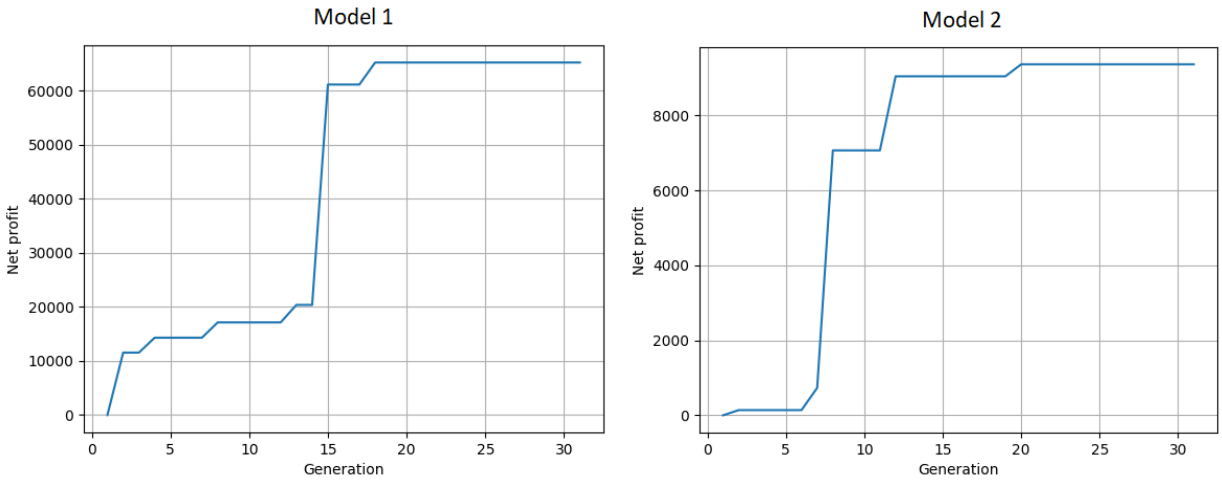

Figure 7.5. The example of convergence paths

The fittest candidate solution for the model 1 was obtained in 18 generations and results net profit of 65734 abstract monetary units. The fittest candidate solution for the model 2 was obtained in 20 generations and results net profit of 8817 abstract monetary units. These examples will be used to demonstrate the potential for risk analysis.

7.3. Risk analysis

Metamodel-based optimization assumes that nearly-optimal candidate solutions derived using a MLP-based metamodel are also likely to be nearly-optimal for the corresponding simulation of

IC system. Indeed, it does not imply that the best solutions will match. As it is mentioned by Juan *et al*. (2015), this assumption is true for the vast majority of practical applications. It is important to keep in mind the fact that MLP-based metamodels rather complement the original simulation model, allowing one to execute a faster and more exhaustive search, but the original model of IC system is surely not discarded. Moreover, once the elapsed time is expired, the original simulation of IC systems can be used to examine promising candidate solutions in detail. Besides, simulation models can be used in order to provide insights on the behavior of real systems and depict industrial dynamics.

Therefore, the notable advantage of such a combined approach is the possibility to obtain additional information through the original simulation model for further risk and analysis. Namely, due to the fact that the simulation output is a random variable, a potential decision maker can be interested in a solution with lower risks, but not in the solution that simply leads to the output with the highest expected value. For example, let's consider 5 of the most promising solutions for the model 1 highlighted during GA-driven metamodel-based optimization (Figure 7.6).

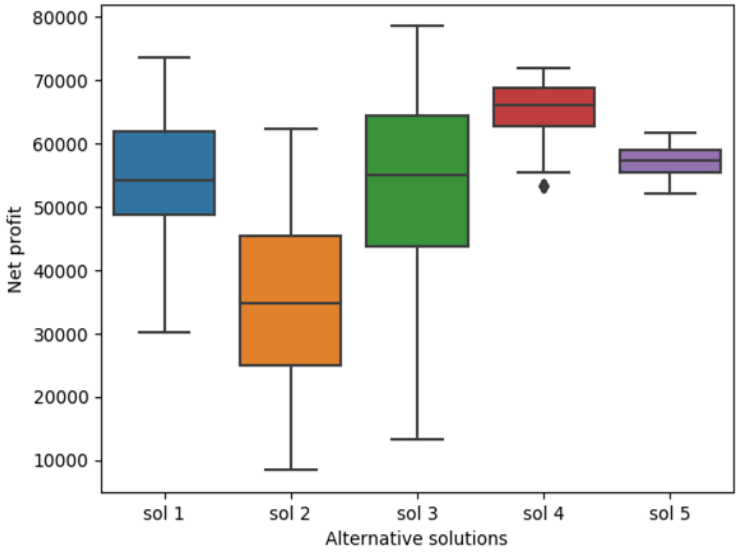

Figure 7.6. The promising solutions highlighted during GA-driven metamodel-based optimization (model 1)

Despite the fact that the solution 4 is distinguished for the highest mean value, the solution 5 has notably smaller standard deviation, thus can be more attractive for a risk-averse decision maker. On the other hand, risk-loving decision maker will be, most likely, interested in the solution 3, since it has the highest possible net profit. Such variability in net profit within a solution can be

explained by the likelihood of backorder. For example, the Figure 7.7 illustrates that the for the solution 4 the probability that demand during the replenishment will be higher than the order size is significantly lower compared to the solution 3.

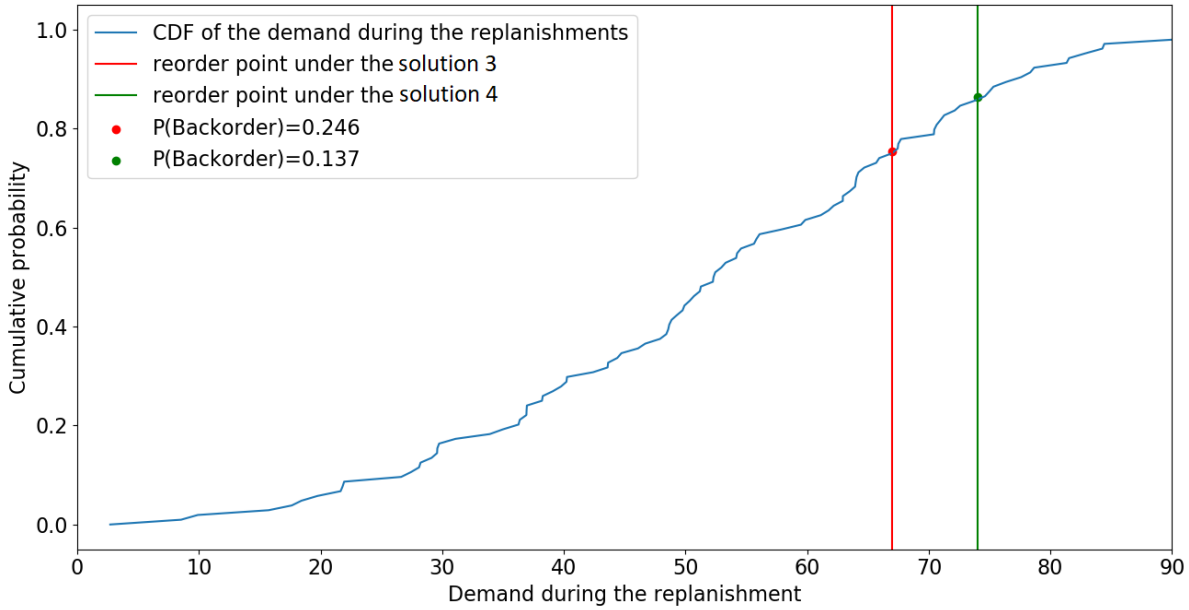

Figure 7.7. Empirical backorder risk comparison for two candidate-solutions

Figure 7.8 demonstrates 5 of the most promising solutions for the model 2 highlighted during metamodel-based optimization. According to the figure, the solution 4 is associated with high risk and has the lowest expected value, thus, is likely to be ignored by any rational decision maker. The solution 1 and the solution 5 are characterized by a low degree of risk, however, since the solution 1 is significantly more profitable in the average case, it will be preferred by a risk-averse decision maker. On the other hand, the solutions 2 and 3 will be pleasant to an extremely risk-loving decision maker, who is rather akin to gambler.

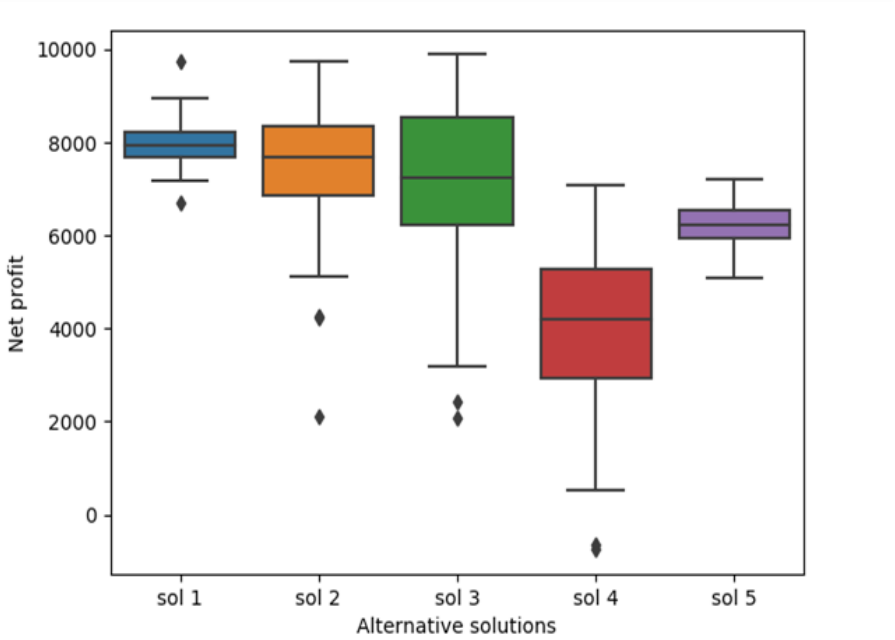

Figure 7.8. The promising solutions highlighted during GA-driven metamodel-based optimization (model 2)

7.4. Tackling the "curse of dimensionality" with unsupervised learning

An average inventory system contains immense number of stock keeping units (SKUs). Unfortunately, the computational effort to solve IC problems expressed in so many variables, as a rule, grows exponentially with the number of products that IC system operates with. Therefore, in general case, it is computationally hard to consider each item individually and manage it under individual inventory policy. In addition, high dimensionality negatively affects the behavior and performance of learning algorithms, including ANNs (Verleysen and Damien, 2005).

In light of these facts, it is not surprising that 30 years ago an essentially important question was asked: "how to aggregate stock units into groups so that the resulting inventory policies are sufficiently close to those policies that would have been generated if every unit was treated individually?" (Ernst and Cohen, 1990). What industry needs, in fact, is an algorithm that will be able to group together similar SKUs. This problem is well-known as clustering, one of the most intensively studied subfield in machine learning (Domingos, 2015).

Confronted with the curse of dimensionality related to the immense number of SKUs, the natural recourse is to group them into manageable number of clusters. That is why, the earliest attempts to group SKUs by cluster analysis may be tracked back to the study of Srinivasan and

Moon (Srinivasan and Moon, 1999). The researchers introduced a hierarchical clustering-based methodology for supporting inventory systems in supply chains. Several heuristics were applied to identify the relationships between items and take into account product features with a significant impact on supply chain. The CH index was used to validate the result of the average linkage clustering. In 2007 the K-means-based SKU segmentation methodology was proposed (Egas and Masel, 2010). The research aimed to reduce the time required to compute the inventory-control parameters in large-scale multi-echelon inventory system. The segmentation methodology was tested on different multi-echelon inventory systems in order to understand an effect of resulting penalty costs. Three years later the similar k-means-based approach by Egas and Masel was applied to determine storage assignments (Desai, 2007). It is also important to emphasize the recent paper (Yang and Nguyen, 2016). The authors conducted a study on the application of a constrained clustering method reinforced with principal component analysis (PCA). According to the research, the proposed method is able to provide significant compactness among item clusters.

Summarizing previous practices, the author's paper discusses an application of various clustering algorithms to solve the SKU-aggregation problem (Jackson *et al*., 2018b). The research uses dataset provided by the "Trialto Latvia LTD", the third-party logistics provider. Based on the approach described in this paper, it is convenient to demonstrated that proposed metamodel-based optimization can be efficiently combined with the popular practice of SKU's clustering. The Python 3.7 implementation is provided in Appendix 9.

7.4.1. Dataset description and data preprocessing

The initial dataset consists of 2279 SKUs (observations) with 9 features. Selected features include only numerical data (Table 7.2). All the features have an undeniable impact on the inventory management and constitute two core groups: handling-related and turnover-related. Such features as expire date, pallet weight, pallet height and number of units per pallet determine the speed and subtlety of handling. On the other hand, total outbound and number of outbound orders indicate how tradable a particular SKU is. The total outbound and the number of outbound orders is represented as different attributes despite the fact of sharing some mutual information. It is done on purpose, since both the demand size and the demand frequency are important for the research. It is also worth to note that the feature "number of outbound orders" is calculated based on arisen demand from 2017-02-06 to 2018-02-13 (537791 orders in total).

Table 7.2. Selected features

ID	Unit price (EUR)	Expire date (days)	Total outbound (units)	Number of outbound orders	Pallet weight (kg)	Pallet height (cm)	Units per pallet
1	0.058	547	2441	9	105.6	1.56	1920
2	0.954	547	0	0	207.68	1.00	384
3	2.385	547	23	12	165.78	1.02	108
…	…	…	…	…	…	…	…
2278	2.02	730	710	354	322.56	1.19	288
2279	1.99	730	765	363	322.56	1.19	288

The excessive presence of missing data was the first relevant problem. Such features as “unit price”, “pallet gross weight”, “pallet height” and “units per pallet” contain 710(31.1%), 371(16.3%), 787(34.5%) and 295(12.9%) missing values respectively. Moreover, 1070(47%) observations include at list one missing value and all four features are missing in 208(9.1%) observations. Since each observation stands for a real tradable product, such precious information cannot be dropped. Since such naive methods for treating missing data as replacement by the feature mean can insert a bias (Zloba and Yatskiv, 2002), it has been decided to impute missing values using the nearest neighbor imputation (NNI). NNI, justifying its name, imputes missing values using values calculated from the k nearest neighbors. The nearest neighbors are found by minimizing a distance function, in our case the Euclidean distance (Jonsson and Wohlin, 2004).

According to Chen and Shao (2001), NNI has two pivotal benefits. Firstly, the method provides asymptotically unbiased and consistent estimators for population means. Secondly, the NNI method is expected to be more robust against model violations than methods based on parametric models, such as ratio imputation and regression imputation. An important parameter for the NNI method is the value of *k*. In this *k* of 10 is used, according to the suggestion (Batista and Monard, 2002).

Since anomaly detection and clustering algorithms incorporated to this research utilize distances between points as a metric, attribute scaling becomes a necessary prerequisite. This research applies Z-score attribute standardization relying on the research by Mohamad and Usman,

which concludes with the statement that the application of Z-score standardization prior to K-means clustering leads to more accurate result compared to decimal scaling and min-max normalization (Mohamad and Usman, 2013).

In light of the fact that clustering algorithms, which use distances between points as a metric, such as k-means, are extremely sensitive to presence of anomalies in dataset, it is crucial for quality of cluster analysis to identify and remove such anomalies. For this purpose, the local outlier factor (LOF) was applied. Despite the fact that LOF is a relatively old algorithm, Campos, *et al*. conclude with the statement that even after 16 years of active research LOF remain the state of the art, especially in datasets with possibly larger amounts of outliers (Campos *et al*., 2016).

Applying the NNI the value of *k* is varied in the range from 3 to 10 and did not observe a significant difference, thus, it was concluded to use *k* of 10 as suggested by Batista and Monard (2002). As the result, all the missing values were successfully imputed. Right after that the data was transformed using Z-score standardization.

Since clustering algorithms that use distances between points as a metric, such as k-means, are extremely sensitive to presence of anomalies in dataset, the LOF algorithm was applied iteratively varying *k* in reasonable range from 5 to 30. What was interesting, in each run exactly 114 outliers (5.1%) have been detected. Roughly speaking, an outlier is an observation that deviates too much from other observations as to arouse suspicion that it is generated by a completely different mechanism. However, for inventory management outliers are the most precious pieces of data. In this particular case identified outliers stand for small SKUs with very high unit price, SKUs with high total demand, but low demand frequency, SKUs with extremely high or extremely low pallet weight and so on. These findings were reported to the representative of the “Trialto Latvia LTD”, however, they were also dropped from the dataset prior to cluster analysis for the purpose of clustering validity. As the result 2165 observation remained.

Right after that mean-shift with a flat kernel is applied to estimate the number of blobs and potential clusters respectively. The algorithm detected 26 potential clusters, however, the vast majority of these clusters (18) consists of 3 or less observations. These observations are, in fact, outliers that were able to survive the anomaly detection with LOF. 35 new-found outliers were trimmed and the procedure was repeated iteratively. In the second iteration 28 more outliers were removed and 10 potential clusters were discovered with no single cluster containing less than 6

observations. Eventually, 63 additional anomalies missed by LOF (2.9 %) were detected and trimmed. As the result 2102 observation remain in the dataset.

7.4.2. Cluster analysis

10 clusters defined by mean-shift (Figure 7.9) were a starting point. Mean-shift served the research well discovering existing blobs and potential outliers, however, the cluster analysis by itself is quite poor, according to all the indexes (see Table 2). DBSCAN showed even worse results due to the fact that the dataset contained quite convex clusters with large differences in densities. This also implies the problem that DBSCAN considers a significant portion of the observations as noise. For instance, the algorithm with ε (local radius for expanding clusters) of 0.747 defined 10 clusters classifying 173 observations (8.2%) as noise. Since each observation stands for a product and 177 outliers were already trimmed, and it is not affordable to lose any more data. Besides that, the clustering result is generally very poor based on all the indexers.

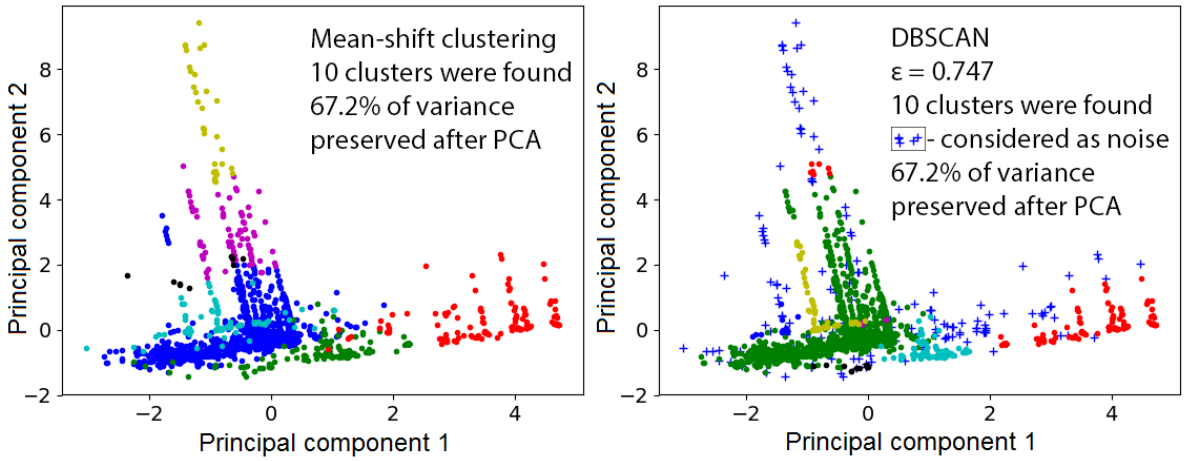

Figure 7.9. Cluster analysis by mean-shift and DBSCAN visualized via PCA

Since k-means clustering requires some prior guessing about the number of clusters (Cheng, 1995), mean-shift is used assuming that the number of found "clots" corresponds to a nearly-optimal value of k parameter in k-means clustering. This guess turned out to be correct, namely k-means with *k* of 10 results the best segmentation based on the silhouette index and DI, holding one of the largest values of the CH-index at the same time. It may also be observed that *k* of 9 demonstrates a quite promising result (Figure 7.10).

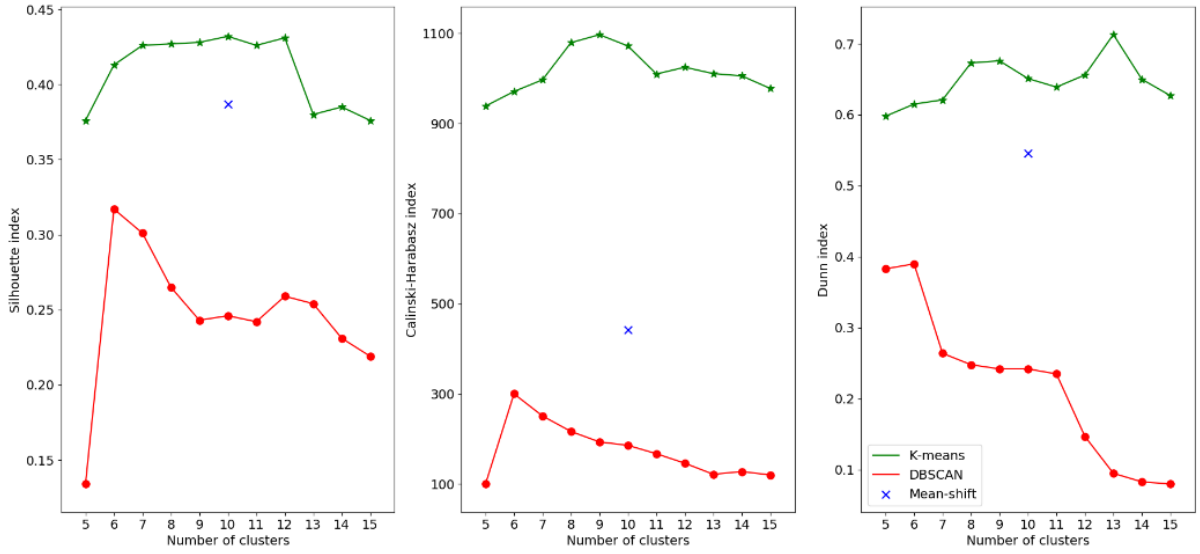

Figure 7.10. Values of validity indexes depending on the number of clusters

Dimensionality reduction gave hope to improve the current result for two main reasons. Firstly, some attributes are interrelated and correlate with each other sharing some mutual information (Table 7.3). Secondly, it is quite natural for real-world data to contain noise, for instance, due to the measurement error. Additionally, noise could be inserted during NNI procedure. Based on that PCA could be useful as a noise-reduction tool.

PCA may be defined as a common statistical procedure that maps the data to a new coordinate system such that the first coordinate contains the greatest variance, the second coordinate contains the second greatest variance and so on. This study incorporates PCA for two major purposes. Firstly, PCA is applied after all preprocessing and clustering for data-visualization. For this purpose, exactly two principal components are derived, such that each principal component corresponds to an axis on 2-d plot. Secondly, the result obtained from k-means can be improved relying on noise-reduction and feature-extraction properties of PCA.

Eventually, the incorporation of PCA was indeed a right decision, since gradually lowering the number of principal components to 4, the initial result was improved on 11% according to silhouette validity index preserving more than 95% of variance (Table 7.4). It is also important to emphasize that such indexes as CH and DI rose even more. However, these figures are less representative due to excessive sensitivity of such indexes to dimensionality.

Table 7.3. Pearson correlation coefficient of attributes

	Expire date	Total outbound	Number of outbound orders	Pallet weight	Pallet height	Units per pallet
Unit Price	-0.08	-0.07	-0.09	-0.09	-0.09	-0.04
Expire Date	1.00	0.08	0.07	-0.35*	-0.36*	0.04
Total Outbound	-	1.00	0.86**	0.04	-0.04	-0.03
Number of orders	-	-	1.00	-0.04	0.01	0.00
Pallet Weight	-	-	-	1.00	0.28*	0.06
Pallet height	-	-	-	-	1.00	-0,04
* indicates moderate linear relationship ** indicates strong linear relationship						

Table 7.4. 10-means after PCA

Number of components	Silhouette	CH	DI	Preserved variance
Prior to PCA	0.432	1072.0	0.639	100%
6	0.441	1120.1	0.662	99.10%
5	0.452	1163.5	0.667	97.82%
4	**0.475**	**1368.3**	**0.750**	**95.11%**
3	0.456	2173.1	0.746	82.09%
2	0.454	3641.7	0.581	67.21%

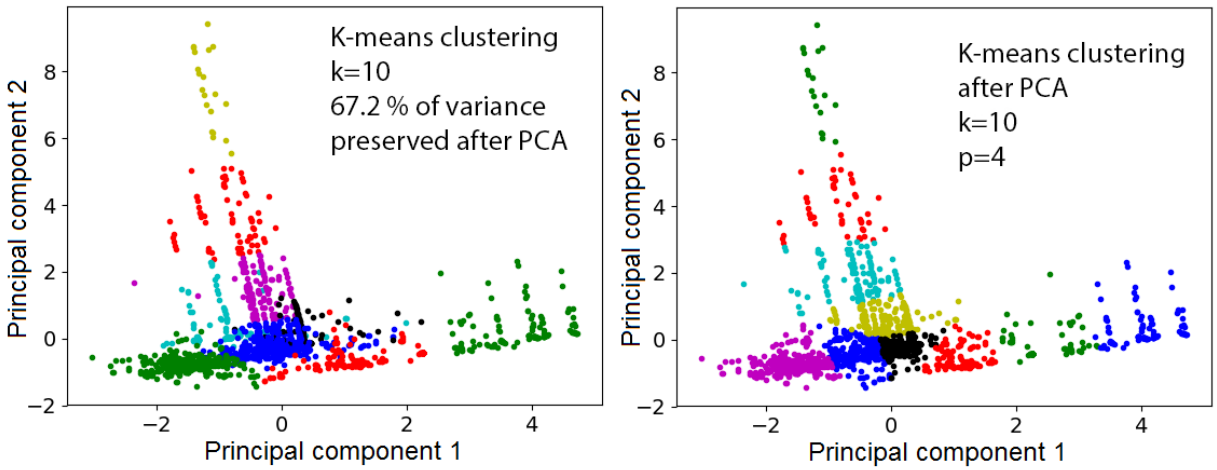

Figure 7.11. K-means clustering prior to and after PCA with 4 principal components

It is worth to mention that with the combination of PCA and k-means high-quality SKUs' segmentation was achieved, according to internal validity tests. On the other hand, due to the fact that that the dataset comprises quite convex clusters with large differences in densities, DBSCAN was quite inefficient. Since the obtained clusters are homogeneous, all observations in it can be treated under a common IC policy saving precious computational resources.

7.5. Conclusions

The detailed IC simulations can be extremely cumbersome from the computational point of view, which is a challenge when optimization procedure requires many evaluations. Fortunately, an accurate MLP-based metamodel allows one to ease the computational burden. Adequate MLP-based metamodel approximates the detailed simulation model of IC system and, by definition, requires less computational effort (Ryberg, 2013). Namely, MLP-based metamodels can be trained once based on a dataset derived from consecutive replications of a simulation runs as it was demonstrated in the chapter 6. After that the optimization can be conducted iteratively using the trained MLP-based metamodel. There are dozens of optimization algorithms suitable for solving this problem, but once again GA will be chosen for demonstration for the following reasons. Firstly, the framework is already equipped with GA and, because of universality of the algorithm, it is convenient to use it for both neuroevolution and metamodel-based optimization. Secondly, GAs are praised by simulation community for robustness in searching through complex spaces (Carson and Anu, 1997).

Optimization using MLP-based metamodels has several distinct advantages over the classical SO approaches. Firstly, the output of the MLP-based metamodel is deterministic. Secondly, MLP-based metamodel is, by definition, are inexpensive to compute and require less computer memory (Barton and Meckesheimer, 2006). This fact is especially true for relatively shallow MLP-based metamodels equipped by AFs from the linear unit family.

Metamodel-based optimization assumes that nearly-optimal candidate solutions derived using a MLP-based metamodel are also likely to be nearly-optimal for the corresponding simulation of IC system. Indeed, it does not imply that the best solutions will match. As it is mentioned by Juan *et al*. (2015), this assumption is true for the vast majority of practical applications. It is important to keep in mind the fact that MLP-based metamodels rather complement the original simulation model, allowing one to execute a faster and more exhaustive search, but the original model of IC system is surely not discarded. Moreover, once the elapsed time is expired, the original simulation of IC systems can be used to examine promising candidate solutions in detail. Besides, simulation models can be used in order to provide insights on the behavior of real systems and depict industrial dynamics.

Therefore, the notable advantage of such a combined approach is the possibility to obtain additional information through the original simulation model for further risk analysis. Namely, due to the fact that the simulation output is a random variable, a potential decision maker can be interested in a solution with lower risks, but not in the solution that simply leads to the output with the highest expected value.

8. CONCLUSIONS

For over a century IC has been a major point of discussion in academic, scientific and business worlds. Analytical models still hold the huge lay of the land in IC theory, especially in academia. However, as it was demonstrated in the chapter 2, analytical models do not guarantee the optimal solution, if the initial assumptions and considerations are violated. Moreover, such assumptions and considerations frequently do not correspond to real-world IC problems. On the other hand, real-world business-driven IC is frequently distinguished by multidimensionality, non-decomposability and replete with numerous specialties and subtleties. The tendency to choose computationally efficient numerical approximations over explicit analytic solutions rises the popularity of SO techniques. Unfortunately, simulation models, especially detailed, are consuming in terms of computational resources including the number of computational steps and memory. Therefore, it is considered to be more reasonable to use an alternative cheaper-to-compute metamodel, which is specifically built in order to approximate a target simulation with a sufficient degree of accuracy. Thanks to recent advances and rapid development in deep learning, metamodeling with ANN looks extremely promising. Nevertheless, involvement of a human expert is the Achilles heel. Namely, ANNs are usually developed manually by data scientists and artificial intelligence developers, which is quite fault prone and requires the sheer amount of time. Taking advantage on the universal approximation theorem behind MLP and Holland's schema theorem behind GA, the proposed neuroevolutionary framework demonstrated solid accuracy in metamodeling of production-inventory system distinguished by both nonlinearity and stochasticity.

Summarizing the most significant results and substantial discoveries of the research, the following conclusions can be drawn:

- MLP-based metamodels derived by neuroevolutionary process were capable to handle different kinds of nonlinearity and generalize complex relations between simulation variables. This advantage is crucially important, because the relationships between variables in real-world IC problems are complex and nonlinear by nature;
- MLP-based metamodels of IC systems need notably less computational time in comparison with the corresponding simulation model imposing no significant limit on the accuracy at the same time. This advantage is especially significant in case of MLPs equipped with AF from linear unit family, such as ReLU, SELU, ELU and their modifications;

- Despite the fact that ANN is generally robust to stochastic noise, the MLP-based metamodel will inevitably have some sort of upper-bound for accuracy. However, the amount of noise can be controlled by increasing the number of simulation's replications with regard to computational budget, which leads to a classical trade-off between accuracy and computational efficiency;
- Paying attention on the components underneath the fittest MLP it should be pointed out that all the derived networks have crucially important things in common. Firstly, all the MLPs are distinguished by relatively compact and shallow architectures of 2-3 hidden layers. Secondly, MLPs incorporate optimizers that are built upon Adam, which highlights the importance of momentum in the learning process. It is also worth to emphasize that the learning rates are relatively low for all cases, because it provides a smoother optimization reducing the chance of skipping the potential optimum. All the MLPs take advantage on AF from linear unit family, which leads to good gradient propagation and prevents problems associated with the vanishing gradient descent;
- Optimization using MLP-based metamodels has several distinct advantages over the classical SO approaches including computational efficiency and deterministic output. Besides, MLP-based metamodel can be used in combination with the original simulation model in order to obtain additional information through the original simulation model for further risk analysis;
- Concerning the potential disadvantages, it should be pointed out that the derived MLP-based metamodels do not depict the dynamic behavior of the system, but only map the vector of inputs into the corresponding output or label. However, this fact simply emphasizes that, when the MLP-based metamodel is applied for deriving optimal control parameters, the original DES model of IC system does not evaporate anywhere and can also be used to doublecheck the result, simulate dynamic behavior and analysis.

The following results can be considered as a scientific novelty of the research:

- The state-of-the-art neuroevolutionary methods in automated machine learning were explored, examined and adapted for metamodeling and optimization of IC systems;
- Potential of metamodel-based optimization with artificial neural networks is firstly studied for IC systems;

- Neuroevolutionary techniques are firstly adapted for metamodeling in general;
- The proposed framework is an example of how machine-learning techniques and simulation modeling can be used together.

The following statements highlight the practical importance of the research:

- The proposed framework can be integrated into warehouse management systems for automated ordering. This solution can be especially beneficial for retail and lean production-oriented industries, because of a high degree of sensitivity to improper IC in these fields.
- It was demonstrated that, if the dataset is labeled in accordance with profitability criteria, metamodeling can be alternatively viewed and approached as a binary classification problem;
- The proposed framework is an example of how machine-learning techniques and simulation modeling could be applied together. Currently global market is at a point, where machine learning is notably expanding beyond research environments right into the industrial applications. In order to facilitate this process, simulation modeling should be increasingly used to leverage the potential of machine learning. Such that, experts working with either simulation modeling or machine learning will stand to benefit from their combination;
- Since the proposed neuroevolutionary framework is driven by simulation, it does not, by definition, exclude the model developer, who is responsible for development of the original simulation model. In this regard such an approach can be considered as a way of transferring expert knowledge about the IC problem to the system equipped with artificial intelligence. This human-centric approach ensures that human vision of the process is a major factor and primary consideration that complies with the latest guidelines of the European Commission Expert Group on Artificial Intelligence.

Further research is needed in order to make the proposed metamodeling approach more tailored to the industrial needs. In future research it worth to pay attention on weights initialization and incorporate such overfitting-preventing techniques as regularization and dropout into the neuroevolutionary framework. Besides, in order to understand the implications of these results, future studies could also address trade-off between the number of replications and upper-bound

metamodeling accuracy. In addition to that, the application of the obtained metamodels for sensitivity analysis can also be studied. Since IC systems are vital components of supply chains, further research could also address the scalability of neuroevolutionary frameworks to deal with metamodeling of supply chain models.

As a small digression in the end, it is also worth to emphasize that this research takes into account the ongoing reproducibility crisis. It has been recently revealed that lots of experiments presented in the scientific papers are nearly-impossible to replicate (Loken and Gelman, 2017). Since the reproducibility of experiments is a pivot of the scientific method, experimental testing has been conducted in as reproduceable manner as possible and documented in detail. All the source-code along with used datasets can be found in the GitHub repository.

BIBLIOGRAPHY

1. Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J. and Kudlur, M. (2016) Tensorflow: A system for large-scale machine learning. In: *the 12th Symposium on Operating Systems Design and Implementation.* pp. 265-283.
2. Ackley, D. (1987) *A Connectionist Machine for Genetic Hillclimbing.* Boston, MA: Kluwer. pp. 170-215.
3. Altiok, T. (2012) *Performance analysis of manufacturing systems.* Springer Science & Business Media, p. 17.
4. Amaran, S., Sahinidis, N., Sharda, B. and Bury, S. (2016) *Simulation optimization: a review of algorithms and applications.* Annals of Operations Research, 240(1), pp.351-380.
5. Anderson, T. and Darling, D. (1954) A test of goodness of fit. *Journal of the American statistical association*, *49*(268), pp. 765-769.
6. Andradottir, S. and Prudius, A. (2010) *Adaptive random search for continuous simulation optimization.* Naval Research Logistics, 57, pp. 583-604.
7. Angeline, P., Saunders, G. and Pollack, J. (1994) *An evolutionary algorithm that constructs recurrent neural networks.* IEEE transactions on neural networks, 51, pp. 54-65.
8. Angun, E. (2011) *A risk-averse approach to simulation optimization with multiple responses.* Simulation Modeling Practice and Theory, 19, pp. 911-923.
9. Angun, E., Kleijnen, J., den Hertog, D. and Gurkan, G (2009) *Response surface methodology with stochastic constraints for expensive simulation.* Journal of the Operational Research Society, 60, pp. 735-746.
10. Ansof, H. and Slevin, P. (1968) *An appreciation of industrial dynamics.* Management Science, 14, pp. 91-106.
11. Antonisse, H. and Keller, K. (1987) Genetic operators for high-level knowledge representations In: *2nd International Conference on Genetic Algorithms. Cambridge, MA,* pp. 69-76.
12. Aras, N., Verter, V. and Boyaci, T. (2006) Coordination and priority decisions in hybrid manufacturing/remanufacturing systems. *Production and Operations Management*, 15(4), pp. 528-543.
13. Archibald, B. (1981) Continuous review (s, S) policies with lost sales. *Management Science*, 27, pp. 1171-1177.

14. Arreola-Risa, A., Gimenez-Garcia, V. and Martinez-Parra, J. (2011) Optimizing stochastic production-inventory systems: A heuristic based on simulation and regression analysis. *European Journal of Operational Research*, 213(1), pp. 107-118.
15. Arrow, K., Harris, T., and Marschak, J. (1951) Optimal Inventory Policy. *Econometrica*, 29, pp. 250-272.
16. Arrow, K., Karlin, S., and Scarf, H. (1958) *Studies in the Mathematical Theory of Inventory and Production*. Stanford, CA: Stanford University Press.
17. Avci, H., Gokbayrak, K., Nadar, E. (2019) Structural results for average-cost inventory models with Markov-modulated demand and partial information. *Production and Operations Management*, 45(2), pp. 93-102.
18. Axsater, S. (1985) Control theory concepts in production and inventory control. *International Journal of Systems Science*, 16, pp. 161-169.
19. Back, T., Fogel, D. and Michalewicz, Z. (2018) *Evolutionary computation 1: Basic algorithms and operators*. CRC press, pp. 25-56.
20. Balkhi, Z. (1999) On the global optimal solution to an integrated inventory system with general time-varying demand and production and deterioration rates. *European Journal of Operational and Research*, 114, pp. 29-37.
21. Balkhi, Z., and Benkherouf, L. (1996). On the optimal replenishment schedule for an inventory system with deteriorating items and time-varying demand and production rates. *Computers & Industrial Engineering*, 30, pp. 823-829.
22. Banks, J., Carson, J., Nelson, B., and Nicol, D. (2000) *Discrete Event Systems Simulation*. 3rdedition. Prentice Hall, Englewood Cliffs, New Jersey.
23. Bartmann, D. and Bach, M. (2012) Inventory control: models and methods. *Springer Science & Business Media,* p. 318.
24. Barton, R., and Meckesheimer, M. (2006) Metamodel-based simulation optimization. *Handbooks in operations research and management science,* 13, pp. 535-574.
25. Batista, G. and Monard, M. (2002) A Study of K-Nearest Neighbour as an Imputation Method. *HIS, 87*, pp. 251-260.
26. Becerril-Arreola, R., Leng, M. and Parlar, M. (2013) Online retailers' promotional pricing, free-shipping threshold, and inventory decisions: A simulation-based analysis. *European Journal of Operational Research*, 230(2), pp. 272-283.

27. Bellman, R. (1957) *Dynamic Programming*. Princeton, Princeton University Press.
28. Bendoly, E. (2004) Integrated inventory pooling for firms servicing both on-line and store demand. *Computers & Operations Research,* 31(9), pp. 1465-1480.
29. Benjaafar, S., Gayon, J. and Tepe, S. (2010) Optimal control of a production–inventory system with customer impatience. *Operations Research Letters*, 38(4), pp. 267-272.
30. Benkherouf, L. and Balkhi, Z. (1997) On inventory model for deteriorating items and time-varying demand. *Mathematical Methods of Operations Research*, 45, pp. 221-233.
31. Bensoussan, A., Cakanyildirim, J., Minjarez-Sosa, A. and Sethi, S. (2008) Inventory Problems with Partially Observed Demands and Lost Sales. *Journal of Optimization Theory and Applications*, 136(3), pp. 321-340.
32. Beyer, D., Cheng, F., Sethi, S. and Taksar, M. (2010) *Markovian demand inventory models.* New York: Springer.
33. Bijvank, M. and Vis, I. (2011) Lost-sales inventory theory: A review. *European Journal of Operational Research*, 215(1), pp. 1-13.
34. Biles, W., Kleijnen, J., Van Beers, W. and Van Nieuwenhuyse, I. (2007) Kriging metamodeling in constrained simulation optimization: an explorative study. In: *2007 Winter Simulation Conference.* IEEE, pp. 355-362.
35. Bisong, E. (2019). *Matplotlib and Seaborn. Building Machine Learning and Deep Learning Models on Google Cloud Platform.* Apress, Berkeley, CA, pp. 151-165.
36. Blanco-Silva, F. (2013) *Learning SciPy for numerical and scientific computing*. Packt Publishing Ltd.
37. Blanning, R. (1975) The construction and implementation of metamodels, *Simulation*, 24, pp. 177-184.
38. Bollapragada, R. and Rao, U. (2006) Replenishment planning in discrete-time, capacitated, non-stationary, stochastic inventory systems. *IIE Transactions*, 38(7), pp. 605-617.
39. Bonney, M. and Jaber, M. (2011) Environmentally responsible inventory models: Non-classical models for a non-classical era. *International Journal of Production Economics*, 133(1), pp. 43-53.
40. Bottou, L. (2010) Large-scale machine learning with stochastic gradient descent. In: *Proceedings of COMPSTAT'2010*. Physica-Verlag HD, pp. 177-186.

41. Bushuev, M., Guiffrida, A., Jaber, M. and Khan, M. (2015) A review of inventory lot sizing review papers. *Management Research Review*, 38(3), pp. 283-298.
42. Byrne, M. (2013) How many times should a stochastic model be run? An approach based on confidence intervals. In: *Proceedings of the 12th International conference on cognitive modeling,* Ottawa.
43. Calinski, T. and Harabasz, J. (1974) A dendrite method for cluster analysis. *Communications in Statistics-theory and Methods*, 3(1), pp. 1-27.
44. Campos, G., Zimek, A., Sander, J., Campello, R., Micenkova, B., Schubert, E., Assent, I. and Houle, M. (2016) On the evaluation of unsupervised outlier detection: measures, datasets, and an empirical study. *Data Mining and Knowledge Discovery*, 30(4), pp. 891-927.
45. Can, B. and Heavey, C. (2012) A comparison of genetic programming and artificial neural networks in metamodeling of discrete-event simulation models. *Computers & Operations Research*, 39, pp. 424-436.
46. Carson, Y., and Anu M. (1997) Simulation optimization: methods and applications. In: *Proceedings of the 29th conference on Winter simulation*. IEEE Computer Society, pp. 317-331.
47. Caruana, R. and Schaffer, J. (1988) Representation and hidden bias: Gray vs. binary coding for genetic algorithms. In: *Proceedings of the 5th International Conference on Machine Learning.* San Mateo, CA: Morgan Kaufmann, pp. 153-161.
48. Cawley, G. and Talbot, N. L. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11, pp. 2079-2107.
49. Chang, C. (2004). An EOQ model with deteriorating items under inflation when supplier credits linked to order quantity. *International Journal of Production Economics*, 88, pp. 307-316.
50. Chen, F., and Song, J. (2001) Optimal Policies for Multi-echelon Inventory Problems with Markov-modulated Demand, *Operations Research*, 49(2), pp. 226-234.
51. Chen, J. (1998) An inventory model for deteriorating items with time-proportional demand and shortages under inflation and time discounting. *International Journal of Production Economics*, 55, pp. 21-30.

52. Chen, J. and Shao, J. (2001) Jackknife variance estimation for nearest-neighbor imputation. *Journal of the American Statistical Association*, 96(453), pp. 260-269.
53. Chen, X., Wang, G., Zhou, W., Chang, S. and Sun, S. (2006). Efficient sigmoid function for neural networks based FPGA design. In: *Proceedings of the International Conference on Intelligent Computing*. Springer, Berlin, Heidelberg, pp. 672-677.
54. Chen, Y., Pekny, J. and Reklaitis, G. (2012) Integrated planning and optimization of clinical trial supply chain system with risk pooling. *Industrial & engineering chemistry research*, 52(1), pp.152-165.
55. Chern, M., Yang, H., Teng, J., and Papachristos, S. (2008) Prtial backlogging inventory lot-size models for deteriorating items with fluctuating demand under inflation. *European Journal of Operational Research*, 191, pp. 127-141.
56. Chollet, F. (2018) *Keras: The python deep learning library.* Astrophysics Source Code Library.
57. Cimen, M. and Kirkbride, C. (2017) Approximate dynamic programming algorithms for multidimensional flexible production-inventory problems. *International Journal of Production Research*, 55(7), pp. 2034-2050.
58. Clark, A and Scarf, H. (1960) Optimal policies for a multi-echelon inventory problem. *Management Science*, pp. 475-490.
59. Clevert, D., Unterthiner, T. and Hochreiter, S. (2015) Fast and accurate deep network learning by exponential linear units (elus). In: *arXiv preprint arXiv:1511.07289.*
60. Comaniciu, D. and Meer, P. (2002) Mean shift: A robust approach toward feature space analysis. *IEEE Transactions on pattern analysis and machine intelligence*, 24(5), pp. 603-619.
61. Cybenko, G. (1989) Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4), pp. 303-314.
62. Darwin, C. (1859) *On the Origin of Species by Means of Natural Selection or the Preservation of Favoured Races in the Struggle for Life*. London, Murray.
63. Datta, T. and Pal, A. (1990) Deterministic inventory systems for deteriorating items with inventory level-dependent demand rate and shortages. *Opsearch*, 27, pp. 213-224.
64. Davis, L. (1991) Hybridization and numerical representation The Handbook of Genetic Algorithms. *New York: Van Nostrand Reinhold*, pp. 61-71.

65. De Jong, K. (1975) *An Analysis of the Behavior of a Class of Genetic Adaptive Systems.* PhD Dissertation, University of Michigan.
66. Deb, K. and Agrawal, R. (1995) Simulated binary crossover for continuous search space. *Complex systems*, 9(2), pp. 115-148.
67. Desai, V. (2007) Evaluating Clustering Methods for Multi-Echelon (r, Q) Policy Setting. In: *IIE Annual Conference. Proceedings*, Institute of Industrial and Systems Engineers, p. 352.
68. Ding, H., Benyoucef, L. and Xie, X. (2009) Stochastic multi-objective production-distribution network design using simulation-based optimization. *International Journal of Production Research*, 47(2), pp. 479-505.
69. Domingos, P. (2015) *The master algorithm: How the quest for the ultimate learning machine will remake our world*. Basic Books, p. 217.
70. Domschke, W., Drexl, A., Klein, R. and Scholl, A. (2015) *Einführung in Operations Research.* 9th Ed., Berlin, Heidelberg: Springer Gabler.
71. Dozat, T. (2016) Incorporating Nesterov Momentum into Adam. *ICLR Workshop*, pp. 105-119.
72. Duan, Q. and Liao, T. (2013) Optimization of replenishment policies for decentralized and centralized capacitated supply chains under various demands. *International Journal of Production Economics*, pp. 194-204.
73. Dubois, G. (2018) Modeling and Simulation: Challenges and Best Practices for Industry. CRC Press.
74. Duchi, J., Hazan, E. and Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, *12*, pp. 2121-2159.
75. DynamicAction and IHL-group (2015) *Research Study: Retailers and the Ghost Economy $1.75 Trillion Reasons to be Afraid.*
76. Edgeworth, F. (1888) The Mathematical Theory of Banking, *Journal of Royal Statistical Society*, 51(1), pp. 113-127.
77. Egas, C. and Masel, D. (2010) Determining warehouse storage location assignments using clustering analysis. In: *11th IMHRC Proceedings on Progress in Material Handling Research*, Milwaukee, Wisconsin, USA, pp. 22-33.
78. Elsken, T., Metzen, J. and Hutter, F. (2019) Neural Architecture Search: A Survey. *Journal of Machine Learning Research*, 20(55), pp. 1-21.

79. Ernst, R. and Cohen, M. (1990) Operations related groups (ORGs): a clustering procedure for production/inventory systems. *Journal of Operations Management*, 9(4), pp. 574-598.
80. Feurer, M. and Hutter, F. (2019) Hyperparameter optimization. *Journal of Machine Learning Research*, 12, pp. 3-38.
81. Floridi, L. (2019) Establishing the rules for building trustworthy AI. *Nature Machine Intelligence,* 1(6), p. 261.
82. Forrester, J. (1961) *Industrial Dynamics*. Cambridge, MA, MIT Press, 36, pp. 37-66.
83. Forrester, J. (1973) *World Dynamics*. Cambridge, MA, MIT Press.
84. Fortin, F., Rainville, F., Gardner, M., Parizeau, M., and Gagne, C. (2012). DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, *13*, pp. 2171-2175.
85. Frank, G. (1953) U.S. Patent No. 2,632,058. *Patent and Trademark Office*, Washington, DC: U.S.
86. Fu, M. and Hill, D (1997) Optimization of discrete event systems via simultaneous perturbation stochastic approximation. *IIE Transactions*, 29, pp. 233-243.
87. Gallino, S., Moreno, A., and Stamatopoulos, I. (2016) Channel integration, sales dispersion, and inventory management. *Management Science*, 63(9), pp. 2813-2831.
88. Geman, S., Bienenstock, E. and Doursat, R. (1992) Neural networks and the bias/variance dilemma, *Neural Computation,* 4, pp. 1–58.
89. Ghare, P. and Schrader, F. (1963) A model for exponentially decaying inventories. *Journal of International Engineering*, 15, 238-243.
90. Goldberg D. (1989) *Genetic Algorithms in Search, Optimization, and Machine Learning*. Reading, MA: Addison-Wesley.
91. Goldberg, D. and Deb, K. (1991) A comparative analysis of selection schemes used in genetic algorithms *Foundations of Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann, pp. 69-93.
92. Goldberg, D. and Holland, J. (1988) Genetic algorithms and machine learning. *Machine learning*, 3(2), pp. 95-99.
93. Goldfeld, S. and Quandt, R. (1965). Some tests for homoscedasticity. *Journal of the American statistical Association, 60*(310), pp. 539-547.
94. Goodfellow, I., Bengio, Y. and Courville, A. (2016). *Deep learning.* MIT press.

95. Gosavi, A. (2015) *Simulation-based optimization.* Berlin: Springer.
96. Goyal, S. (1985) Economic order quantity under conditions of permissible delay in payments. *Journal of Operational Research Society*, 36, pp. 335-338.
97. Grassberger, P. and Procaccia, I. (1988) Measuring the strangeness of strange attractors. *Physica: Nonlinear Phenomena*, 9(1-2), pp. 189-208.
98. Grefenstette, J. (1986) Optimization of control parameters for genetic algorithms. *IEEE Transportation Systems,* SMC-16, pp. 122-128.
99. Grossberg, S. (1988) *Neural Networks and Natural Intelligence,* Cambridge, MA: MIT Press.
100. Gruen, T. and Corsten, D. (2007) *A comprehensive guide to retail out-of-stock reduction in the fast-moving consumer goods industry.* Research study, P&D, pp. 848-866.
101. Gruen, T., Corsten, D. and Bharadwaj, S. (2002) *Retail out-of-stocks: A worldwide examination of extent causes and consumer responses.* Grocery Manufacturers of America.
102. Hadley, G., Whitin, T. (1963) *Analysis of Inventory Systems.* Prentice-Hall, Englewood Cliffs, New York.
103. Hancock, P. (1994) An empirical comparison of selection methods in evolutionary algorithms. *Evolutionary Computing,* Berlin, Springer, pp. 80-94
104. Hanne, T., and Dornberger, R. (2017) *Computational Intelligence in Logistics and Supply Chain Management*. Springer, Cham, pp. 13-41.
105. Hanin, B. (2018) Approximating Continuous Functions by ReLU Nets of Minimal Width. *arXiv preprint arXiv:1710.11278.*
106. Hara, K., Saito, D. and Shouno, H. (2015). Analysis of function of rectified linear unit used in deep learning. In: *2015 International Joint Conference on Neural Networks,* IEEE, pp. 1-8.
107. Hariga, M. (1993) The inventory replenishment problem with a linear trend in demand. *Computers Industrial Engineering*, 24, pp. 143-150.
108. Harris, F. (1913) *Operations and Cost.* Factory Management Series, Chicago: A.W. Shaw and Company.
109. Hastie, T., Tibshirani, R., Friedman, J. and Franklin, J. (2005) The elements of statistical learning: data mining, inference and prediction. *The Mathematical Intelligencer, 27*(2), pp. 83-85.
110. Haykin, S. (2009) *Neural networks and learning machines.* New York, Prentice Hall.

111. Henriksen, J. (1977) An improved events list algorithm. In: *the 9th conference on Winter Simulation Conference, 2*, pp. 546-557.
112. Hexa Research (2018) Report: Online grocery market expected to reach $26.9B by 2025 [online] Available at: https://www.grocerydive.com/news/report-online-grocery-market-expected-to-reach-269b-by-2025/543134/ [Accessed 1 Feb. 2020].
113. Hochmuth, C. and Kochel, P. (2012) How to order and transship in multi-location inventory systems: The simulation optimization approach. *International Journal of Production Economics*, 140, pp. 646-654.
114. Holland, J. (1962) Outline for a logical theory of adaptive systems. *ACM,* 9, pp. 297-314.
115. Holland, J. (1967) Nonlinear environments permitting efficient adaptation. *Computer and Information Science,* 2, pp. 111-122.
116. Holland, J. (1969) Adaptive plans optimal for payoff-only environments. In: *2nd Hawaii International Conference on System Sciences,* pp. 917-920.
117. Holland, J. (1971) Processing and processors for schemata. *Associative information processing.* New York, Elsevier, pp. 127-146.
118. Holland, J. (1973) Genetic algorithms and the optimal allocation of trials *SIAM J. Computing,* 2, pp. 88-105.
119. Holland, J. (1975) *Adaptation in Natural and Artificial Systems.* Ann Arbor, MI, Michigan Press.
120. Holland, J. (1992) *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence.* MIT press.
121. Hong, L. and Nelson, B. (2009) A brief introduction to optimization via simulation. In: *Proceedings of the 2009 Winter Simulation Conference*, pp. 314-322.
122. Hornik, K. (1991) Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2), pp. 251-257.
123. Hornik, K. (1993) Some new results on neural network approximation. *Neural networks*, 6(8), pp. 1069-1072.
124. Hu, W., Kim, S., Banerjee, A. (2009) An inventory model with partial backordering and unit backorder cost linearly increasing with the waiting time. *European Journal of Operational Research,* 197, pp. 581-587.

125. Hunter, J. (2007) Matplotlib: A 2D graphics environment. *Computing in science & engineering*, *9*(3), p. 90.
126. Hurlimann, T. (2007) *Index notation in mathematics and modelling language LPL: theory and exercises*. Department of Informatics University of Fribourg.
127. Hutter, F., Kotthoff, L., and Vanschoren, J. (2019) *Automatic Machine Learning: Methods, Systems, Challenges*. Berlin, Springer.
128. Iglehart, D., and Karlin, S. (1962) Optimal Policy for Dynamic Inventory Process Inventory Process With Nonstationary Stochastic Demands. Studies. In: *Applied Probability and Management Science*, Stanford University Press, pp. 81-99.
129. IHL-group and Buzek G. (2015) *Research Study:* We Lost Australia! Retail's $1.1 Trillion Inventory Distortion Problem.
130. Iverson, K. (1962) A programming language. In: *Proceedings of the spring joint computer conference ACM*, pp. 345-351.
131. Jackson, I. (2019a) Simulation-Optimisation Approach to Stochastic Inventory Control with Perishability. *Information Technology & Management Science (RTU Publishing House)*, *p. 22*.
132. Jackson, I. (2019b) Neuroevolutionary Approach to Metamodeling of Production-Inventory Systems with Lost-Sales and Markovian Demand. In: *Proceedings of the 19th International Conference on Reliability and Statistics in Transportation and Communication.* Springer, Cham.
133. Jackson, I. (2020) *GitHub repository "metainventory"* [online] Available at: https://github.com/Jackil1993/metainventory [Accessed 1 Feb. 2020].
134. Jackson, I., Tolujevs, J. and Reggelin, T. (2018a) The Combination of Discrete-Event Simulation and Genetic Algorithm for Solving the Stochastic Multi-Product Inventory Optimization Problem. *Transport and Telecommunication Journal*, 19(3), pp. 233-243.
135. Jackson, I., Avdeikins, A. and Tolujevs, J. (2018b) Unsupervised Learning-Based Stock Keeping Units Segmentation. In: *Proceedings of the 18th International Conference on Reliability and Statistics in Transportation and Communication.* Springer, Cham, pp. 603-612.

136. Jackson, I. and Tolujevs, J. (2019) The Discrete-Event Approach to Simulate Stochastic Multi-Product (Q, r) Inventory Control Systems. *Information Modelling and Knowledge Bases XXX*, 312, pp. 32-39.
137. Jackson, I. and Grakovski, A. (2019) Combining LSTM Artificial Recurrent Neural Networks and Fractal Analysis for Inventory Dynamics Prediction. In: *Proceedings of the 19th International Conference on Reliability and Statistics in Transportation and Communication.* Springer, Cham.
138. Jackson, I., Tolujevs, J., Lang, S., and Kegenbekov, Z. (2019) Metamodeling of Inventory Control Simulations based on a Multilayer Perceptron. *Transport and Telecommunication Journal*, 20(3), pp. 251-259.
139. Jad, F. and Owayjan, M. (2017) ERP Neural Network Inventory Control. *Procedia Computer Science*, 114, pp. 288-295.
140. Jalali, H. and Nieuwenhuyse, I. (2015) Simulation optimization in inventory replenishment: a classification. *IIE Transactions*, 47(11), pp.1217-1235.
141. Johansen, S., Thorstenson, A. (1996) Optimal (r, Q) inventory policies with Poisson demands and lost sales: discounted and undiscounted cases. *International Journal of Production Economics,* 46, pp. 359-371.
142. Johansen, S., Thorstenson, A. (2004) *The (r, q) Policy for the Lost-sales Inventory System When More than One Order May Be Outstanding*. Technical Report, Aarhus School of Business.
143. Jonsson, P. and Wohlin, C. (2004) An evaluation of k-nearest neighbour imputation using likert data. In: *10th International Symposium on Software Metrics*, pp. 108-118.
144. Juan, A., Faulin, J., Grasman, S., Rabe, M. and Figueira, G. (2015) A review of simheuristics: Extending metaheuristics to deal with stochastic combinatorial optimization problems. *Operations Research Perspectives*, 2, pp. 62-72.
145. Kalpakam, S., Arivarignan, G. (1989) A lost sales inventory system in a random environment. *Stochastic Analysis and Applications*, 7, pp. 367-385.
146. Kapuscinski, R. and Tayur, S. (1998) A capacitated production-inventory model with periodic demand. *Operations Research*, 46(6), pp. 899-911.
147. Karlin, S. (1960) Dynamic Inventory Policy with Varying Stochastic Demands. *Management Science*, 6(3), pp. 231-258.

148. Karlin, S., and Fabens, A. (1960) The (s, S) Inventory Model under Markovian Demand Process. *Mathematical Methods in the Social Sciences*, Stanford, CA, Stanford University Press, pp. 159-175.
149. Kerlirzin, P., and Vallet, F. (1993) Robustness in multilayer perceptrons, *Neural Computation*, 5, pp. 473-482.
150. Khanlarzade, N., Yegane, B., Kamalabadi, I. and Farughi, H. (2014) Inventory control with deteriorating items: A state-of-the-art literature review. *International Journal of Industrial Engineering Computations*, 5(2), pp.179-198.
151. Khmelnitsky, E. and Gerchak, Y. (2002) Optimal control approach to production systems with inventory-level-dependent demand. *Transactions on Automatic Control*, IEEE, 47(2), pp. 289-292.
152. Kingma, D., and Ba, J. (2014) Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.
153. Kleijnen, J. (2008) *Design and Analysis of Simulation Experiments.* Springer, New York.
154. Koulouriotis, D., Xanthopoulos, A., and Tourassis, V. (2010) Simulation optimization of pull control policies for serial manufacturing lines and assembly manufacturing systems using genetic algorithms. *International Journal of Production Research*, 48, pp. 2887-2912.
155. Kriesel, D. (2007). *A brief introduction on neural networks*.
156. Krose, B., van der Smagt, P., and Smagt, P. (1993) *An introduction to neural networks.*
157. Landi, A., Piaggi, P., Laurino, M. and Menicucci, D. (2010) Artificial neural networks for nonlinear regression and classification. In: *10th International Conference on Intelligent Systems Design and Applications*, IEEE, pp. 115-120.
158. Law, A. and Kelton, W. (2000) *Simulation modeling and analysis*. New York, McGraw-Hill.
159. Lechevalier, D., Hudak, S., Ak, R., Lee, Y. T., and Foufou, S (2015) A neural network meta-model and its application for manufacturing. In: *Proceedings of the IEEE International Conference on Big Data,* Santa Clara, USA.
160. Lehmacher, W., Betti, F., Beecher, P., Grotemeier, C., and Lorenzen, M. (2017) Impact of the Fourth Industrial Revolution on Supply Chains. In: *World Economic Forum*, Geneva, Switzerland, REF, pp. 311-329.
161. Lehmer, D. (1951) Mathematical methods in large-scale computing units. *Annual Computational Laboratory*, 26, Harvard University, pp. 141-146.

162. Liao, H., Tsai, C., and Su, T. (2000). An inventory model for deteriorating items under inflation when delay in payment is permissible. *International Journal of Production Economics*, 63, pp. 207-214
163. Liao, J., Huang, K., and Chung, K. (2012) Lot-sizing decisions for deteriorating items with two warehouses under an order-size-dependent trade credit. *International Journal of Production Economics*, 137(1), pp. 102-115.
164. Lin, Y., Shie, J. and Tsai, C. (2009) Using an artificial neural network prediction model to optimize work-in-process inventory level for wafer fabrication. *Expert Systems with Applications,* 36(2), pp. 3421-3427.
165. Liu, H., Simonyan, K., Vinyals, O., Fernando, C. and Kavukcuoglu, K. (2018) Hierarchical Representations for Efficient Architecture Search. In: *International Conference on Learning Representations*, pp. 47-55.
166. Loken, E., and Gelman, A. (2017) Measurement error and the replication crisis. *Science*, 355, pp. 584-585.
167. Luke, S. (2015) *Essentials of Metaheuristics. A Set of Undergraduate Lecture Notes.* Second Edition, 2.2, pp. 31-55.
168. Lukinskiy, V. and Lukinskiy, V. (2017) Models of Inventory Management in Multi-level Distribution Systems. In: *International Conference on Reliability and Statistics in Transportation and Communication,* Springer, Cham, pp. 313-319.
169. Mahata, G. (2012) An EPQ-based inventory model for exponentially deteriorating items under retailer partial trade credit policy in supply chain. *Expert systems with Applications*, 39(3), pp. 3537-3550.
170. Mahdavi, A., Wolfe-Adam, T. (2019) *Artificial Intelligence and Simulation in Business.* White Paper, AnyLogic company, pp. 3-32.
171. Mathias, K. and Whitley, L. (1994) Changing representations during search: A comparative study of delta coding. *Evolutionary Computation*, 2(3), pp. 249-278.
172. Mayr, E. (1988) *Toward a New Philosophy of Biology: Observations of an Evolutionist*. Cambridge, MA, Belknap.
173. McCulloch, W. and Pitts, W. (1943) A logical calculus of the ideas immanent in nervous activity. *Bulletin of Mathematical Biology,* 5(4), pp. 115-133.

174. McKinney, W. (2011). pandas: a foundational Python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, *14*.
175. Merkuryev, Y. (2012) The modelling and simulation of complex systems: Methodology and practice. An overview. *Information technology and management science*, 15(1), pp. 32-44.
176. Merkuryev, Y., Petuhova, J. and Buikis, M. (2004) Simulation based statistical analysis of the bullwhip effect in supply chains. In: *18th European Simulation Multiconference Networked Simulations and Simulaed Networks*, pp. 312-329.
177. Merkuryeva, G. (2004) Metamodelling for simulation applications in production and logistics. In: *Proceedings of the Sim-Serv Workshop: Roadmap of simulation in manufacturing and logistics,* pp. 1-8.
178. Michalewicz, Z. (1996) Evolution strategies and other methods. In: *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, Berlin, Heidelberg, pp. 159-177.
179. Miikkulainen, R., Liang, J., Meyerson, E., Rawal, A., Fink, D., Francon, O., Raju, B., Shahrzad, H., Navruzyan, A. Duffy, D. and Hodjat, B. (2017) Evolving Deep Neural Networks. *In: arXiv:1703.00548.*
180. Miller, B. and Goldberg D. (1995) Genetic algorithms, tournament selection, and the effects of noise. *Complex systems*, 9(3), pp. 193-212.
181. Minsky, M. and Papert, S. (1969) *Perceptrons*. MIT Press, Cambridge, Mass.
182. Mohamad, I. and Usman, D. (2013) Standardization and its effects on K-means clustering algorithm. *Research Journal of Applied Sciences, Engineering and Technology*, 6(17), pp. 3299-3303.
183. Murphy, K. (2012) *Machine learning: a probabilistic perspective*. MIT press.
184. Nezhad, A. and Mahlooji, H. (2014) An artificial neural network meta-model for constrained simulation optimization. *Journal of the Operational Research Society*, 65, pp. 1232-1244.
185. Olafsson, S. (2006) Metaheuristics. In: *Handbooks in Operations Research and Management Science*, 13, North Holland. Elsevier, pp. 633-654.
186. Ortega, M. and Lin, L. (2004) Control theory applications to the production–inventory problem: a review. *International Journal of Production Research*, 42(11), pp. 2303-2322.
187. Ouyang, L., Chuang, B. and Lin, Y. (2005) Periodic review inventory models with controllable lead time and lost sales rate reduction. *Journal of the Chinese Institute of Industrial Engineers*, 22, pp. 355-368.

188. Pagan, A. and Hall, A. (1983). Diagnostic tests as residual analysis. *Econometric Reviews*, *2*(2), pp. 159-218.
189. Pakkala, T. and Achary, K. (1992) A deterministic inventory model for deteriorating items with two warehouse and finite replenishment rate. *European Journal of Operational Research*, 57, pp. 71-76.
190. Pandey, G. and Dukkipati, A. (2014) To go deep or wide in learning?. In: *arXiv preprint arXiv:1402.5634*.
191. Papachristos, S. and Skouri, K. (2000) An optimal replenishment policy for deteriorating items with time-varying demand and partial exponential type backlogging. *Operations Research Letters*, 27(4), pp. 175-184.
192. Pasupathy, R. and Ghosh, S. (2013) Simulation optimization: A concise overview and implementation guide. In: Operations Research, 7, INFORMS, pp. 122-150.
193. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V. and Vanderplas, J. (2011) Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12, pp. 2825-2830.
194. Peirleitner, A., Altendorfer, K. and Felberbauer, T. (2016) December. A simulation approach for multi-stage supply chain optimization to analyze real world transportation effects. In: *2016 Winter Simulation Conference,* IEEE, pp. 2272-2283.
195. Pitts, S., Blitstein, J., Gustafson, A. and Niculescu, M. (2018) Online grocery shopping: promise and pitfalls for healthier food and beverage purchases. *Public health nutrition*, 21, pp. 3360-3376.
196. Powell, J. and Bradford, J. (2000) Targeting intelligence gathering in a dynamic competitive environment. *International Journal of Information Management*, 20, pp. 181-195.
197. Powell, W (2006) What you should know about approximate dynamic programming. *Naval Research Logistics*, 56(3), pp. 239-249.
198. Prestwich, S. D., Tarim, S. A., Rossi, R. and Hnich, B. (2012) A neuroevolutionary approach to stochastic inventory control in multi-echelon systems. *International Journal of Production Research,* 50, pp. 2150-2160.
199. ProModel (2010) *SimRunner User Guide.* ProModel Corporation.

200. Radcliffe, N. (1991) Forma analysis and random respectful recombination. In: *4th International Conference on Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann, pp 222-229.
201. Ravichandran, N. (1984) Note on (s, S) inventory policy. *IIE Transactions*, 16, pp. 387-390.
202. Rawal, A., Rajagopalan, P. and Miikkulainen, R. (2010) Constructing competitive and cooperative agent behavior using coevolution. In: *Proceedings of the 2010 IEEE conference on computational intelligence and games,* IEEE, pp. 107-114.
203. Razali, N. and Geraghty, J. (2011). Genetic algorithm performance with different selection strategies in solving TSP. In: *Proceedings of the world congress on engineering*, Hong Kong: International Association of Engineers, 2, pp. 1-6.
204. Razali, N. and Wah, Y. (2011) Power comparisons of shapiro-wilk, kolmogorov-smirnov, lilliefors and anderson-darling tests. *Journal of statistical modeling and analytics*, 2(1), pp. 21-33.
205. Real, E. Moore, S., Selle, A., Saxena, S., Suematsu, Y., Le, Q. and Kurakin A. (2017) Large-scale evolution of image classifiers. *International Conference on Machine Learning*, pp. 111-124.
206. Real, E., Aggarwal, A., Huang, Y. and Le, Q. (2019) Aging Evolution for Image Classifier Architecture Search. In: *AAAI Conference on Artificial Intelligence,* pp. 211-227.
207. Richman, J. and Moorman, J. (2000) Physiological time-series analysis using approximate entropy and sample entropy. *American Journal of Physiology-Heart and Circulatory Physiology*, 278(6), pp. 2039-2049.
208. Risi, S., Lehman, J., D'Ambrosio, D., Hall, R. and Stanley, K. (2012) Combining search-based procedural content generation and social gaming in the petalz video game. In: *18th Artificial Intelligence and Interactive Digital Entertainment Conference*, pp. 111-125.
209. Rojas, R. (1996) The backpropagation algorithm. In: *Neural networks.* Springer, Berlin, Heidelberg, pp. 149-182.
210. Rosenblatt, F. (1958) The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, *65*(6), p. 386.
211. Rossini, P. (2000) Using expert systems and artificial intelligence for real estate forecasting. In: 16th Annual Pacific-Rim Real Estate Society Conference. Sydney, Australia, pp. 24-27.

212. Rousseeuw, P. (1987) Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20, pp. 53-65.
213. Rumelhart, D., Hinton, G., and Williams, R. (1986) Learning representations by back-propagating errors. Nature, 323, pp. 533-536.
214. Ryberg, A. (2013) *Metamodel-Based Design Optimization: A Multidisciplinary Approach for Automotive Structures*. Linkoping University Electronic Press.
215. Sana, S., Goyal, S., and Chaudhuri, K. (2004) A production-inventory model for a deteriorating item with trended demand and shortages. *European Journal of Operational Research*, 157, pp. 357-371.
216. Sarimveis, H., Patrinos, P., Tarantilis, C. and Kiranoudis, C. (2008) Dynamic modeling and control of supply chain systems: A review. *Computers & operations research*, 35(11), pp. 3530-3561.
217. Scarf, H. (1960) The Optimality of (s, S) Policies in the Dynamic Inventory Problem. *Mathematical Methods in the Social Sciences.* Stanford, CA: Stanford University Press, pp. 196-202.
218. Schenk, M., Tolujew, J. and Reggelin, T. (2010) A mesoscopic approach to the simulation of logistics systems. In: *International Heinz Nixdorf Symposium*. Springer, Berlin, Heidelberg, pp. 15-25.
219. Schmidt-Hieber, J. (2017) Nonparametric regression using deep neural networks with ReLU activation function. In: *arXiv preprint arXiv:1708.06633*.
220. Seabold, S. and Perktold, J. (2010) Statsmodels: Econometric and statistical modeling with python. In: *Proceedings of the 9th Python in Science Conference*, 57, p. 61.
221. Sethi, S. and Thompson, G. (1981) Dynamic optimal control models in advertising: recent developments. *Management Science*, 40, pp. 195-226.
222. Sethi, S., and Cheng, F. (1997) Optimality of (s, S) Policies in Inventory Models with Markovian Demand. *Operations Research*, 45, pp. 931-939.
223. Shreve, S. (1976) *Abbreviated proof in the lost sales case. Dynamic Programming and Stochastic Control.* New York: Academic Press, pp. 105-106.
224. Silver, E. (1981) Operations Research in Inventory Management: a Review and Critique. *Operations Research*, 29, pp. 628-645.

225. Simao, H. and Powell, W. (2009) Approximate dynamic programming for management of high-value spare parts. *Journal of Manufacturing Technology Management*, 20(2), pp. 147-160.
226. Simchi-Levi, D. and Zhao Y. (2003) The value of information sharing in a two-stage supply chain with production capacity constraints. *Naval Research Logistics*, 50, pp. 888-916.
227. Simon, H. (1952) On the application of servomechanism theory in the study of production control. *Econometrica*, 20, pp. 247-268.
228. Srinivasan, M. and Moon, Y. (1999) A comprehensive clustering algorithm for strategic analysis of supply chain networks. *Computers & industrial engineering*, 36(3), pp. 615-633.
229. Stanley, K. and Miikkulainen, R. (2002) Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10, pp. 99–127.
230. Stanley, K., D'Ambrosio, D. and Gauci, J. (2009) A hypercube-based encoding for evolving large-scale neural networks. *Artificial Life*, 15(2), pp. 185-212.
231. Sutskever, I., Martens, J., Dahl, G. and Hinton, G. (2013) On the importance of initialization and momentum in deep learning. In: *International conference on machine learning*, pp. 1139-1147.
232. Taleizadeh, A., Zarei, H. and Sarker, B. (2017) An optimal control of inventory under probablistic replenishment intervals and known price increase. *European Journal of Operational Research*, 257(3), pp. 777-791.
233. The Statistics Portal (2018) US consumers: online grocery shopping – statistics & facts. http://www.statista.com/topics/1915/us-consumers-online-grocery-shopping/ (accessed on 01.01.2020).
234. Tocher, K., and Owen, D. (1960) The automatic programming of simulations. In: *Proceedings of the 2nd International Conference on Operational Research*. London: The English Universities Press Ltd.
235. Tolujevs, J., Lorenz, P., Beier, D. and Schriber, T. (1998) Assessment of simulation models based on trace-file analysis: a metamodeling approach. In: *Proceedings of the Winter Simulation Conference.* IEEE, pp. 443-450.
236. Tran, T.N., Drab, K. and Daszykowski, M. (2013) Revised DBSCAN algorithm to cluster data with dense adjacent clusters. *Chemometrics and Intelligent Laboratory Systems*, 120, pp. 92-96.

237. Tsai, S. and Zheng, Y. (2013) A simulation optimization approach for a two-echelon inventory system with service level constraints. *European Journal of Operational Research*, 229, pp. 364-374.
238. Van Beek, P., Bremer, A. and Putten, C. (1985) Design and optimization of multi-echelon assembly networks: Savings and potentialities. *European Journal of Operational Research*, 19, pp. 57-67.
239. Van Der Walt, S., Colbert, S. C. and Varoquaux, G. (2011) The NumPy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, *13*(2), p. 22.
240. Van Rijsbergen, C. (2004) *The geometry of information retrieval*. Cambridge University Press.
241. Verhoef, P. and Sloot, L. (2006) Out-of-stock: Reactions, antecedents, management solutions, and a future perspective. In: *the 21st Century: Current and Future Trends.* Springer Publishers, Philadelphia, Pennsylvania, USA, pp. 239-253.
242. Verleysen, M. and Damien F. (2005) The curse of dimensionality in data mining and time series prediction. In: *International Work-Conference on Artificial Neural Networks*. Springer, Berlin, Heidelberg, pp. 313-321.
243. Vickson, R. (1982) Optimal control of production sequences: a continuous parameter analysis. *Operations Research*, 30, pp. 659–679.
244. Wagner, H., and Whitin, T. (1958) Dynamic Version of the Economic Lot Size Model. *Management Science*, 50(12), pp. 1770-1774.
245. Wee, H. (1993) Economic production lot-size model for deteriorating items with partial backordering. *Computer & Industrial Engineering*, 24, pp. 449-458.
246. Werbos, P. (1974) *Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences.* PhD thesis, Harvard University.
247. Weron, R. (2002) Estimating long-range dependence: finite sample properties and confidence intervals. *Physica: Statistical Mechanics and its Applications*, 312(2), pp. 285-299.
248. Whitehead, A. (2017) *An introduction to mathematics.* Courier Dover Publications.
249. Whitin, T. (1957) *Theory of Inventory Management.* Princeton University Press, Princeton, NJ.

250. Williams, C. and Patuwo, B. (1999) A perishable inventory model with positive order lead times. *European Journal of Operational Research.* 116, pp. 352-373.
251. Williams, E. and Crossley, W. (1998) Empirically-derived population size and mutation rate guidelines for a genetic algorithm with uniform crossover. In: *Soft computing in engineering design and manufacturing*. Springer, London, pp. 163-172.
252. Winskel, G. (2010) Set theory for computer science. *Unpublished lecture notes*, pp. 17-38.
253. Wolpert, D. (1996). The lack of a priori distinctions between learning algorithms. *Neural computation*, 8(7), pp. 1341-1390.
254. Wolpert, D. and Macready, W. (1997) No free lunch theorems for optimization. *Transactions on evolutionary computation*, IEEE, 1, pp. 67-82.
255. Wolstenholme, E. (1982) Managing international mining - case study concerning complex ownership systems. *European Journal of Operational Research*, 9, pp. 133-143.
256. Wright, A. (1991) Genetic algorithms for real parameter optimization. *Foundations of Genetic Algorithms.* San Mateo, CA: Morgan Kaufmann, pp. 205-218.
257. Wu, K. (2002) EOQ inventory model for items with Weibull distribution deterioration and time-varying demand and partial backlogging. *International Journal of systems Science*, 33, pp. 323-329.
258. Xie, S., Zheng, H., Liu, C. and Lin, L. (2019) SNAS: stochastic neural architecture search. In: *International Conference on Learning Representations*, pp. 319-328.
259. Xu, H. and Wang, H. (1992). Optimal inventory policy for perishable items with time proportional demand. *IIE Transactions*, 24, pp. 105-110.
260. Yang, C. and Nguyen, T. (2016) Constrained clustering method for class-based storage location assignment in warehouse. *Industrial Management & Data Systems*, 116(4), pp. 667-689.
261. Yang, H. (2005). A comparison among various partial backlogging inventory lot-size models for deteriorating items on the basis of maximum profit. *International Journal of Production Economics*, 96, pp. 119-128.
262. Yang, X. (2010) *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, p. 39.
263. Yatskiv, I. and Gusarova, L. (2005) The methods of cluster analysis results validation. *Transport and Telecommunication*, 6(1), pp. 217-229.

264. Yentes, J., Hunt, N., Schmid, K., Kaipust, J., McGrath, D. and Stergiou, N. (2013) The appropriate use of approximate entropy and sample entropy with short data sets. *Annals of biomedical engineering*, 41(2), pp. 349-365.
265. Zahedi-Hosseini, F. (2018) Modeling and simulation for the joint maintenance-inventory optimization of production systems. In: *Proceedings of the 2018 Winter Simulation Conference*, IEEE Press, pp. 3264-3274.
266. Zamanlooy, B. and Mirhassani, M. (2013). Efficient VLSI implementation of neural networks with hyperbolic tangent activation function. *IEEE Transactions on Very Large-Scale Integration (VLSI) Systems*, *22*(1), pp. 39-48.
267. Zeballos, A., Seifert, R. and Protopappa-Sieke, M. (2013) Single product, finite horizon, periodic review inventory model with working capital requirements and short-term debt. *Computers & Operations Research*, 40(12), pp. 2940-2949.
268. Zeiler, M. (2012). ADADELTA: an adaptive learning rate method. In: *arXiv preprint arXiv:1212.5701*.
269. Zeng, X., Zhang, Z. and Wang, D. (2016) AdaMax Online Training for Speech Recognition.
270. Zhang, G. and Li, H. (2018) Effectiveness of Scaled Exponentially-Regularized Linear Units (SERLUs). In: *arXiv preprint arXiv:1807.10117*.
271. Zheng, H., Yang, Z., Liu, W., Liang, J. and Li, Y. (2015) Improving deep neural networks using softplus units. In: *International Joint Conference on Neural Networks (IJCNN)*, IEEE, pp. 1-4.
272. Zipkin, P. (2008) Old and New Methods for Lost-Sales Inventory Systems. *Operations Research*, 56(5), pp. 1256-1263.
273. Zloba, E. and Yatskiv, I. (2002) Statistical methods of reproducing of missing data. *Computer Modelling & New Technologies*, 6(1), pp. 51-61.
274. Zoph, B. and Le, Q. (2016). Neural architecture search with reinforcement learning. In: *arXiv preprint arXiv:1611.01578*.
275. Zvirgzdina, B. and Tolujew, J. (2016) Experience in Optimization of Discrete Rate Models Using ExtendSim Optimizer. In: *9th International Doctoral Students Workshop on Logistics, June, 2016.Magdeburg*, Otto von Guericke University, pp. 95-101.

APPENDICES

Appendix 1. Source-code of the simulation of the stochastic multiproduct inventory control system with perishability

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

# create a .txt file to write down all the discrete events for further
tracing
file = open("protocol.txt","w")

# a static class containing all the utilized random number generators

class Generator:
    def normal(self, mu, sigma):
        return np.random.normal(mu, sigma)

    def exponential(self, betta):
        return np.random.exponential(betta)

# the "Warehouse" includes the parameters shared by all the operable
products

class Warehouse:
    def __init__(self, adjustments, parameters, costs, strategy,
total_capacity=300):
        self.total_capacity = total_capacity
        self.free_capacity = total_capacity
        self.time = 0.0
        self.products = []

        self.to_report = bool(adjustments[0][1])

        # we generate the operable products
        for product in range(len(parameters)):
            self.products.append(Product(product,
adjustments[product], parameters[product], costs[product],
strategy[product], total_capacity))

    def check_free_capacity(self):
        used = 0.0
        for product in range(len(self.products)):
            used += sum(self.products[product].lots)
        self.free_capacity = self.total_capacity - used
```

```
        if self.free_capacity < 0.0:
            self.free_capacity = 0.0
        if self.to_report is True:
            file.write('\nFree capacity =
{}'.format(round(self.free_capacity, 2)))

    def advance_time(self):
        demands = list(map(lambda x: x.next_demand, self.products))
        replenishments = list(map(lambda x: x.next_rep,
self.products))
        self.check_free_capacity()
        if min(demands) <= min(replenishments):
            self.time =
self.products[np.argmin(demands)].handle_demand(self.time)
        else:
            self.time =
self.products[np.argmin(replenishments)].handle_replenishment(self.tim
e, self.free_capacity)

class Product:
    def __init__(self, id, adjustments, parameters, costs, strategy,
total_capacity):
        self.g = Generator()
        self.id = id
        self.interarrivals = parameters[0]
        self.demand = parameters[1]
        self.replenishment_lead = parameters[2]
        self.expiry = parameters[3]
        self.lots = [strategy[0]]
        self.expiry_date = [self.g.normal(parameters[3][0],
parameters[3][1])]
        self.status = False
        self.next_demand = self.g.exponential(self.interarrivals)
        self.next_rep = float('inf')
        self.next_event = self.next_demand
        self.reorder_point = strategy[1]
        self.reorder_size = strategy[0]
        #counters
        self.backorders = 0.0
        self.overflows = 0.0
        self.expired = 0.0
        self.purchase_prise = costs[0] #includes delivery
        self.sales_prise = costs[1]
        self.total_capacity = total_capacity
        self.purchase_prise = self.purchase_prise if self.reorder_size
< 0.5*self.total_capacity else 0.7*self.purchase_prise
```

```
        self.handling_cost = costs[2]
        self.backorder_fee = costs[3]
        self.overflow_fee = costs[4]
        self.recycle_fee = costs[5]
        self.income = 0.0
        self.costs = 0.0
        self.to_plot = bool(adjustments[0])
        self.to_report = bool(adjustments[1])
        if self.to_plot == True:
            self.stats = Statistics(self.reorder_point,
self.reorder_size)


    def check_expiry(self, time):
        for lot in range(len(self.expiry_date)):
            if self.expiry_date[lot] <= 0.0 and self.lots[lot] > 0:
                if self.to_report is True:
                    file.write('\nThe lot № {} of product {} is
expired. {} pieces will be recycled'.format(lot, self.id,

round(float(self.lots[lot]), 2)))
                try:
                    self.stats.expires.append(time)
                    self.stats.expires_time.append(sum(self.lots))
                except AttributeError:
                    pass
                if len(self.lots) > 1:
                    self.expiry_date.pop(lot)
                    tmp = self.lots.pop(lot)
                    self.expired += tmp
                    self.costs += self.recycle_fee * tmp
                else:
                    self.expiry_date[0] = 0
                    self.expired += self.lots[0]
                    self.costs += self.recycle_fee * self.lots[0]
                    self.lots[0] = 0
                break

    def handle_demand(self, time):
        to_handle = abs(self.g.normal(self.demand[0], self.demand[1]))
        self.expiry_date = list(map(lambda x: x - (self.next_demand -
time), self.expiry_date))
        self.check_expiry(time)
        self.costs += (self.next_demand -
time)*sum(self.lots)*self.handling_cost
        time = self.next_demand
```

```
        self.next_demand = time +
self.g.exponential(self.interarrivals)
        if self.to_report is True:
            file.write('\ntime {}: {} pieces of product {} have been
demanded'.format(round(time, 2), round(to_handle, 2), self.id))
        empty_lots = 0
        for lot in range(len(self.lots)):
            if self.lots[lot] >= to_handle > 0.0:
                self.lots[lot] -= to_handle
                self.income += to_handle * self.sales_prise
                if self.to_report is True:
                    file.write('\nNo backorder arose')
                break
            else:
                to_handle -= self.lots[lot]
                self.income += self.lots[lot] * self.sales_prise
                self.backorders += to_handle
                empty_lots += 1
                try:
                    self.stats.backorders.append(time)
                except AttributeError:
                    pass
                if self.to_report is True:
                    file.write('\nBackorder of {}
arose'.format(round(to_handle, 2)))
        if empty_lots != 0.0 and len(self.lots) > 1:
            for i in range(empty_lots):
                self.lots.pop(i)
                self.expiry_date.pop(i)
                break
            empty_lots = 0.0
        if self.to_report is True:
            file.write('\nStorge {} is {}. Backorder is
{}'.format(self.id, [round(element) for element in self.lots],
round(self.backorders)))
        if sum(self.lots) <= self.reorder_point and self.status ==
False:
            self.replenish(time)
            self.costs += self.purchase_prise * self.reorder_size
#product is orderd
        try:
            self.stats.update_storage(time, sum(self.lots),
                                      (self.income - self.costs))
#gather stats
        except AttributeError:
            pass
```

```
        return time

    def replenish(self, time):
        self.status = True
        self.next_rep = time +
self.g.normal(self.replenishment_lead[0], self.replenishment_lead[1])
        if self.to_report is True:
            file.write('\n{} pieces of product {} have been
ordered'.format(round(self.reorder_size, 2), self.id))

    def handle_replenishment(self, time, free_capacity):
        self.expiry_date = list(map(lambda x: x - (self.next_rep -
time), self.expiry_date))
        self.check_expiry(time)
        if free_capacity >= self.reorder_size:
            self.lots.append(self.reorder_size)
            self.expiry_date.append(self.g.normal(self.expiry[0],
self.expiry[1]))
            if self.to_report is True:
                file.write('\ntime {}: {} pieces of product {} have
been replehished'.format(round(time, 2),

round(self.reorder_size, 2), self.id))
        else:
            self.lots.append(free_capacity)
            self.expiry_date.append(self.g.normal(self.expiry[0],
self.expiry[1]))
            self.overflows += (self.reorder_size - free_capacity)
            self.costs += (self.reorder_size - free_capacity) *
self.overflow_fee #fee for overflow
            if self.to_report is True:
                file.write('\nStorage overflow {} pieces of product {}
are sent back'.format((self.reorder_size - free_capacity), self.id))
        self.costs += (self.next_rep - time) * sum(self.lots) *
self.handling_cost #handling costs
        time = self.next_rep
        self.status = False
        self.next_rep = float('inf')
        if free_capacity < self.reorder_size or free_capacity < 0.0:
            free_capacity = 0.0
        return time


class Statistics:
    def __init__(self, reorder_line, size_line):
        self.time = []
```

```
        self.storage = []
        self.profit = []
        self.reorder_line = reorder_line
        self.size_line = size_line
        self.backorders = []
        self.expires = []
        self.expires_time =[]
        self.overflows = []

    def update_storage(self, time, storage, profit):
        self.time.append(time)
        self.storage.append(storage)
        self.profit.append(profit)

    def plot_storage(self):
        storage_dynamics = plt.figure()
        plt.plot(self.time, self.storage, color='orange',
label='storage')
        plt.axhline(self.reorder_line, color='red', linestyle='--',
label='reorder point')
        plt.axhline(self.size_line, color='black', linestyle='--',
label='reorder size')
        plt.scatter(self.expires, self.expires_time, color='black',
marker='x', label='goods are expired')
        plt.scatter(self.backorders, [0 for i in
range(len(self.backorders))], color='red', s=2, label='backorders')
        plt.legend()
        plt.xlabel('modeling time')
        plt.ylabel('inventory')
        plt.show()

    def plot_profit(self):
        money_dynamics = plt.figure()
        plt.plot(self.time, self.profit, color='orange')
        plt.axhline(self.reorder_line, color='red', linestyle='--',
label='break-even point')
        plt.legend()
        plt.xlabel('modeling time')
        plt.ylabel('net profit')
        plt.show()

    def plot_phase(self):
        phase = plt.figure()
        #plt.plot(self.storage, self.profit)
        plt.plot(self.storage[0:-1], self.storage[1:], color='blue')
        #plt.plot(self.profit[0:-1], self.profit[1:])
```

```
        plt.title('Pseudo phase portait')
        plt.xlabel('I(t)')
        plt.ylabel('I(t+1)')
        plt.show()

class Simulation:
    def __init__(self, adjustments, parameters, costs, strategy,
horizon=660.0):
        self.w = Warehouse(adjustments, parameters, costs, strategy)
        self.horizon = horizon

    def simulate(self):
        while self.w.time < self.horizon:
            self.w.advance_time()
        try:
            self.w.products[0].stats.plot_storage()
            self.w.products[0].stats.plot_profit()
            self.w.products[0].stats.plot_phase()
        except AttributeError:
            pass
        total_cost = 0.0
        for i in range(len(self.w.products)):
            total_cost += (self.w.products[i].income -
self.w.products[i].costs -
self.w.products[i].backorders*self.w.products[i].backorder_fee)
        return total_cost
```

Appendix 2. Source-code of the simulation of the stochastic multiproduct production-inventory system with lost-sales and Markov-modulated demand

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

class Generator:
    def normal(self, mu, sigma):
        return np.random.normal(mu, sigma)

    def exponential(self, betta):
        return np.random.exponential(betta)

    def uniform(self):
        return np.random.uniform()

class Plant:
    def __init__(self, markets, interarrival, rate, T,
markovian=False):
        self.g = Generator()
        self.time = 0.0
        self.markets = []
        self.T = T
        self.rate = rate
        self.status = [True, True, True, True]
        for market in markets:
            self.markets.append(Market(market, T, markovian))
        self.inventory = [100, 100, 100, 100]
        self.total_capacity = 1000
        self.overflow_fee = 10
        self.prices = [20, 20, 20, 20]
        self.production_costs = [15, 15, 15, 15]
        self.launch_costs = [500, 500, 500, 500]
        self.start = [100, 100, 100, 100]
        self.stop = [200, 200, 200, 200]
        self.queue = []
        self.max_orders = 5
        self.backlog_fees = [30, 30, 30, 30]
        self.interarrival = interarrival
        self.next_demand = self.g.exponential(self.interarrival)
        self.profit = 0.0
        for i,j in zip(self.inventory, self.production_costs):
            self.profit -= i*j
```

```
    def produce(self, lag):
        for i in range(len(self.inventory)):
            if self.status[i] == True:
                produced = self.g.normal(self.rate[i][0],
self.rate[i][0]*self.rate[i][1])*lag
                self.profit -= produced*self.production_costs[i]
                self.inventory[i] += produced
            if self.inventory[i] >= self.stop[i] and
self.status[i]==True:
                self.status[i] = False
                #print('!!! production of product {} has been
interrupted'.format(i))
            if sum(self.inventory) > self.total_capacity:
                #print('overflow!')
                self.profit -= (sum(self.inventory) -
self.total_capacity)*self.overflow_fee

    def advance_time(self):
        lag = self.next_demand - self.time
        self.time = self.next_demand
        self.produce(lag)
        self.next_demand = self.time +
self.g.exponential(self.interarrival)
        self.queue.append(self.markets[0].demand_arises())
        for i in range(len(self.queue)):
            if all(self.inventory[j] >= self.queue[i][j] for j in
range(len(self.inventory))) is True:
                #print('{} pieces are sold and
shipped'.format(sum(self.queue[i])))
                self.inventory = list(map(lambda x, y: x-y,
self.inventory, self.queue[i]))
                for j,k in zip(self.queue[i], self.prices):
                    self.profit += j*k
                self.queue[i] = [0 for i in range(len(self.queue[i]))]

        for i in range(len(self.queue)):
            try:
                if self.queue[i][0] == 0:
                    self.queue.pop(i)
            except:
                pass

        if len(self.queue) > self.max_orders:
            #print('backlog!')
            tmp = self.queue.pop(-1)
            for i,j in zip(tmp, self.backlog_fees):
```

```
                self.profit -= i*j

        for i in range(len(self.inventory)):
            if self.inventory[i] < self.start[i] and self.status[i] ==
False:
                self.status[i] = True
                self.profit -= self.launch_costs[i]
                #print('!!! production of product {} has been
launched'.format(i))

class Market:
    def __init__(self, parameters, transitions, markovian):
        self.g = Generator()
        self.demands = parameters[0]
        self.markovian = markovian
        self.transitions = transitions
        self.states = [0, 0, 0, 0]

    def shift(self):
        for i in range(len(self.demands)):
            r = self.g.uniform()
            if r <= self.transitions[i][self.states[i]][0]:
                self.states[i] = 0
            elif self.transitions[i][self.states[i]][0] < r <=
self.transitions[i][self.states[i]][1] +
self.transitions[i][self.states[i]][0]:
                self.states[i] = 1
                self.demands[i][0] = self.demands[i][0]*1.01
            else:
                self.states[i] = 2
                self.demands[i][0] = self.demands[i][0] * 0.99

    def demand_arises(self):
        demand = []
        if self.markovian == False:
            self.shift()
        for i in range(len(self.demands)):
            demand.append(self.g.normal(self.demands[i][0],
self.demands[i][0]*self.demands[i][1]))
        #print('demand arises {}'.format(demand))
        return demand

class Statistics:
    def __init__(self):
        self.time_vector = []
        self.inventory_vector = []
```

```
        self.individual_inventory_vector = []
        self.profit_vector = []
        self.statuses = [[], [], [], []]

    def gather_inventory_stats(self, time, inventory,
individual_inventory, profit, status):
        self.time_vector.append(time)
        self.inventory_vector.append(inventory)
        self.individual_inventory_vector.append(individual_inventory)
        self.profit_vector.append(profit)
        for i in range(len(status)):
            self.statuses[i].append(status[i])

    def plot_inventory(self, x, y):
        plt.grid()
        plt.plot(x, y, color='orange')
        plt.xlabel('time')
        plt.ylabel('inventory level')
        plt.show()

    def plot_individual_inventory(self, x, y, down, up):

        fig, axs = plt.subplots(2, 1)
        axs[0].grid(True)
        axs[0].plot(x, y, color='orange', label='inventory dynamics')
        axs[0].hlines(down, 0, max(x)-1, color='green',
linestyles=':', linewidth=3, label='resume production')
        axs[0].hlines(up, 0, max(x) - 1, color='red', linestyles=':',
linewidth=3, label='interrupt production')
        axs[0].set_ylabel('inventory level')
        #axs[0].set_xlabel('time')
        axs[0].legend()

        axs[1].grid(True)
        axs[1].plot(x, self.statuses[0], color='blue', label='status')
        axs[1].set_yticks([0.0, 1.0])
        axs[1].set_yticklabels(["False", "True"])
        axs[1].set_xlabel('time')
        axs[1].set_ylabel('production status')
        axs[1].legend()
        fig.tight_layout()
        plt.show()

    def plot_phase(self, x):
        plt.grid()
        plt.plot(x[0:-1], x[1:], color='blue')
```

```
        plt.xlabel('inventory level (t)')
        plt.ylabel('inventory level (t+1)')
        plt.show()

    def plot_profit(self, x, y):
        plt.grid()
        plt.plot(x, y, color='green', label='monetary dynamics')
        plt.hlines(0,0,self.time_vector[-1], color='red',
linestyles=':', linewidth=3, label='break-even point')
        plt.xlabel('time')
        plt.ylabel('net profit')
        plt.legend()
        plt.show()

    def plot_status(self, x, y):
        plt.grid()
        colors = ['red', 'green', 'blue', 'orange']
        for i in range(len(y)):
            plt.plot(x, y[i], color=colors[i], label='status
{}'.format(i+1))
        plt.xlabel('time')
        plt.ylabel('production status')
        plt.legend()
        plt.show()

class Simulator:
    def __init__(self, planning_horizon):

        self.plant = Plant([[[[10, 0.2], [8, 0.2], [7, 0.2], [10,
0.2]]]],
    1.0,
    [[12, 0.2], [10, 0.2], [9, 0.2], [12, 0.2]],
                           [np.array([[0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33]]),
                            np.array([[0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33]]),
                            np.array([[0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33]]),
                            np.array([[0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33],
                                      [0.33, 0.34, 0.33]])
                            ])
        self.stats = Statistics()
```

```
        self.planning_horizon = planning_horizon

    def simulate(self):
        while self.plant.time < self.planning_horizon:
            self.plant.advance_time()

            self.stats.gather_inventory_stats(self.plant.time,
sum(self.plant.inventory), self.plant.inventory[0],
                                                self.plant.profit,
self.plant.status)
        self.stats.plot_inventory(self.stats.time_vector,
self.stats.inventory_vector)
        self.stats.plot_individual_inventory(self.stats.time_vector,
self.stats.individual_inventory_vector,
                                             self.plant.start[0],
self.plant.stop[0])
        self.stats.plot_profit(self.stats.time_vector,
self.stats.profit_vector)
        #self.stats.plot_status(self.stats.time_vector,
self.stats.statuses)
        self.stats.plot_phase(self.stats.inventory_vector)
        print(self.plant.markets[0].demands)
        return self.plant.profit

sim1 = Simulator(250)
sim1.simulate()
```

Appendix 3. Defining the number of replications based on confidence intervals

```
from simulations import simulation, simulation2
import matplotlib.pyplot as plt
import numpy as np
import seaborn as sns
from scipy import stats

def sample(type):
    dist = []
    if type==1:
        for i in range(10000):
            s = simulation.Simulation([[0, 0]],  # to_plot, to_report
                                      [[0.1, [0.5, 0.1], [4, 0.5],
[20, 2]]],  # interarrivals, demand, replenishment_lead, expiry
                                      [[70.0, 100.0, 1.0, 30.0, 100.0,
100.0]],
# purchase price, sales price, handling, backorder, overflow, recycle
                                      [[35, 20]])  # storage, reorder
            dist.append(s.simulate())
            print(i)
        mu, sigma = stats.norm.fit(dist)
        # create a normal distribution with loc and scale
        test_sample = np.random.normal(mu, sigma, 10000)
        n = stats.norm(loc=mu, scale=sigma)
        print(stats.kstest(dist, n.cdf))
        print(stats.chisquare(dist, test_sample))
        print(stats.anderson(dist, dist='norm'))
        left, right = stats.norm.interval(0.95, loc=mu,
scale=(sigma/np.sqrt(len(dist))))

        sns.distplot(np.asarray(dist), bins=100, kde=True,
                     kde_kws={"color": "r", "lw": 3, "label": "Kernel
density estimation"})
        plt.axvspan(left, right, alpha=0.35, color='red', label='95%
Confidence interval')
        plt.legend()
        plt.xlabel('Cost function')
        plt.show()
        ax = sns.boxplot(np.asarray(dist))
        plt.xlabel('Cost function')
        plt.show()

    else:
        for i in range(10000):
```

```
            s = simulation2.Simulator(250)
            dist.append(s.simulate())
            print(i)

        mu, sigma = stats.norm.fit(dist)
        # create a normal distribution with loc and scale
        test_sample = np.random.normal(mu, sigma, 10000)
        print('mu: ', mu, 'n/sigma: ', sigma)
        print(stats.anderson(dist, dist='norm'))
        left, right = stats.norm.interval(0.95, loc=mu, scale=(sigma /
np.sqrt(len(dist))))

        sns.distplot(np.asarray(dist), bins=100, kde=True,
                     kde_kws={"color": "r", "lw": 3, "label": "Kernel
density estimation"})
        plt.axvspan(left, right, alpha=0.35, color='red', label='95%
Confidence interval')
        plt.legend()
        plt.xlabel('Cost function')
        plt.show()
        ax = sns.boxplot(np.asarray(dist))
        plt.xlabel('Cost function')
        plt.show()
```

Appendix 4. Source-code of the Monte Carlo samplers

```
from simulations import simulation, simulation2_sample
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

def evaluate(q_r, c, par):
    costs = []
    for sim in range(10):
        s = simulation.Simulation([[0, 0] for i in range(10)],
# to_plot, to_report
                                  par,  # interarrivals, demand,
replenishment_lead, expiry
                                  c,  # purchase price, sales price,
handling, backorder, overflow, recycle
                                  q_r)  # storage, reorder point
        costs.append(s.simulate())
    return sum(costs)/len(costs)

#strategy
q_r = []
#environment
c = []
par = []
#output
profit = 0.0
dataset=[]

for i in range(500):
    dataset.append([])
    for j in range(5):
        q_r.append([np.random.randint(10, 50), np.random.randint(8,
40)])
        while q_r[j][1] > q_r[j][0]:
            q_r[j][1] = np.random.randint(8, 30)
        dataset[i].append(q_r[-1][0])
        dataset[i].append(q_r[-1][1])

        par.append([np.random.uniform(0.2, 0.7),
[np.random.uniform(0.2, 0.7), np.random.uniform(0.05, 0.3)],
                    [np.random.randint(7, 12), np.random.uniform(0.2,
3)], [np.random.randint(8, 15), np.random.uniform(0.2, 3)]])
```

```
        while par[j][1][0] <= 2*par[j][1][1]:
            par[j][1][1] = np.random.uniform(0.05, 0.3)

        dataset[i].append(par[-1][0])
        dataset[i].append(par[-1][1][0])
        dataset[i].append(par[-1][1][1])
        dataset[i].append(par[-1][2][0])
        dataset[i].append(par[-1][2][1])
        dataset[i].append(par[-1][3][0])
        dataset[i].append(par[-1][3][1])

        c.append([np.random.randint(60, 150), np.random.randint(150,
200), np.random.uniform(2, 7),
                  np.random.randint(10, 50), np.random.randint(30,
130), np.random.randint(30, 130)])

        while c[j][0] > c[j][1]:
            c[j][0] = np.random.randint(60, 150)

        dataset[i].append(c[-1][0])
        dataset[i].append(c[-1][1])
        dataset[i].append(c[-1][2])
        dataset[i].append(c[-1][3])
        dataset[i].append(c[-1][4])
        dataset[i].append(c[-1][5])

    profit = evaluate(q_r, c, par)
    dataset[i].append(profit)

    q_r = []
    c = []
    par = []
    profit = 0.0
    print(i)

df = pd.DataFrame(dataset)
ax = sns.boxplot(df.iloc[:,-1])
plt.show()
writer = pd.ExcelWriter('new_10_trainingset_full.xlsx')
df.to_excel(writer,'Sheet1', header=False, index=False)
writer.save()


dataset = []
def to_sample(n, k):
    global pos, neg
```

```
    def generate_candidate_solution():
        global sim, parameters
        parameters = []
        for j in range(4):
            parameters.append(np.random.uniform(5, 20))
        for j in range(4):
            parameters.append(np.random.uniform(7, 25))
        for j in range(4):
            parameters.append(np.random.uniform(20, 200)) #initial I
        parameters.append(np.random.uniform(0, 100))  # overflow fee
        for j in range(4):
            parameters.append(np.random.uniform(10, 200))  # prices
        for j in range(4):
            parameters.append(np.random.uniform(10, 200))  #
production costs
        for j in range(4):
            parameters.append(np.random.uniform(5, 40))  # lost-sasles

        sim = simulation2_sample.Simulator(250,  # planning horizon
                           [parameters[0], parameters[1], parameters[2],
parameters[3]],  # demands
                           1.0,  # interarrivals
                           [parameters[4], parameters[5], parameters[6],
parameters[7]],  # supplies
                           [parameters[8], parameters[9],
parameters[10], parameters[11]],  # initial inventory levels
                           1000,  # total capacity
                           parameters[12],  # overflow_fee
                           [parameters[13], parameters[14],
parameters[15], parameters[16]],  # prices
                           [parameters[17], parameters[18],
parameters[19], parameters[20]],  # production costs
                           [100, 100, 100, 100],  # launch costs
                           [100, 100, 100, 100],  # condition to start
production
                           [200, 200, 200, 200],  # condition to stop
production
                           [parameters[21], parameters[22],
parameters[23], parameters[24]]) # lost-sales fees

    pos = 0
    neg = 0

    for i in range(k):
        generate_candidate_solution()
        fitness = []
```

```
            for j in range(n):
                fitness.append(sim.simulate())
            parameters.append(sum(fitness)/len(fitness))
            if parameters[-1] <0:
                parameters.append(0)
                neg += 1
            else:
                parameters.append(1)
                pos += 1
            if parameters[-2] > -1000000:
                dataset.append(parameters)
            print(i)

to_sample(30, 1000)
df = pd.DataFrame(dataset)
print(df.iloc[:,-2].mean())
print(pos, neg)
ax = sns.boxplot(df.iloc[:,-2])
plt.show()
writer = pd.ExcelWriter('simulation2_trainingset_full.xlsx')
df.to_excel(writer,'Sheet1', header=False, index=False)
writer.save()
```

Appendix 5. Source-code of the neuroevolutionary framework

```
import numpy as np
import pandas as pd
import random
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense, Dropout, ActivityRegularization
from keras.wrappers.scikit_learn import KerasRegressor,
KerasClassifier
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import scale
from sklearn.metrics import r2_score, f1_score
from deap import algorithms
from deap import base
from deap import creator
from deap import tools


class Regressors:
    def __init__(self, individual):
        depths = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        widths = [10, 20, 30, 40, 50, 60, 70, 80, 90]
        afs = ['relu', 'elu', 'selu', 'sigmoid', 'tanh', 'relu',
'elu', 'selu', 'sigmoid']
        opts = ['sgd', 'adam', 'adagrad', 'adamax', 'nadam', 'adam',
'adagrad', 'adamax', 'nadam']
        drops = [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
        l1s = [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
        l2s = [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
        self.depth = depths[individual[0]]
        self.width = widths[individual[1]]
        self.af = afs[individual[2]]
        self.opt = opts[individual[3]]
        self.drop = drops[individual[4]]
        self.l1 = l1s[individual[5]]
        self.l2 = l2s[individual[6]]

    def baseline_model(self):
        model = Sequential()
        model.add(Dense(self.width, input_dim=75,
kernel_initializer='normal', activation=self.af))
        for layer in range(self.depth):
            model.add(Dense(self.width, kernel_initializer='normal',
activation=self.af))
```

```
            model.add(Dropout(self.drop, noise_shape=None, seed=None))
            model.add(ActivityRegularization(l1=self.l1, l2=self.l2))
        model.add(Dense(1, kernel_initializer='normal'))
        model.compile(loss='mse', optimizer=self.opt)
        return model

class Classifiers:
    def __init__(self, individual):
        depths = [1, 2, 3, 4, 5, 6, 7, 8, 9]
        widths = [10, 20, 30, 40, 50, 60, 70, 80, 90]
        afs = ['relu', 'elu', 'selu', 'sigmoid', 'tanh', 'relu',
'elu', 'selu', 'sigmoid']
        opts = ['sgd', 'adam', 'adagrad', 'adamax', 'nadam', 'adam',
'adagrad', 'adamax', 'nadam']
        drops = [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
        l1s = [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
        l2s = [0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4]
        self.depth = depths[individual[0]]
        self.width = widths[individual[1]]
        self.af = afs[individual[2]]
        self.opt = opts[individual[3]]
        self.drop = drops[individual[4]]
        self.l1 = l1s[individual[5]]
        self.l2 = l2s[individual[6]]

    def baseline_model(self):
        model = Sequential()
        model.add(Dense(self.width, input_dim=75,
kernel_initializer='normal', activation=self.af))  # 17
        for layer in range(self.depth):
            model.add(Dense(self.width, kernel_initializer='normal',
activation=self.af))
            model.add(Dropout(self.drop, noise_shape=None, seed=None))
            model.add(ActivityRegularization(l1=self.l1, l2=self.l2))
        model.add(Dense(1, kernel_initializer='normal',
activation='sigmoid'))
        model.compile(loss='binary_crossentropy', optimizer=self.opt)
        return model

class BlindWatchmaker:
    def __init__(self, pop_size, generations, m_rate, c_rate, t_size,
epochs, b_size, cv, type):
        self.pop_size = pop_size #population size
        self.generations = generations #number of generations
        self.m_rate = m_rate #mutation rate
        self.c_rate = c_rate #cross-over rate
```

```
        self.t_size = t_size #tournament size
        self.epochs = epochs #number of epochs
        self.b_size = b_size #batch size
        self.cv = cv #folds in cross validation
        self.type = type

    def get_data(self):
        if self.type == 'regressor':
            #df = pd.read_excel('simulation2_trainingset_full.xlsx')
            df = pd.read_excel('new_10_trainingset_full.xlsx')
            dataset = scale(df.values)
            X = dataset[:, :75]
            Y = dataset[:, 75]
        else:
            # df = pd.read_excel('simulation2_trainingset_class.xlsx')
            df = pd.read_excel('10_class.xlsx')
            dataset = df.values
            X = scale(dataset[:, :75])
            Y = dataset[:, 75]

        return X, Y

    def ev(self):
        X, Y = self.get_data()
        creator.create("FitnessMax", base.Fitness, weights=(1.0,))
        creator.create("Individual", list, typecode='b',
fitness=creator.FitnessMax)

        toolbox = base.Toolbox()

        # Attribute generator
        toolbox.register("attr_bool", random.randint, 0, 8)

        # Structure initializers
        toolbox.register("individual", tools.initRepeat,
creator.Individual, toolbox.attr_bool, 7)
        toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

        def eval(individual):
            if self.type == 'regressor':
                r = Regressors(individual)
                model = r.baseline_model
                estimator = KerasRegressor(build_fn=model,
epochs=self.epochs, batch_size=self.b_size, verbose=0)
```

```
                results = cross_val_predict(estimator, X, Y,
cv=self.cv)
                r2 = r2_score(Y, results)
                return r2,
            else:
                c = Classifiers(individual)
                model = c.baseline_model
                estimator = KerasClassifier(build_fn=model,
epochs=self.epochs, batch_size=self.b_size, verbose=0)
                results = cross_val_predict(estimator, X, Y,
cv=self.cv)
                f1 = f1_score(Y, results)
                return f1,

        toolbox.register("evaluate", eval)
        toolbox.register("mate", tools.cxOnePoint)
        toolbox.register("mutate", tools.mutUniformInt, low=0, up=8,
indpb=0.2)
        toolbox.register("select", tools.selTournament,
tournsize=self.t_size)

        def evolve():
            random.seed(64)

            pop = toolbox.population(n=self.pop_size)
            hof = tools.HallOfFame(1)
            stats = tools.Statistics(lambda ind: ind.fitness.values)
            stats.register("avg", np.mean)
            stats.register("std", np.std)
            stats.register("max", np.max)
            stats.register("min", np.min)

            pop, log = algorithms.eaSimple(pop, toolbox,
cxpb=self.c_rate, mutpb=self.m_rate, ngen=self.generations,
                                           stats=stats,
halloffame=hof, verbose=True)

            return log, hof

        log, hof = evolve()
        print('Praise the fittest one!', hof)
        def plot():
            gen = log.select("gen")
            fits = log.select("avg")
            std = log.select("std")
            std = [i/2 for i in std]
```

```
            plt.grid()
            plt.errorbar([i+1 for i in gen], fits, std, uplims=True,
fmt='-o', color='blue')
            plt.xlabel("Generation")
            plt.ylabel("F1 score")
            plt.show()
        plot()

evolver = BlindWatchmaker(15, 9, 0.1, 0.4, 3, 40, 4, 2, 'regressor')
evolver.ev()
```

Appendix 6. Source-code of the MLP-based metamodel (regression)

```
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
from scipy import stats
from math import sqrt
from keras.models import Sequential
from keras.layers import Dense, Dropout, ActivityRegularization
from keras.wrappers.scikit_learn import KerasRegressor
from keras.utils import plot_model
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import scale
from sklearn.metrics import explained_variance_score
from sklearn.metrics import r2_score, mean_squared_error
from statsmodels.stats.diagnostic import het_breuschpagan

class Data:
    def __init__(self, simulation, scaling=False):
        self.simulation = simulation
        self.scaling = scaling

    def load(self):
        # z-score standartization
        def scaler(data):
            return scale(data.values)
        # load dataset
        if self.simulation == 1:
            df = pd.read_excel('new_10_trainingset_full.xlsx').values
            if self.scaling == True:
                df = scaler(df)
            # split into input (X) and output (Y) variables
            X = df[:, :75]
            Y = df[:, 75]

        else:
            df = pd.read_excel('simulation2_trainingset_full.xlsx')
            if self.scaling == True:
                df = scaler(df)
            # split into input (X) and output (Y) variables
            X = df[:, :25]
            Y = df[:, 25]

        return X, Y
```

```
class Model:
    def __init__(self, plot=False, summary=False):
        self.plot = plot
        self.summary = summary

    def baseline_model(self):
        model = Sequential()
        model.add(Dense(200, input_dim=75,
kernel_initializer='normal', activation='elu'))
        model.add(Dropout(0.1, noise_shape=None, seed=None))
        model.add(ActivityRegularization(l1=300, l2=300))
        model.add(Dense(100, kernel_initializer='normal',
activation='elu'))
        model.add(Dropout(0.1, noise_shape=None, seed=None))
        model.add(ActivityRegularization(l1=200, l2=200))
        model.add(Dense(100, kernel_initializer='normal',
activation='elu'))
        model.add(Dropout(0.1, noise_shape=None, seed=None))
        model.add(ActivityRegularization(l1=100, l2=100))
        model.add(Dense(1, kernel_initializer='normal'))
        model.compile(loss='mse', optimizer='adamax')

        if self.summary == True:
            print(model.summary())

        if self.plot == True:
            plot_model(model, to_file='model_plot.png',
show_shapes=True, show_layer_names=False)

        return model

class Training:
    def __init__(self, cv=False):
        model = Model()
        data = Data(1)
        self.X, self.Y = data.load()

        if cv == False:
            self.estimator =
KerasRegressor(build_fn=model.baseline_model, epochs=200,
batch_size=2, verbose=0, validation_split=0.25)
        else:
            self.estimator =
KerasRegressor(build_fn=model.baseline_model, epochs=100,
batch_size=2, verbose=0)
```

```
    def train_model(self, plot_distribution=False,
residual_analysis=False, learning_path=False, ols=False, save=False):
        history = self.estimator.fit(self.X, self.Y)
        results = self.estimator.predict(self.X)
        r2 = r2_score(self.Y, results)
        adjusted_r2 = 1 - (1-r2)*(1000-1)/(1000-75-1)
        see = sqrt(sum((self.Y-results)**2)/(1000-75))
        mse = mean_squared_error(self.Y, results)
        print('explained variance ', explained_variance_score(self.Y,
results))
        print('r2 ', r2)
        print('adjusted ', adjusted_r2)
        print('mse ', mse)
        print('Standard error of the estimate ', see)

        if plot_distribution == True:
            ax = sns.boxplot(x=['data', 'prediction'], y=[self.Y,
results])
            plt.show()
            ax = sns.violinplot(data=[self.Y, results])
            plt.show()

        if residual_analysis == True:
            residuals = [i - j for i, j in zip(self.Y, results)]
            print(stats.anderson(residuals, dist='norm'))
            print('mean ', sum(residuals) / len(residuals))
            sns.distplot(residuals, bins=20, kde=True,
                         kde_kws={"color": "r", "lw": 3, "label":
"Kernel density estimation"})
            plt.legend()
            plt.xlabel('residuals')
            plt.show()
            res = stats.probplot(residuals, plot=plt)
            plt.show()

        if learning_path == True:
            # Plot training & validation loss values
            plt.plot([i for i in history.history['loss']],
label='Train')
            plt.plot([i for i in history.history['val_loss']],
label='Test')
            plt.title('Model loss')
            plt.ylabel('Loss')
            plt.xlabel('Epoch')
            plt.legend()
            plt.show()
```

```
        if ols == True:
            model = sm.OLS(self.Y, self.X)
            results = model.fit()
            resids = results.resid
            exog = results.model.exog
            print(results.summary())
            print(het_breuschpagan(resids, exog))

    def cv_score(self, cv=10):
        cv_score = cross_val_score(self.estimator, self.X, self.Y,
cv=cv, scoring='r2')
        print(cv_score)
        ax = sns.boxplot(cv_score)
        plt.show()

if __name__== "__main__":
    t = Training()
    t.train_model(plot_distribution=True, learning_path=True)
```

Appendix 7. Source-code of the MLP-based metamodel (classification)

```
import pandas as pd
import seaborn as sns
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.wrappers.scikit_learn import KerasClassifier
from sklearn.preprocessing import scale
import matplotlib.pyplot as plt
import keras_metrics
from sklearn.metrics import confusion_matrix, cohen_kappa_score
from sklearn.model_selection import cross_val_score
from keras.utils import plot_model

class Data:
    def __init__(self, simulation, scaling=True):
        self.simulation = simulation
        self.scaling = scaling

    def load(self):
        # z-score standartization
        def scaler(data):
            return scale(data)
        # load dataset
        if self.simulation == 1:
            df = pd.read_excel('10_class.xlsx').values
            # split into input (X) and output (Y) variables
            X = df[:, :75]
            Y = df[:, 75]
            if self.scaling == True:
                X = scaler(X)

        else:
            df =
pd.read_excel('simulation2_trainingset_class.xlsx').values
            # split into input (X) and output (Y) variables
            X = df[:, :17]
            Y = df[:, 17]
            if self.scaling == True:
                X = scaler(X)

        return X, Y

class Model:
    def __init__(self, plot=False, summary=False):
```

```
        self.plot = plot
        self.summary = summary

    def baseline_model(self):
        model = Sequential()
        model.add(Dense(30, input_dim=75, kernel_initializer='normal',
activation='elu'))
        model.add(Dropout(0.3, noise_shape=None, seed=None))
        model.add(Dense(20, kernel_initializer='normal',
activation='elu'))
        model.add(Dense(1, kernel_initializer='normal',
activation='sigmoid'))
        model.compile(loss='binary_crossentropy', optimizer='adamax',
                      metrics=['accuracy', keras_metrics.precision(),
keras_metrics.recall()])

        if self.summary == True:
            print(model.summary())

        if self.plot == True:
            plot_model(model, to_file='model_plot.png',
show_shapes=True, show_layer_names=False)

        return model

class Training:
    def __init__(self, cv=False):
        model = Model()
        data = Data(1)
        self.X, self.Y = data.load()

        if cv == False:
            self.estimator =
KerasClassifier(build_fn=model.baseline_model, epochs=60,
batch_size=200, verbose=0, validation_split=0.35)
        else:
            self.estimator =
KerasClassifier(build_fn=model.baseline_model, epochs=60,
batch_size=200, verbose=0)

    def train_model(self, to_plot=True):
        history = self.estimator.fit(self.X, self.Y)
        results = self.estimator.predict(self.X)

        cm = confusion_matrix(self.Y, results)
        kappa = cohen_kappa_score(self.Y, results)
```

```
        print(cm)
        tn, fp, fn, tp = cm.ravel()
        acc = (tp + tn) / (tp + tn + fp + fn)
        prec = tp / (tp + fp)
        rec = tp / (tp + fn)
        f1 = 2 * tp / (2 * tp + fp + fn)
        loss = history.history['val_loss'][-1]
        print('acc ', acc)
        print('prec ', prec)
        print('rec ', rec)
        print('f1 ', f1)
        print('kappa', kappa)
        print('loss ', loss)

        if to_plot == True:
            plt.title('Accuracy')
            plt.plot(history.history['acc'], label='Train')
            plt.plot(history.history['val_acc'], label='Test')
            plt.xlabel('Epoch')
            plt.legend()
            plt.show()

            plt.plot(history.history['val_precision'],
label='Precision')
            plt.plot(history.history['val_recall'], label='Recall')
            plt.xlabel('Epoch')
            plt.legend()
            plt.show()

            plt.title('Loss')
            plt.plot(history.history['loss'], label='Train')
            plt.plot(history.history['val_loss'], label='Test')
            plt.xlabel('Epoch')
            plt.legend()
            plt.show()

    def cv_score(self, cv=10):
        cv_score = cross_val_score(self.estimator, self.X, self.Y,
cv=cv, scoring='f1')
        print(cv_score)
        ax = sns.boxplot(cv_score)
        plt.show()

if __name__== "__main__":
    t = Training(cv=True)
    t.train_model()
```

Appendix 8. Source-code of the metamodel-based optimization algorithm

```
import numpy as np
import random
import matplotlib.pyplot as plt
from simulation2_sample import Simulator
from simulation import Simulation
from deap import algorithms
from deap import base
from deap import creator
from deap import tools

class SimpleOpt:
    def __init__(self, pop_size, generations, m_rate, c_rate, t_size,
type=1):
        self.pop_size = pop_size #population size
        self.generations = generations #number of generations
        self.m_rate = m_rate #mutation rate
        self.c_rate = c_rate #cross-over rate
        self.t_size = t_size #tournament size
        self.type = type

    def ev(self):
        creator.create("FitnessMax", base.Fitness, weights=(1.0,))
        creator.create("Individual", list, typecode='b',
fitness=creator.FitnessMax)

        toolbox = base.Toolbox()

        # Attribute generator
        toolbox.register("attr_bool", random.randint, 0, 400)

        # Structure initializers
        if type == 1:
            toolbox.register("individual", tools.initRepeat,
creator.Individual, toolbox.attr_bool, 2)
        else:
            toolbox.register("individual", tools.initRepeat,
creator.Individual, toolbox.attr_bool, 16)
        toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

        def eval(individual, n=1):
            fitness = []
            if self.type == 1:
```

```
                for i in range(n):
                    sim1 = Simulation([[0, 0]],  # to_plot, to_report
                                      [[0.1, [0.5, 0.1], [4, 0.5],
[20, 2]]],
                                      # interarrivals, demand,
replenishment_lead, expiry
                                      [[70.0, 200.0, 1.0, 30.0, 100.0,
100.0]],
                                      # purchase price, sales price,
handling, backorder, overflow, recycle
                                      [[individual[0],
individual[1]]])  # storage, reorder point

                    fitness.append(sim1.simulate())
            else:
                for i in range(n):
                    sim1 = Simulator(250,  # planning horizon
                                     [10, 8, 7, 10],  # demands
                                     1.0,  # interarrivals
                                     [individual[0], individual[1],
individual[2], individual[3]],  # supplies
                                     [individual[4], individual[5],
individual[6], individual[7]],  # initial inventory levels
                                     1000,  # total capacity
                                     10,  # overflow_fee
                                     [50, 50, 50, 50],  # prices
                                     [15, 15, 15, 15],  # production
costs
                                     [500, 500, 500, 500],  # launch
costs
                                     [individual[8], individual[9],
individual[10], individual[11]],
                                     # condition to start production
                                     [individual[12], individual[13],
individual[14], individual[15]],
                                     # condition to stop production
                                     [30, 30, 30, 30]  # lost-sales
                                     )
                    fitness.append(sim1.simulate())

            return (sum(fitness) / len(fitness)),

        toolbox.register("evaluate", eval)
        toolbox.register("mate", tools.cxOnePoint)
        toolbox.register("mutate", tools.mutUniformInt, low=0, up=400,
indpb=0.2)
```

```
        toolbox.register("select", tools.selTournament,
tournsize=self.t_size)

        def evolve():
            random.seed(64)

            pop = toolbox.population(n=self.pop_size)
            hof = tools.HallOfFame(1)
            stats = tools.Statistics(lambda ind: ind.fitness.values)
            stats.register("avg", np.mean)
            stats.register("std", np.std)
            stats.register("max", np.max)
            stats.register("min", np.min)

            pop, log = algorithms.eaSimple(pop, toolbox,
cxpb=self.c_rate, mutpb=self.m_rate, ngen=self.generations,
                                           stats=stats,
halloffame=hof, verbose=True)

            return log, hof

        log, hof = evolve()
        print('The fittest solution: ', hof)
        p = Plotter(log.select("gen"), log.select("max"))
        p.plot()


class CustomOpt:
    def __init__(self):
        pass

    creator.create("FitnessMax", base.Fitness, weights=(1.0,))
    creator.create("Individual", list, fitness=creator.FitnessMax)

    toolbox = base.Toolbox()
    toolbox.register("attr_bool", random.randint, 0, 400)


    toolbox.register("individual", tools.initRepeat,
creator.Individual, toolbox.attr_bool, 16)

    # define the population to be a list of individuals
    toolbox.register("population", tools.initRepeat, list,
toolbox.individual)

    # the goal ('fitness') function to be maximized
```

```
    def eval(individual, n=1):
        fitness = []
        if self.type == 1:
            for i in range(n):
                sim1 = Simulation([[0, 0]],  # to_plot, to_report
                                  [[0.1, [0.5, 0.1], [4, 0.5], [20,
2]]],
                                  # interarrivals, demand,
replenishment_lead, expiry
                                  [[70.0, 200.0, 1.0, 30.0, 100.0,
100.0]],
                                  # purchase price, sales price,
handling, backorder, overflow, recycle
                                  [[individual[0], individual[1]]])  #
storage, reorder point

                fitness.append(sim1.simulate())
        else:
            for i in range(n):
                sim1 = Simulator(250,  # planning horizon
                                 [10, 8, 7, 10],  # demands
                                 1.0,  # interarrivals
                                 [individual[0], individual[1],
individual[2], individual[3]],  # supplies
                                 [individual[4], individual[5],
individual[6], individual[7]],
                                 # initial inventory levels
                                 1000,  # total capacity
                                 10,  # overflow_fee
                                 [50, 50, 50, 50],  # prices
                                 [15, 15, 15, 15],  # production costs
                                 [500, 500, 500, 500],  # launch costs
                                 [individual[8], individual[9],
individual[10], individual[11]],
                                 # condition to start production
                                 [individual[12], individual[13],
individual[14], individual[15]],
                                 # condition to stop production
                                 [30, 30, 30, 30]  # lost-sales fees
                                 )
                fitness.append(sim1.simulate())

        return (sum(fitness) / len(fitness)),

    # register the fitness function
    toolbox.register("evaluate", eval)
```

```
    # register the crossover operator
    toolbox.register("mate", tools.cxTwoPoint)
    # register a mutation operator
    toolbox.register("mutate", tools.mutUniformInt, indpb=0.05, low=0,
up=400)
    # operator for selecting individuals for breeding
    toolbox.register("select", tools.selTournament, tournsize=3)

    def main():
        random.seed(64)
        # create an initial population of 40 individuals
        pop = toolbox.population(n=40)
        # CXPB  is the crossover probability
        # MUTPB is the probability for mutating an individual
        CXPB, MUTPB = 0.5, 0.2
        print("Start of evolution")
        # Evaluate the population
        fitnesses = list(map(toolbox.evaluate, pop))
        for ind, fit in zip(pop, fitnesses):
            ind.fitness.values = fit

        print("  Evaluated %i individuals" % len(pop))

        # Extracting all the fitnesses of
        fits = [ind.fitness.values[0] for ind in pop]
        # Variable keeping track of the number of generations
        g = 0
        # Begin the evolution
        while g < 30:
            # A new generation
            g = g + 1
            print("-- Generation %i --" % g)

            # Select the next generation individuals
            offspring = toolbox.select(pop, len(pop))
            # Clone the selected individuals
            offspring = list(map(toolbox.clone, offspring))

            # Apply crossover and mutation on the offspring
            for child1, child2 in zip(offspring[::2],
offspring[1::2]):
                # cross two individuals with probability CXPB
                if random.random() < CXPB:
                    toolbox.mate(child1, child2)
                    del child1.fitness.values
```

```
                    del child2.fitness.values

            for mutant in offspring:
                # mutate an individual with probability MUTPB
                if random.random() < MUTPB:
                    toolbox.mutate(mutant)
                    del mutant.fitness.values

            # Evaluate the individuals with an invalid fitness
            invalid_ind = [ind for ind in offspring if not
ind.fitness.valid]
            fitnesses = map(toolbox.evaluate, invalid_ind)
            for ind, fit in zip(invalid_ind, fitnesses):
                ind.fitness.values = fit

            print("  Evaluated %i individuals" % len(invalid_ind))

            # The population is entirely replaced by the offspring
            pop[:] = offspring

            # Gather all the fitnesses in one list and print the stats
            fits = [ind.fitness.values[0] for ind in pop]
            length = len(pop)
            mean = sum(fits) / length
            sum2 = sum(x * x for x in fits)
            print("  Min %s" % min(fits))
            print("  Max %s" % max(fits))
            print("  Avg %s" % mean)
        print("-- End of (successful) evolution --")
        best_ind = tools.selBest(pop, 1)[0]
        print("Best individual is %s, %s" % (best_ind,
best_ind.fitness.values))

class Plotter:
    def __init__(self, gens, fits):
        self.gen = gens
        self.fits = fits

    def plot(self):
        fits_plot = [max(0, self.fits[0])]
        for i in range(1, len(self.fits)):
            fits_plot.append(max(self.fits[i], fits_plot[i - 1]))
        plt.grid()
        plt.plot([i + 1 for i in self.gen], fits_plot)
        plt.xlabel("Generation")
        plt.ylabel("Net profit")
```

Appendix 9. Source-code of the pipeline for clustering-driven stock keeping units segmentation

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from itertools import cycle
from fancyimpute import KNN
from sklearn import preprocessing
from sklearn.neighbors import LocalOutlierFactor
from sklearn.decomposition import PCA
from sklearn.cluster import MeanShift, KMeans, DBSCAN,
estimate_bandwidth
from sklearn import metrics
from sklearn.metrics.pairwise import euclidean_distances
from sklearn.decomposition import FactorAnalysis
from scipy import ndimage

class Pipeline:

    def __init__(self, methods):
        self.methods = methods

    def pump(self):
        for method in self.methods:
            method
        w = Writer(pd.DataFrame(p.data))
        w.write_to_excel()

class Processing:

    def __init__(self, data, k=10, n_neighbors=20):
        self.data = data
        self.k = k
        self.n_neighbors = n_neighbors

    def knn_imputation(self):
        self.data = pd.DataFrame(KNN(self.k).fit_transform(self.data))

    def standardization(self):
        self.data = preprocessing.scale(self.data)

    def local_outlier_factor(self, drop_anomalies=True):
        lof = LocalOutlierFactor(self.n_neighbors)
        predicted = lof.fit_predict(self.data)
```

```
        data_with_outliers = pd.DataFrame(self.data)
        data_with_outliers['outliers'] = pd.Series(predicted,
index=data_with_outliers.index)

        if drop_anomalies is True:
            def drop_outliers(data):
                data = data_with_outliers
                data = data.sort_values(by=['outliers'])
                outliers_number = -data[data.outliers == -
1].sum().loc['outliers'].astype(int)
                print(outliers_number, " outliers are found")
                return data, data.iloc[outliers_number:],
outliers_number

            data_with_outliers, data_without_outliers, outliers_number
= drop_outliers(data_with_outliers)

        w = Writer(data_without_outliers, sheet='Sheet2',
file='outliers.xlsx')
        w.write_to_excel()

    def get_data(self):
        return self.data

class Reduction:

    def __init__(self, n_components=2):
        self.n_components = n_components

    def pca(self, data):
        compressor = PCA(self.n_components)
        compressor.fit(data)
        return compressor.transform(data),
compressor.explained_variance_ratio_.sum()

    def factor_analysis(self, data):
        def ortho_rotation(lam, method='varimax', gamma=None, eps=1e-
6, itermax=100):

            if gamma == None:
                if (method == 'varimax'):
                    gamma = 1.0

                nrow, ncol = lam.shape
                R = np.eye(ncol)
```

```
                var = 0

                for i in range(itermax):
                    lam_rot = np.dot(lam, R)
                    tmp = np.diag(np.sum(lam_rot ** 2, axis=0)) / nrow
* gamma
                    u, s, v = np.linalg.svd(np.dot(lam.T, lam_rot ** 3
- np.dot(lam_rot, tmp)))
                    R = np.dot(u, v)
                    var_new = np.sum(s)
                    if var_new < var * (1 + eps):
                        break
                    var = var_new
            print(var)
            print(R)
            return R

        transformer = FactorAnalysis(n_components=self.n_components,
random_state=0)
        transformed_data = transformer.fit_transform(data)
        r = ortho_rotation(transformed_data)
        transformed_data = np.matmul(r,
np.transpose(transformed_data))
        return transformed_data

class Clustering:

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

    def mean_shift_clustering(self, plot=False,
drop_small_clusters=True, threshold=4):

        def shift(data):
            bandwidth = estimate_bandwidth(data, quantile=0.2,
n_samples=500)
            ms = MeanShift(bandwidth=bandwidth, bin_seeding=True)
            ms.fit(data)
            return ms.labels_, ms.cluster_centers_, pd.DataFrame(data)

        labels, cluster_centers, labeled_data = shift(self.data)

        old_id = labeled_data.index
        iteration = 1
        while drop_small_clusters is True and iteration<4:
            dropped = 0
```

```
            labeled_data['mean-shift'] = pd.Series(labels)
            labeled_data['clusters_sorted'] =
pd.Series(labels).value_counts()

            to_drop = []
            labeled_data.reset_index(drop=True)

            for cluster in labeled_data.index:

                if labeled_data.loc[cluster, 'clusters_sorted']<
threshold:

                    for row in range(0, len(labeled_data['mean-
shift'])):
                        if labeled_data.loc[row, 'mean-shift'] ==
cluster:
                            to_drop.append(row)

            for i in to_drop:
                labeled_data = labeled_data.drop(i)
                dropped += 1

            labeled_data = labeled_data.drop(['clusters_sorted',
'mean-shift'], axis=1)
            labels, cluster_centers, labeled_data =
shift(labeled_data)
            iteration += 1
            print("iteration: ", iteration, "clusters: ",
max(labels)+1, "dropped: ", dropped)


        labeled_data['mean-shift'] = pd.Series(labels)
        labeled_data['clusters_sorted'] =
pd.Series(labels).value_counts()
        w = Writer(labeled_data, sheet='Sheet2', file='labeled.xlsx')
        w.write_to_excel()

        labeled_data = labeled_data.drop('mean-shift', axis=1)
        labeled_data = labeled_data.drop('clusters_sorted', axis=1)
        self.data = labeled_data
        labels_unique = np.unique(labels)
        n_clusters_ = len(labels_unique)
        print("mean-shift ", '\n', "number of estimated clusters: {}
".format(n_clusters_))
        e = Evaluation(self.data, labels)
        e.evaluate()
```

```
        if plot is True:
            red = Reduction(n_components=2)
            to_plot, variance = red.pca(labeled_data)
            plotter = Plotter()
            plotter.plot_clustering(to_plot, n_clusters_, labels,
cluster_centers, variance, 'mean-shift')

    def k_means_clustering(self, k, plot_best=True, compress=0,
method='pca'):
        if compress != 0:
            if method=='pca':
                r = Reduction(compress)
                self.data, variance = r.pca(self.data)
                print(variance)
            else:
                r = Reduction(compress)
                self.data = r.factor_analysis(self.data)

        km = KMeans(n_clusters=k, random_state=0).fit(self.data)
        labels = km.labels_
        cluster_centers = km.cluster_centers_

        if plot_best is True:
            red = Reduction(n_components=2)
            to_plot, variance = red.pca(self.data)
            labels_unique = np.unique(labels)
            n_clusters_ = len(labels_unique)
            plotter = Plotter()
            plotter.plot_clustering(to_plot, n_clusters_, labels,
cluster_centers, variance, 'K-means')

        e = Evaluation(self.data, labels)
        print(k, '-means')
        e.evaluate()

    def dbscan(self, eps, plot_best=False):
        db = DBSCAN(eps=eps, min_samples=10).fit(self.data)
        core_samples_mask = np.zeros_like(db.labels_, dtype=bool)
        core_samples_mask[db.core_sample_indices_] = True
        labels = db.labels_
        n_clusters_ = len(set(labels))

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

```
        e = Evaluation(self.data, labels)
        print('DBSCAN ', 'eps=', eps, '\n', n_clusters_, ' clusters
found')
        print(list(labels).count(-1), " observations considered as
noise")
        e.evaluate()

        if plot_best is True:
            red = Reduction(n_components=2)
            to_plot, variance = red.pca(self.data)
            labels_unique = np.unique(labels)
            n_clusters_ = len(labels_unique)
            plotter = Plotter()
            plotter.plot_clustering(to_plot, n_clusters_, labels,
np.nan, variance, 'DBSCAN')

class Evaluation:

    def __init__(self, data, labels, metric='euclidean'):
        self.data = data
        self.labels = labels
        self.metric = metric

    def silhouette(self):
        return metrics.silhouette_score(self.data, self.labels,
metric=self.metric)

    def calinski_harabaz(self):
        return metrics.calinski_harabaz_score(self.data, self.labels)

    def dunn_index(self):
        def normalize_to_smallest_integers(labels):
            max_v = len(set(labels))

            sorted_labels = np.sort(np.unique(labels))
            unique_labels = range(max_v)
            new_c = np.zeros(len(labels), dtype=np.int32)
            for i, clust in enumerate(sorted_labels):
                new_c[labels == clust] = unique_labels[i]
            return new_c

        def dunn(labels, distances):
            labels = normalize_to_smallest_integers(labels)

            unique_cluster_distances =
np.unique(min_cluster_distances(labels, distances))
```

```
            max_diameter = max(diameter(labels, distances))

            if np.size(unique_cluster_distances) > 1:
                return unique_cluster_distances[1] / max_diameter
            else:
                return unique_cluster_distances[0] / max_diameter

        def min_cluster_distances(labels, distances):
            labels = normalize_to_smallest_integers(labels)
            n_unique_labels = len(np.unique(labels))

            min_distances = np.zeros((n_unique_labels,
n_unique_labels))
            for i in np.arange(0, len(labels) - 1):
                for ii in np.arange(i + 1, len(labels)):
                    if labels[i] != labels[ii] and distances[i, ii] >
min_distances[labels[i], labels[ii]]:
                        min_distances[labels[i], labels[ii]] =
min_distances[labels[ii], labels[i]] = distances[i, ii]
            return min_distances

        def diameter(labels, distances):
            labels = normalize_to_smallest_integers(labels)
            n_clusters = len(np.unique(labels))
            diameters = np.zeros(n_clusters)

            for i in np.arange(0, len(labels) - 1):
                for ii in np.arange(i + 1, len(labels)):
                    if labels[i] == labels[ii] and distances[i, ii] >
diameters[labels[i]]:
                        diameters[labels[i]] = distances[i, ii]
            return diameters

        return dunn(self.labels, euclidean_distances(self.data))

    def evaluate(self):
        coeff = ['Silhouette: ', self.silhouette(), 'Calinski-Harabaz:
',
                 self.calinski_harabaz(), 'Dunn: ', self.dunn_index()]
        print(coeff)

class Plotter:

    def plot_clustering(self, data, n_clusters_, labels,
cluster_centers, variance, name):
        plt.figure()
```

```
        plt.rc('font', size=14)
        colors = cycle('bgrcmykbgrcmykbgrcmykbgrcmyk')
        if -1 in labels:
            for k, col in zip(range(-1, n_clusters_-1), colors):
                my_members = labels == k
                if k==-1:
                    plt.plot(data[my_members, 0], data[my_members, 1],
col + '+')
                else:
                    plt.plot(data[my_members, 0], data[my_members, 1],
col + '.')
        else:
            for k, col in zip(range(n_clusters_), colors):
                my_members = labels == k
                plt.plot(data[my_members, 0], data[my_members, 1], col
+ '.')
        plt.title('Algoritm: {} Number of clusters: {}. \n'
                  '{}% of variance is preserved after
PCA'.format(name, n_clusters_, round(variance*100, 2)))
        plt.show()

class Writer:

    def __init__(self, data, sheet='Sheet1', file='new_data.xlsx'):
        self.data = data
        self.sheet = sheet
        self.file = file
        self.writer = pd.ExcelWriter(self.file, engine='xlsxwriter')

    def write_to_excel(self):
        self.data.to_excel(self.writer, sheet_name=self.sheet)
        self.writer.save()

file = pd.ExcelFile("initial_data.xlsx")
data = file.parse("Sheet1")
data[['Unitprice', 'Expire date', 'Pal grossweight', 'Pal height',
      'Units per pal']] = data[['Unitprice', 'Expire date', 'Pal
grossweight','Pal height', 'Units per pal']].replace(0.0, np.nan)
data = data.drop(["ID",    "Tradability",  "Init status"], axis=1)

p = Processing(data, 10)
preprocessing_methods = [p.knn_imputation(), p.standardization(),
p.local_outlier_factor(drop_anomalies=True)]
pipe1 = Pipeline(preprocessing_methods)
pipe1.pump()
c = Clustering(p.get_data())
```