Pod autoscaling in Kubernetes clusters

Master thesis by Angelina Horn
Date of submission: February 18, 2021

1. Review: Prof. Dr. Felix Wolf
2. Review: Dr. Hamid Mohammadi Fard
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
Parallel Programming

Erklärung zur Abschlussarbeit gemäß §22 Abs. 7 APB TU Darmstadt

Hiermit versichere ich, Angelina Horn, die vorliegende Masterarbeit gemäß §22 Abs. 7 APB der TU Darmstadt ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs. 2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.

Bei einer Thesis des Fachbereichs Architektur entspricht die eingereichte elektronische Fassung dem vorgestellten Modell und den vorgelegten Plänen.

Darmstadt, 18. Februar 2021

A. Horn

Contents

1 Introduction

1.1 Motivation

Lately, microservice architecture is gaining popularity, especially among developers[1]. The microservice architecture describes a new and highly flexible way of application architecture and deployment. For this approach, an application gets split into smaller but independent chunks called microservices. The created microservices will be deployed, for example as docker containers in a server cluster. These clusters will often not be maintained or owned by the applications developer but rather by an enterprise cloud provider like Google or Amazon. Microservices that belong to one deployment can communicate with one another via lightweight API interfaces e.g. REST.

Applications will have a dynamic load based on their workload [2]. This load however is not distributed equally among all microservices. Therefore, it is necessary to be able to scale the microservices individually while they are under load to maintain the performance of the whole application. Additionally, because of limited and costly resources, one wants to use only the minimal amount of resources that are necessary to sustain performance.

This can be achieved via automatic scaling [3]. Automatic scaling describes a mechanism which periodically monitors every resource of a microservice, determines if and how a container should be scaled based on metrics defined by the developer and executes the scaling automatically. Because all these described architectures and ways of deployment are rather new there is room for research and improvement regarding this topic.

1.2 Objective

The objective of this thesis is to implement a novel autoscaling approach in Kubernetes to maintain performance measured by the response time of a given microservice. Additionally, this approach should also decrease the monetary cost while possibly using public cloud resources. Therefore the objective can be divided into three separated goals:

1. Depp study and research of Kubernetes and available autoscaling methods.
2. Development and implementation of a novel autoscaling approach using machine learning.
3. Evaluation of the developed approach, including comparison, with minimizing response time and costs in mind.

2 Background

In this chapter of the thesis, basic topics of which their understanding is necessary will be pointed out and explained. These topics are microservices, Kubernetes and autoscaling. Furthermore, three machine learning approaches will be explained. These topics will build the foundation and background needed for this thesis following approaches, research and statements.

2.1 Microservices

The term "microservice" was first introduced by James Lewis in a conference [4]. It describes an application architecture which is characterized by subdivided applications, that communicate with each other trough lightweight interfaces.
Each of these applications has a single responsibility or task and is deployed independently. This means that each of these microservices can be implemented in a different programming language and can be deployed on an entirely different operating system on a different server as long as they share a common interface and can communicate with each other through it. James Lewis describes this approach as "[...] rooted in the Unix Philosophy of small and simple" [4].

On one hand, this architecture offers several advantages compared to the monolithic application architecture which was popular prior (see figure 2.1). First and foremost it provides modularity which is highly beneficial regarding code complexity as well as team coordination [5]. The introduction phase of new programmers is highly decreased as well as the possibilities to work on different parts of the overall application at the same time.

Secondly, this kind of architecture makes it possible to incrementally upgrade an application and therefore possible to reduce time to market [5]. Certain parts of the application

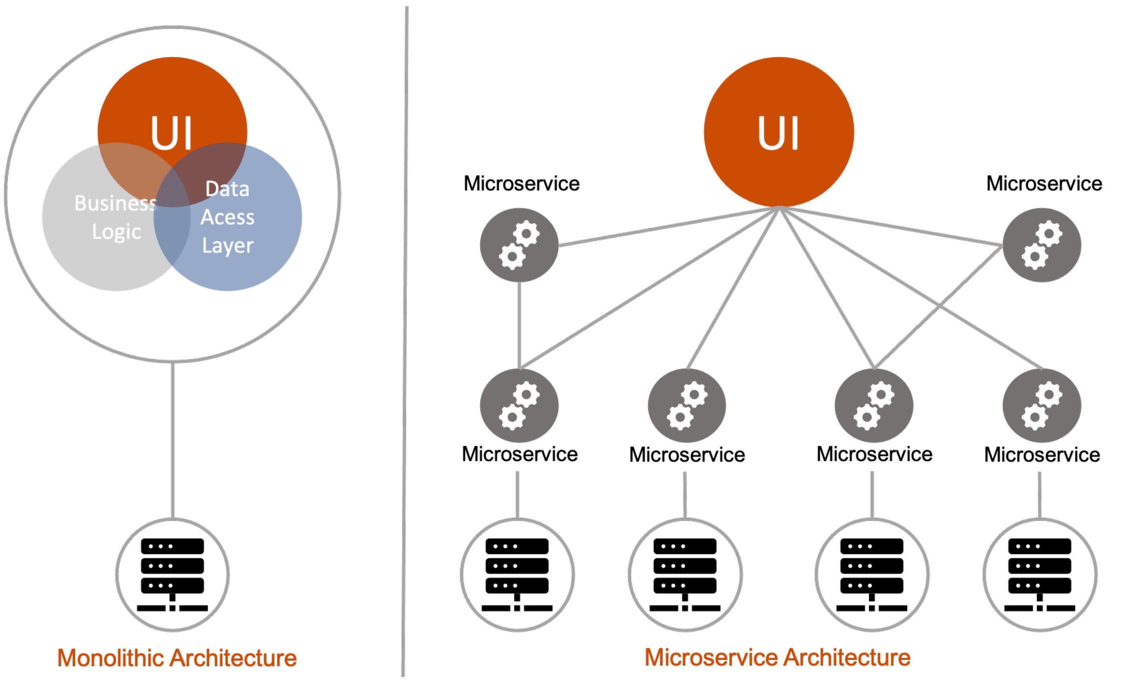


Figure 2.1: Monolithic and microservice architectures

can be updated independently from the other microservices regarding not only the application code itself but also its dependencies such as libraries and environments. This way the so-called "dependency hell" [6] which many monolithic applications suffer from can be avoided.

Furthermore, microservices are more resistant to failure due to their containerized existence. If one microservice fails it does not necessarily lead to the failure of the whole application. Additionally, this failed microservice can be easily restarted in a short amount of time because of its lightweight nature.

Last but not least, microservices introduce a new capability of scaling [5], [7]. Regarding the application, each microservice is under inconsistent load and can be scaled individually. This allows an optimal usage of resources in dependency of desired performance and costs.

On the other hand, this architecture brings certain disadvantages. One major disadvantage is the time it costs to convert any application in several microservices [5]. This time should not be underestimated but can also bring major benefits as stated above. Another disadvantage is the complexity of the deployment [5]. Today there are several ways to deploy microservices including the use of Jenkins[1] or lately the use of orchestral cluster tools such as Docker Swarm[2] and Kubernetes[3].

2.2 Kubernetes

The term Kubernetes gained a lot of popularity over the past five years (see figure 2.2). It is the name of an orchestral tool for containerized applications, originally designed by Google and now owned by the Cloud Native Computer Foundation[4]. Its capabilities include deployment, scaling and monitoring of containerized microservices in clusters. These clusters can be hosted in a public or private cloud.

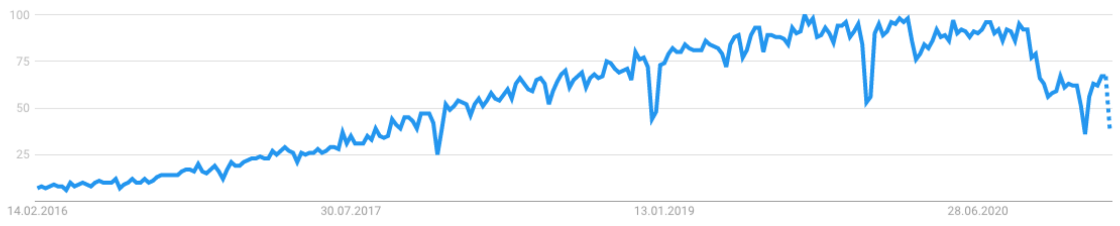


Figure 2.2: Google Trends report for the keyword "Kubernetes" [8]

Deployment Containerized microservices are mostly available as so-called docker images. The configuration of an application based on several microservices gets described in a "yaml" file which can be read by Kubernetes. In it several aspects of the deployment can be configured - examples are the ports on which the microservice is reachable, the number of replicas that should be deployed or the application to which the

[1]https://www.jenkins.io
[2]https://docs.docker.com/engine/swarm
[3]https://kubernetes.io
[4]https://www.cncf.io

microservice belongs. Once this configuration file is made the deployment of this particular microservice is automated.

Scaling There are two general ways to realize the scaling of a microservice. The first one is to scale a microservice manually by for example defining the resources that are available to an application or define the number of replicas in its configuration file. The second and more complex method is the automatic scaling of microservices.

Monitoring To provide stability of an application it is necessary to monitor its behavior. This enables the discovery of failures, resource use and load on the individual microservices of the application. This information is important to maintain an application. Kubernetes offers several automatic services to support maintenance. One of them is the capability to restart failed microservices according to certain settings automatically.

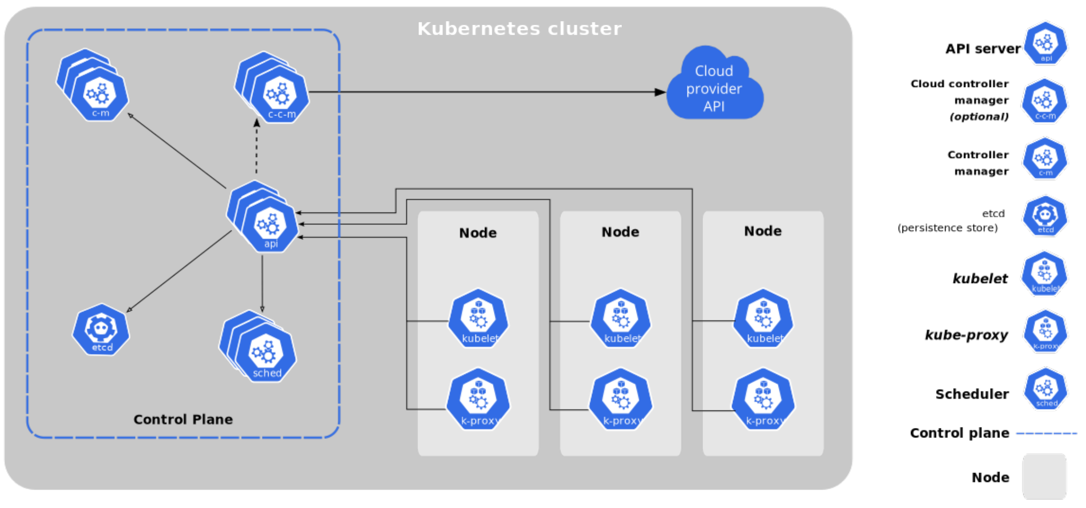


Figure 2.3: Kubernetes cluster overview [9]

A Kubernetes cluster consists of two main structures (see figure 2.3). The first one is called "Master" and contains the API interface, the controller manager, the scheduler and configuration storage called "etcd". The API interface is the connection point between the developer and Kubernetes. The controller manager is the executing unit of it and the scheduler plans its actions.
The second important structure of a Kubernetes cluster is the node. In contrary to the master there can be several nodes in a cluster. Each node contains several Pods. Each pod

represents one microservice. Furthermore, each pod contains a "Kubelet" which is the executing unit of a node similar to the controller manager in the master. Additional the "cAdvisor" unit monitors the pods of a node and the "Kube-Proxy" is the connection point between a pod and the users.

Kubernetes offers a very powerful tool for the life cycle of an application in a cluster and paths the way for extensive development of more efficient methods to develop and maintain applications.

2.3 Autoscaling

The term Autoscaling describes the automatic resource scaling of applications and more detailed microservices [10], [11]. As described in the chapter about Kubernetes: autoscaling represents the counterpart to the traditional manual scaling of applications. With this method, it is possible to allocate more resources for one microservice when it is under especially heavy load or allocate fewer resources when the load gets easier depending on its degree of use by the user.

MAPE loop The acronym "MAPE" stands for the terms monitor, analysis, planning and execution. They describe the basic loop that every autoscaler follows (see figure 2.4). The first step is monitoring. This includes fetching all the metrics that are available for a given application. The second step is to analyze the fetched data regarding for example if certain metrics exceed a given threshold. Following this, it has to be planned on how to proceed for example to increase a certain parameter. In the last step of the loop, the planned measure has to be executed. After that, the loop is completed by returning to the monitoring step. Mostly the MAPE loop is executed and repeated in a specified period.

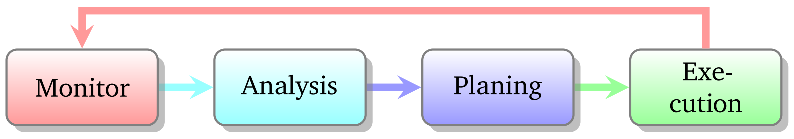


Figure 2.4: MAPE loop

There are several methods considering and categorizing amongst other things when, how and where to scale. An organized overview gives the following taxonomy (see figure 2.5).

Scaling methods There are in general two types of scaling methods: horizontal and vertical [10], [11]. A microservice gets scaled horizontally when its number of instances gets increased or decreased and a microservice gets scaled vertically when the resources of one instance are being increased or decreased.

Scaling indicators

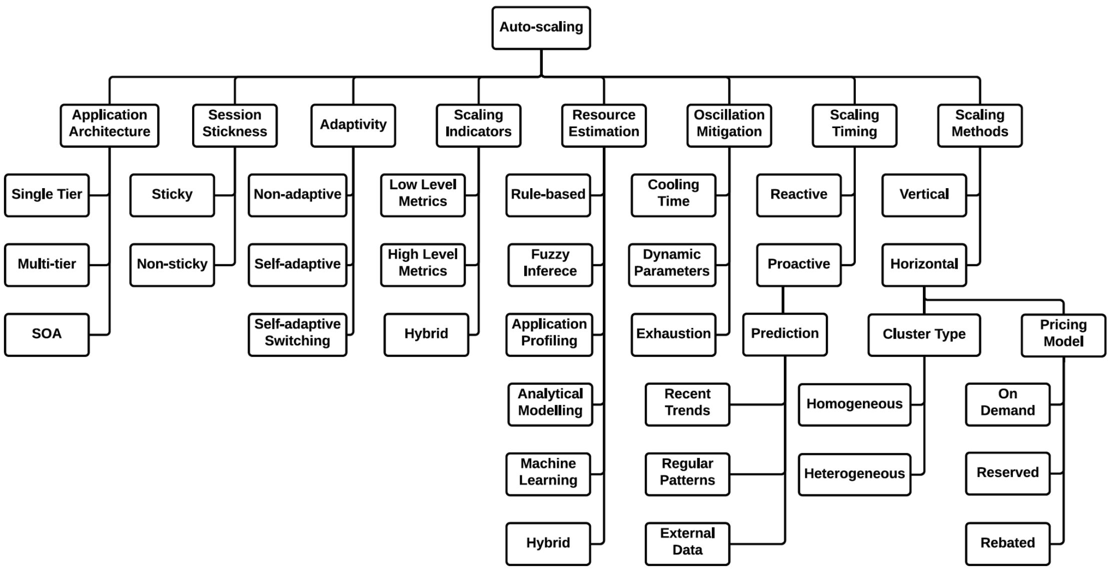


Figure 2.5: Autoscaling taxonomy [10]

Resource estimation

Scaling timing

2.4 Machine Learning

In this chapter, a selection of related machine learning approaches are presented. Each method represents a different complexity and application level of machine learning. All of these methods will be used and compared with the same dataset later in this thesis.

2.4.1 Linear Regression

Linear Regression is originally a statistical method which attempts to model the relationship between one dependent variable and at least one independent variable [12]. It does so by fitting a linear equation to the given data. The dependent variable depends on the independent or explanatory variables. The formal description of the linear equation (see 2.1):

$$Y = a + b \cdot X \tag{2.1}$$

The Y term represents the dependent and X the explanatory variable. The value of a is the y-axis intercept and b is the gradient of the resulting line. There are several approaches to fitting a regression line. The basic methods are least-squares and maximum-likelihood estimation.

Least-Squares estimation The most common one is the method of least-squares. It minimizes the sum of the squares of the vertical difference from each point to the line by minimizing the quadratic loss function (see 2.2). The variable p_i represents the dependent variable y_i corresponding point on the regression line. There are no cancellations of positive and negative values because the differences are first squared and then summed. Related methods are ordinary least-squares, weighted least-squares and generalized least-squares estimators [12].

$$L = \sum_{i=1}^{N} (y_i - p_i)^2 \tag{2.2}$$

Maximum-likelihood estimation This approach searches for a set of parameters which fulfils a certain likelihood function [13]. This likelihood function represents the conditional probability to observe the given data, where a specific probability distribution and its parameters are set. This function can be interpreted as the multiplication of the conditional probability for observing each data point given the same distribution parameters. Formally this function uses the natural logarithm because of its mathematical stability and is also called the log-likelihood function (see 2.3).

$$log\, L(\theta) = log\, p(X|\theta) = \sum_{n=1}^{N} log\, p(x_n|\theta) \tag{2.3}$$

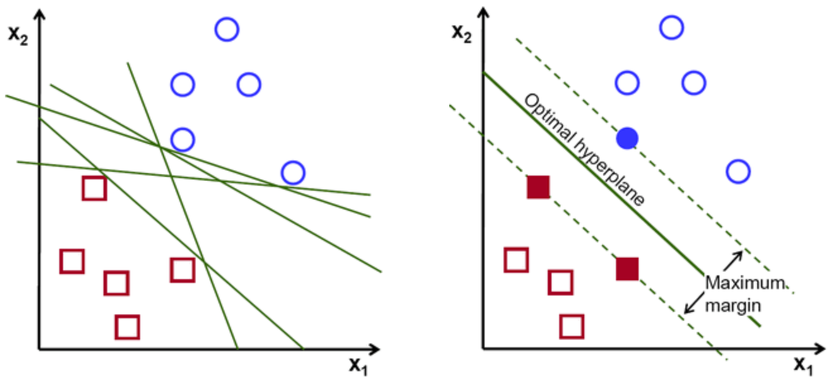
Because linear regression tries to model the relationship between certain variables, one should first determine if the relationship between these variables are of interest. This can be accomplished by for example using visualization methods like scatterplots or calculating the correlation coefficient corresponding to the given variables. A high correlation coefficient or a visible trend in the scatterplot can indicate the level of association between variables [12], [14].

When the regression line is calculated, there can be data points that lie far away from the fitted line. These are called outliners. Outliners can represent for example measurement errors or indicate a not optimal regression line. These outliners can have a massive impact on the gradient of the computed line and are therefore called influential observation [12], [14].

2.4.2 Support Vector Machines

The Support Vector Machine theory was introduced by Vapnik of the AT&T Bell Laboratories in 1995 [15]. It describes a supervised machine learning model which is used for classification and regression.
The SVM algorithm is designed to find a hyperplane in a n-dimensional space which distinguishes given data points [16]. In general, several hyperplanes are possible, but this algorithm picks the hyperplane with the constraint such that it provides the maximal distance between itself and each class of data points (see figure 2.6). This distance is also called margin, while the nearest data points from each class, that determine the maximal margin and therefore influence the hyperplane, are called support vectors. The dimension of the hyperplane depends on the number of features R^n. A hyperplane in for example R^2 represents a line while it represents a two-dimensional plane in R^3 [16].
In order to handle non-linear problems it uses the kernel trick to transfer the function into higher dimensional feature space [18]. Therefore the used kernel function influences the performance of the SVM algorithm significantly and should hence be chosen wisely. The most common kernel functions are "polynomials, radial basis functions and certain sigmoid functions" [18].
Support Vector Machines can, as mentioned before, also be used for regression. To realize this, it is necessary to introduce a loss function that includes a distance measure [18]. Possible loss functions are for example Quadratic, Laplace or Huber functions (see figure 2.7).

2.4.3 Neural Networks



(a) Possible Hyperplanes

(b) Hyperplane with maximum margin

Figure 2.6: SVM Hyperplanes [17]

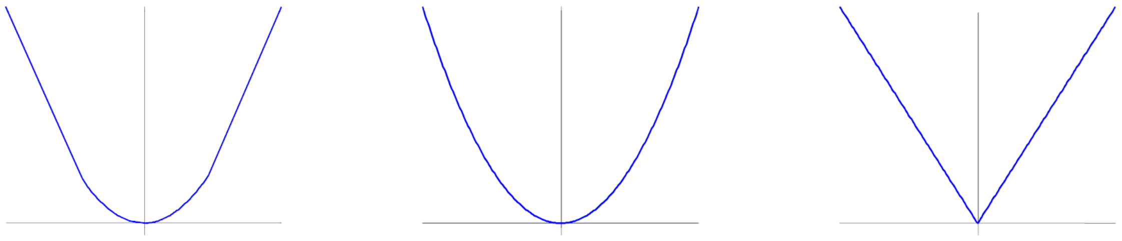


Figure 2.7: SVR loss functions [18]

3 Related Work

4 Method

4.1 Overview

4.2 Environment Setup

4.2.1 Kubernetes

Linkerd

Prometheus

4.2.2 Python

K8s tools

Benchmark

Formatting

4.2.3 Microservice

Robot shop

4.3 Implementation

4.3.1 GUI

4.3.2 Benchmark

4.3.3 Machine Learning Model

Selection of parameters * CPU utilization * Memory utilization * Number of pods Selection of metric: * Response time * Support Vector Machine

4.3.4 Control Script

4.4 Experimentation Methodology

comparision of performance models

5 Evaluation

5.1 Results

5.2 Analysis

6 Conclusion

6.1 Summary

6.2 Outlook

References

[1] N. Alshuqayran, N. Ali, and R. Evans, “A systematic mapping study in microservice architecture”, in *2016 IEEE 9th International Conference on Service-Oriented Computing and Applications (SOCA)*, IEEE, 2016, pp. 44–51.

[2] M. Villamizar, O. Garcés, H. Castro, M. Verano, L. Salamanca, R. Casallas, and S. Gil, “Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud”, in *2015 10th Computing Colombian Conference (10CCC)*, IEEE, 2015, pp. 583–590.

[3] F. Rossi, M. Nardelli, and V. Cardellini, “Horizontal and vertical scaling of container-based applications using reinforcement learning”, in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, IEEE, 2019, pp. 329–338.

[4] J. Lewis, “Microservices-java, the unix way”, in *Proceedings of the 33rd Degree Conference for Java Masters*, 2012.

[5] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, “Microservices: Yesterday, today, and tomorrow”, in *Present and ulterior software engineering*, Springer, 2017, pp. 195–216.

[6] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment”, *Linux journal*, vol. 2014, no. 239, p. 2, 2014.

[7] D. Namiot and M. Sneps-Sneppe, “On micro-services architecture”, *International Journal of Open Information Technologies*, vol. 2, no. 9, pp. 24–27, 2014.

[8] “Google trends”, Google LLC. (Oct. 2020), [Online]. Available: `https://trends.google.com`.

[9] “Kubernetes documentation”, Kubernetes. (Nov. 2020), [Online]. Available: `https://kubernetes.io/docs/`.

[10] C. Qu, R. N. Calheiros, and R. Buyya, “Auto-scaling web applications in clouds: A taxonomy and survey”, *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–33, 2018.

[11] D. Midigudla, *Performance analysis of the impact of vertical scaling on application containerized with docker: Kubernetes on amazon web services-ec2*, 2019.

[12] D. Montgomery, E. Peck, and G. Vining, *Introduction to Linear Regression Analysis*, ser. Wiley Series in Probability and Statistics. Wiley, 2021, ISBN: 9781119578741. [Online]. Available: `https://books.google.de/books?id=N8EZEAAAQBAJ`.

[13] P. D. K. Kersting, *Statistical machine learning, lecture 06: Probability density estimation*, Accessed: 2021–02-10, 2020. [Online]. Available: `https://www.ml.informatik.tu-darmstadt.de/lectures/sml2020sose/slides/lecture_06_Probability_density_estimation.pdf`.

[14] "Linear regression". (), [Online]. Available: `http://www.stat.yale.edu/Courses/1997-98/101/linreg.htm` (visited on 02/10/2021).

[15] V. N. Vapnik, "The nature of statistical learning theory", *New York: Springer Verlag*, 1995.

[16] M. Awad and R. Khanna, *Efficient learning machines: theories, concepts, and applications for engineers and system designers*. Springer Nature, 2015.

[17] Y. Lin, H. Yu, F. Wan, and T. Xu, "Research on classification of chinese text data based on svm", in *IOP Conference Series: Materials Science and Engineering*, 2017, pp. 1–5.

[18] S. R. Gunn *et al.*, "Support vector machines for classification and regression", *ISIS technical report*, vol. 14, no. 1, pp. 5–16, 1998.