

Preface

In recent years, machine learning has been applied in different areas of science and engineering including computer science, medical science, cognitive science, psychology, and so on. In these fields, machine learning non-specialists utilize it to address their problems.

One of the most popular family of algorithms in machine learning is Support Vector Machine (SVM) algorithms. Traditionally, these algorithms are used for binary classification problems. But recently, the SVM algorithms have been utilized in various areas including numerical analysis, computer vision, and so on. Therefore, the popularity of SVM algorithms have risen in recent years.

The main part of the SVM algorithms is the kernel function and the performance of a given SVM is related to the power of the kernel function. Different kernels provide different capabilities to the SVM algorithms therefore, understanding the properties of the kernel functions is a crucial part of utilizing the SVM algorithms. Up until now, various kernel functions have been developed by researchers. One of the most significant families of the kernel functions is the orthogonal kernel function which has been attracting much attention. The computational power of this family of kernel functions has been illustrated by the researchers in the last few years. But despite the computational power of orthogonal kernel functions they have not been used in real application problems. This issue has various reasons, some of which are summarized in the following:

1. The mathematical complexity of orthogonal kernel functions formulation,
2. Lack of a simple and comprehensive resource for expressing the orthogonal kernels and their properties,
3. Implementation difficulties of these kernels and lack of a convenient package that implements these kernels.

For the purpose of solving the aforementioned issues, in this book, we are going to present a simple and comprehensive tutorial on orthogonal kernel functions and a Python package that is named ORSVM that contains the orthogonal kernel functions. The reason we chose Python as the language of the ORSVM package are:

1. Python is open source, very popular, easy to learn, and there are lots of tutorial for it,
2. Python has a lot of packages for manipulating data and they can be used besides the ORSVM for solving a machine learning problem,
3. Python is a multi-platform language and can be launched on different operating systems.

In addition to the developed orthogonal kernels, we aim to introduce some new kernel functions which are called fractional orthogonal kernels. The name fractional comes from the order x being a positive real number instead of being an integer. In fact, the fractional orthogonal kernels are extensions of integer order orthogonal functions. All introduced fractional orthogonal kernels in this book are implemented in the ORSVM package and their performance is illustrated by testing on some real datasets.

This book contains 13 chapters, including 1 appendixes which are brought at the end of the book for covering programming preliminaries.

The first chapter includes the fundamental concepts of machine learning. In this chapter, we explain the definitions of pattern and similarity and then a geometrical intuition of the SVM algorithm is presented. At the end of this chapter, a historical review of the SVM and the current applications of SVM are discussed.

In the second chapter, we present the basics of SVM and least-squares SVM (LS-SVM). The mathematical background of SVM is presented in this chapter in detail. Moreover, Mercier's theorem and kernel trick are discussed too. In the last part of this chapter, function approximation using the SVM is illustrated.

In the third chapter, the discussion is about Chebyshev polynomials. At first, the properties of Chebyshev polynomials and fractional Chebyshev functions are explained. After that, a review of Chebyshev kernel functions is presented and the fractional Chebyshev kernel functions are introduced. In the final section of this chapter, the performance of fractional Chebyshev kernels on real data sets is illustrated and compared with other state-of-the-art kernels.

In the fourth chapter, the Legendre polynomials are considered. In the beginning, the properties of the Legendre polynomials and fractional Legendre functions are explained. In the next step after reviewing the Legendre kernel functions, the fractional Legendre kernel functions are introduced. Finally, the performance of fractional Legendre kernels is illustrated by applying them to real datasets.

Another orthogonal polynomial series is discussed in chapter 5; “The Gegenbauer polynomials”. Similar to the previous chapters, this chapter includes properties of the Gegenbauer polynomials, properties of the fractional Gegenbauer functions, a review on Gegenbauer kernels, introducing fractional Gegenbauer kernels, and showing the performance of fractional Gegenbauer kernels on real datasets.

In the sixth chapter, we focus on Jacobi polynomials. This family of polynomials are the general form of the previous polynomials which are presented in chapters 3, 4, and 5. Therefore, the relations between the polynomials and the kernels are discussed in this chapter. In Addition to the relations between the polynomials, other parts of this chapter are similar to the three previous chapters.

In chapter seven and eight, some applications of the SVM in scientific computing are presented and the procedure of using LS-SVM for solving ordinary/partial differential equations is explained. Mathematical basics of ordinary/partial differential equations and traditional numerical algorithms for approximating the solution of ordinary/partial differential equations are discussed in these chapters too.

Chapter nine consists of the procedure of using the LS-SVM algorithm for solving integral equations, basics of integral equations, traditional analytical and numerical algorithms for solving integral equations, and a numerical algorithm based on LS-SVR for solving various kinds of integral equations.

Another group of dynamic models in real-world problems are distributed-order fractional equations, which have received much attention recently. In chapter 10, we discuss in detail the numerical simulation of these equations using LS-SVM based on orthogonal kernels, and evaluate the power of this method in fractional dynamic models.

The aim of chapter eleven is parallelizing and accelerating the SVM algorithms that are using orthogonal kernels using graphical processing units GPUs.

Finally, we have also released an online and free software package called orthogonal SVM (ORSVM). In fact, ORSVM is a free package that provides an SVM classifier with some novel orthogonal kernels. This library provides a complete path of using the SVM classifier from normalization to calculation of SVM equation and the final evaluation. In the last chapter, a comprehensive tutorial on the ORSVM package is presented. Also, for more information and to use this package please visit: orsvm.readthedocs.io.

Since the goal of this book is that the readers use the book without needing any other resources, a short tutorial on Python programming is presented in the appendix. This tutorial on Python programming is suitable for those who are not familiar with Python programming language. In this appendix, some popular packages that are used for working with data such as NumPy, Pandas, and Matplotlib are explained with some examples.

The present book is written for undergraduate as well as graduate students and who want to work with the SVM method for their machine learning problems. Moreover, it can be useful to scientists who also work on applications of orthogonal functions in different fields of science and engineering. The book will be self-contained and there is no need to have any advanced background in mathematics or programming.

Tehran, Iran
Waterloo, Canada
Rourkela, India

*Jamal Amani Rad
Kourosh Parand
Snehashish Chakraverty*

Contents

Part I Basics of Support Vector Machines

- 1 Introduction to SVM 3
Hadi Veisi
- 2 Basics of SVM Method and Least Squares SVM 19
Kurosh Parand and Fatemeh Baharifard and Alireza Afzal Aghaei and Mostafa Jani

Part II Special Kernel Classifiers

- 3 Fractional Chebyshev Kernel Functions: Theory and Application . . . 39
Amir Hosein Hadian Rasanan and Sherwin Nedaei Janbesaraei and Dumitru Baleanu
- 4 Fractional Legendre Kernel Functions: Theory and Application . . . 71
Amirreza Azmoon and Snehashish Chakraverty and Sunil Kumar
- 5 Fractional Gegenbauer Kernel Functions: Theory and Application . . 95
Sherwin Nedaei Janbesaraei and Amirreza Azmoon and Dumitru Baleanu
- 6 Fractional Jacobi Kernel Functions: Theory and Application . . . 123
Amir Hosein Hadian Rasanan and Jamal Amani Rad and Malihe Shaban and Abdon Atangana

Part III Applications of orthogonal kernels

- 7 Solving Ordinary Differential Equations by LS-SVM 151
Mohsen Razzaghi and Simin Shekarpaz and Alireza Rajabi
- 8 Solving Partial Differential Equations by LS-SVM 177
Mohammad Mahdi Moayeri and Mohammad Hemami

- 9 Solving Integral Equations by LS-SVR** 205
Kourosh Parand and Alireza Afzal Aghaei and Mostafa Jani and Reza Sahleh
- 10 Solving Distributed-Order Fractional Equations by LS-SVR** 231
Amir Hosein Hadian Rasanan and Arsham Gholamzadeh Khoei and Mostafa Jani

Part IV Orthogonal kernels in action

- 11 GPU Acceleration of LS-SVM, Based on Fractional Orthogonal Functions** 253
Armin Ahmadzadeh, Mohsen Asghari, Dara Rahmati, Saeid Gorgin, and Behzad Salami
- 12 Classification Using Orthogonal Kernel Functions: Tutorial on ORSVM Package** 271
Amir Hosein Hadian Rasanan and Sherwin Nedaei Janbesarai and Amirreza Azmoon and Mohammad Akhavan and Jamal Amani Rad

Part V Appendixes

- A Python Programming Prerequisite** 285
Mohammad Akhavan

Editors and Contributors

About the Editors:

Jamal Amani Rad

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran.

Jamal Amani Rad received a Ph.D. in applied mathematics (scientific computing) from the Department of Computer Science at Shahid Beheshti University (SBU) in October 2015. Following his Ph.D., he started a one-year computational cognitive modeling postdoctoral fellowship at Institute for Cognitive and Brain Sciences in May 2016 where he is currently midway through his fourth year as an assistant professor. In addition to having at his relatively young age 81 publications in quality journals/conferences, he has worked independently during his five years after graduation. Over the past 10 years of his academic research life (since he started his MSc in scientific computing in the department of computer science), his research path has grown step by step with some jumps. In his master's degree in computer science, he first became interested in numerical methods, especially meshless approaches & radial basis functions, to simulate dynamic models, so he was able to publish several high-level articles, such as "Numerical solution of non-linear Volterra–Fredholm–Hammerstein integral equations via collocation method based on radial basis functions, Applied Mathematics and Computation". Later, in his Ph.D., he focused on computational finance using numerical meshless methods, entitled "Meshless methods for option pricing in financial mathematics" so that he could publish several top-level articles such as "Pricing European and American options by radial basis point interpolation, Applied Mathematics and Computation", and "Forward deterministic pricing of options using Gaussian radial basis functions, Journal of Computational Science". It was there that he realized that traditional numerical methods could not be used in many real applications (such as stochastic high-dimensional partial differential equations like baskets of 20 financial stocks), so as a one-year postdoctoral fellow, he entered a project on mathematical modeling of human cognitive processes at the Institute for Cognitive and Brain Sciences (ICBS)

where he is currently midway through his fourth year as an assistant professor. It was here that he realized that it is very important for him to model human decisions as well as reinforcement learning from the reward and punishment of individuals. More interestingly, for this new way, he was able to use all the knowledge gained in this path, namely numerical solutions, stochastic dynamics, financial modeling, neural networks, and deep learning to estimate parameters, etc. he has also worked well with psychological and neurological theories on human decisions to build mathematical models over the past five years.

Kourosh Parand

Department of Statistics and Actuarial Science, University of Waterloo, Canada.
Kourosh Parand received his Ph.D. in Applied Mathematics – Numerical Analysis and Control from the Amirkabir University of Technology, Iran in 2004. Professor Parand is the academic staff at the department of Data & Computer Science in the Faculty of Mathematics and also Institute for Cognitive and Brain Sciences (ICBS), Shahid Beheshti University, Tehran, Iran. He has an outstanding and unique curriculum vitae with experience in numerical analysis (spectral methods, meshless methods, etc.), cognitive science (epilepsy), mathematical physics, neural networks, deep learning, and support vector machines. Kourosh Parand is a world leader in applying methods of spectral methods as well as machine learning approaches to nonlinear dynamical models and his H-index is 30. During his academic life, he was the chair of the department of Data & Computer Science, director of the science and technology park (2018–2019), and also a member of the board of auditors (2020–present), at Shahid Beheshti University and he is awarded as the best researcher multiple times. Moreover, he was a sabbatical at the University of Alberta in 2003–2004 and the University of Waterloo in 2019–2020 and this book is written when he was at the University of Waterloo. Also, he was the author of 240 papers in peer-reviewed journals and many international conferences. Besides, he was the reviewer and member of the editorial board for a long-range of very prestigious journals, and currently, he is the chief editor of Computational Mathematics and Computer Modeling with Applications (CMCMA) journal. He successfully supervised 11 Ph.D. students as well as six postdoctoral fellows.

Snehashish Chakraverty

Department of Mathematics, National Institute of Technology Rourkela, Sundargarh, Odisha, India.

Snehashish Chakraverty has an experience of 29 years as a researcher and teacher. Presently, he is working in the Department of Mathematics (Applied Mathematics Group), National Institute of Technology Rourkela, Odisha, as a senior (HAG) professor. Before this, he was with CSIR-Central Building Research Institute, Roorkee, India. After completing graduation from St. Columba's College (Ranchi University), his career started from the University of Roorkee (Now, Indian Institute of Technology Roorkee) and did MSc (Mathematics) and MPhil (Computer Applications) from there securing the first position in the university. Dr. Chakraverty received his PhD from IIT Roorkee in 1992. Thereafter, he did his postdoctoral research at the Institute of Sound and Vibration Research (ISVR), University of Southampton, UK, and at the

Faculty of Engineering and Computer Science, Concordia University, Canada. He was also a visiting professor at Concordia and McGill Universities, Canada, during 1997–1999 and a visiting professor of University of Johannesburg, South Africa, during 2011–2014. He has authored/coauthored 16 books, published 333 research papers (till date) in journals and conferences, 2 more books are in press, and 2 books are ongoing. He is in the editorial boards of various International Journals, Book Series, and Conferences. Prof. Chakraverty is the chief editor of the “International Journal of Fuzzy Computation and Modelling” (IJFCM), Inderscience Publisher, Switzerland (<http://www.inderscience.com/ijfcm>), associate editor of “Computational Methods in Structural Engineering, Frontiers in Built Environment,” and happens to be the editorial board member of “Springer Nature Applied Sciences,” “IGI Research Insights Books,” “Springer Book Series of Modeling and Optimization in Science and Technologies,” “Coupled Systems Mechanics (Techno Press),” “Curved and Layered Structures (De Gruyter),” “Journal of Composites Science (MDPI),” “Engineering Research Express (IOP),” and “Applications and Applied Mathematics: An International Journal.” He is also the reviewer of around 50 national and international journals of repute, and he was the President of the Section of Mathematical Sciences (including Statistics) of “Indian Science Congress” (2015–2016) and was the Vice President – “Orissa Mathematical Society” (2011–2013). Prof. Chakraverty is a recipient of prestigious awards, viz. Indian National Science Academy (INSA) nomination under International Collaboration/Bilateral Exchange Program (with Czech Republic), Platinum Jubilee ISCA Lecture Award (2014), CSIR Young Scientist (1997), BOYSCAST (DST), UCOST Young Scientist (2007, 2008), Golden Jubilee Director’s (CBRI) Award (2001), INSA International Bilateral Exchange Award (2010–2011 [selected but could not undertake], 2015 [selected]), Roorkee University Gold Medals (1987, 1988) for first positions in MSc and MPhil (Computer Applications), etc. He has already guided 15 PhD students and 9 are ongoing. Prof. Chakraverty has undertaken around 16 research projects as the principle investigator funded by international and national agencies totaling about 1.5 crores. A good number of international and national conferences, workshops, and training programs have also been organized by him. His present research area includes differential equations (ordinary, partial, and fractional), Numerical Analysis and Computational Methods, Structural Dynamics (FGM, Nano) and Fluid Dynamics, Mathematical Modeling and Uncertainty Modeling, Soft Computing and Machine Intelligence (Artificial Neural Network, Fuzzy, Interval, and Affine Computations).

Contributors:

Hadi Veisi

Faculty of New Sciences and Technologies, Data and Signal Processing Lab, University of Tehran, Tehran, Iran.

Mohsen Razzaghi

Department of Mathematics and Statistics, Mississippi State University, Mississippi State, USA.

Abdon Atangana

Institute for Groundwater Studies, Faculty of Natural and Agricultural Sciences,
University of the Free State, Bloemfontein 9300, South Africa.

Dumitru Baleanu

Department of Mathematics, Cankaya University, Ankara, 06530, Turkey.

Sunil Kumar

Department of Mathematics, National Institute of Technology, Jamshedpur 831014,
Jharkhand, India.

Dara Rahmati

Computer Science and Engineering Department, Shahid Beheshti University, Tehran,
Iran.

Saeid Gorgin

Department of Electrical Engineering and Information Technology, Iranian Research
Organization for Science and Technology (IROST), Tehran, Iran.

Behzad Salami

Computer Science Department, Barcelona Supercomputing Center (BSC), Spain.

Maliheh Shaban Tameh

Department of Chemistry, University of Minnesota, Minneapolis, MN, 55455, USA.

Fatemeh Baharifard

School of Computer Science, Institute for Research in Fundamental Sciences (IPM),
Tehran, Iran.

Mostafa Jani

Department of Computer and Data Science, Faculty of Mathematical Sciences,
Shahid Beheshti University, Tehran, Iran.

Amir Hosein Hadian Rasanan

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences,
Shahid Beheshti University, Tehran, Iran.

Mohammad Mahdi Moayeri

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences,
Shahid Beheshti University, Tehran, Iran.

Mohammad Hemami

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences,
Shahid Beheshti University, Tehran, Iran.

Alireza Afzal Aghaei

Department of Computer and Data Science, Faculty of Mathematical Sciences,
Shahid Beheshti University, Tehran, Iran.

Simin Shekarpaz

Department of Applied Mathematics, Brown University, Providence, RI, 02912,
USA.

Alireza Rajabi

Department of Computer and Data Sciences, Faculty of Mathematical Sciences,
Shahid Beheshti University, Tehran, Iran.

Sherwin Nedaei Janbesaraei

School of Computer Sciences, Institute for Research in Fundamental Sciences (IPM),
Tehran, Iran.

Amirreza Azmoon

Department of Computer Science , The Institute for Advance Studies in Basic Sciences (IASBS), Zanjan, Iran.

Mohammad Akhavan

School of Computer Science, Institute for Research in Fundamental Sciences (IPM),
Tehran, Iran.

Mohsen Asghari

School of Computer Science, Institute for Research in Fundamental Sciences (IPM),
Tehran, Iran.

Reza Sahleh

Department of Computer and Data Sciences, Faculty of Mathematical Sciences,
Shahid Beheshti University, Tehran, Iran.

Arsham Gholamzadeh Khoee

Department of Computer Science, School of Mathematics, Statistics, and Computer
Science, University of Tehran, Tehran, Iran.

Armin Ahmadzadeh

School of Computer Science, Institute for Research in Fundamental Sciences (IPM),
Tehran, Iran.

Part I

Basics of Support Vector Machines

Chapter 1

Introduction to SVM

Hadi Veisi

Abstract In this chapter, a review of the machine learning (ML) and pattern recognition concepts is given, and basic ML techniques (supervised, unsupervised, and reinforcement learning) are described. Also, a brief history of ML development from the primary works before the 1950s (including Bayesian theory) up to the most recent approaches (including deep learning) is presented. Then, an introduction to the support vector machine (SVM) with a geometric interpretation is given, and its basic concepts and formulations are described. A history of SVM progress (from Vapnik's primary works in the 1960s up to now) is also reviewed. Finally, various ML applications of SVM in several fields such as medical, text classification, and image classification are presented.

1.1 What is Machine Learning?

Recognizing a person from his/her face, reading a handwritten letter, understanding a speech lecture, deciding to buy the stock of a company after analyzing the company's profile, and driving a car in a busy street are some examples of using human intelligence. Artificial Intelligence (AI) refers to the simulation of human intelligence in machines, i.e., computers. Machine Learning (ML) as a subset of AI is the science of automatically learning computers from experiences to do intelligent and human-like tasks. Similar to other actions in computer science and engineering, ML is realized by computer algorithms that can learn from their environment (i.e., data) and can generalize this training to act intelligently in new environments. Nowadays, computers can recognize people from their face using face recognition algorithms, converting a handwritten letter to its editable form using handwritten recognition, understand a speech lecture using speech recognition and natural language under-

Hadi Veisi

Faculty of New Sciences and Technologies, University of Tehran, Tehran, Iran, e-mail:
h.veisi@ut.ac.ir

standing, buy a stock of a company using algorithmic trading methods, and can drive a car automatically in self-driving cars. The term machine learning was coined by Arthur Samuel in 1959, an American pioneer in the field of computer gaming and artificial intelligence that defines this term as "it gives computers the ability to learn without being explicitly programmed" [1]. In 1997, Tom Mitchell, an American computer scientist and a former Chair of the Machine Learning Department at the Carnegie Mellon University (CMU) gave a mathematical and relational definition that "A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E" [2]. So, if you want your ML program to predict the growth of a stock (task T) to decide for buying it, you can run a machine learning algorithm with data about past price patterns of this stock (experience E, that is called training data) and, if it has successfully "learned", it will then do better at predicting future price (performance measure P). The primary works in ML return to the 1950s and this field has received several improvements during the last 70 years. There is a short history of ML:

- **Before the 1950s:** Several ML related theories have been developed including Bayesian theory [3], Markov chain [4], regression, and estimation theories [5]. Also, Donald Hebb in 1949 [6] presented his model of brain neuron interactions which is the basis of McCulloch-Pitts's neural networks [7].
- **The 1950s:** In this decade, ML pioneers have proposed the primary ideas and algorithms for machine learning. The Turing test, originally called the imitation game was proposed by Alan Turing as a test of a machine's ability to exhibit intelligent behavior equivalent to, or indistinguishable from, that of a human. Arthur Samuel of IBM developed a computer program for playing checkers [8]. Frank Rosenblatt extended Hebb's learning model of brain cell interaction with Arthur Samuel's ideas and created the perceptron [9].
- **The 1960s:** Bayesian methods are introduced for probabilistic inference [10]. The primary idea of Support Vector Machines (SVMs) is given by Vapnik and Lerner [11]. Widrow and Hoff developed the delta learning rules for neural networks which was the precursor of the backpropagation algorithm [11]. Sebestyen [13] and Nilsson [14] proposed the nearest neighbor idea. Donald Michie used reinforcement learning to play Tic-tac-toe [15]. The decision tree was introduced by Morgan and Sonquist [16].
- **The 1970s:** The quit years which is also known as AI Winter caused by pessimism about machine learning effectiveness due to the limitation of the ML methods in solving only linearly separable problems [17].
- **The 1980s:** The birth of brilliant ideas resulted in renewed enthusiasm. Backpropagation publicized by Rumelhart, Hinton, and Williams [18], causes a resurgence in machine learning. Hopfield popularizes his recurrent neural networks [19]. Watkins develops Q-learning in reinforcement learning [20]. Fukushima published his work on the Neocognitron neural network [21] which later inspires Convolutional Neural Networks (CNNs). Boltzmann machine [22] was proposed which was later used in Deep Belief Networks (DBNs).

- **The 1990s:** This is the decade for the birth of today's form of SVM by Vapnik and his colleagues in [23] and is extended in [24] and [25]. In these years, ML works shifts from knowledge-driven and rule-based approaches to the data-driven approach, and other learning algorithms such as Recurrent Neural Networks (RNNs) are introduced. Hochreiter and Schmidhuber invent long short-term memory (LSTM) recurrent neural networks [26] which became a practicality successful method for sequential data modeling. IBM's Deep Blue beats the world champion at chess, the grand-master Garry Kasparov [27]. Tin Kam Ho introduced random decision forests [28]. Boosting algorithms are proposed [29].
- **The 2000s:** Using ML methods in real applications, dataset creation, and organizing ML challenges become widespread. Support Vector Clustering and other Kernel methods were introduced [30]. A deep belief network was proposed by Hinton which is among the starting points for deep learning [31].
- **The 2010s:** Deep learning becomes popular and is the overcome most ML methods, results in becoming integral to many real-world applications [32]. Various deep neural networks such as autoencoders [33], Convolutional neural networks (CNNs) [34], Generative Adversarial Networks (GANs) [35] were introduced. ML achieved higher performance than human in various fields such as lipreading (e.g., LipNet [36]), playing Go (e.g., Google's AlphaGo and AlphaGo Zero programs [37]), information retrieval using natural language processing (e.g., IBM's Watson in Jeopardy competition [38]).

The highly complex nature of most real-world problems, often means that inventing specialized algorithms that will solve them perfectly every time is impractical, if not impossible. Examples of machine learning problems include, “Can we recognize the spoken words by only looking at the lip movements?”, “Is this cancer in this mammogram?”, “Which of these people are good friends with each other?”, “Will this person like this movie?”. Such problems are excellent targets for ML, and in fact, machine learning has been applied to such problems with great success, as mentioned in the history. Machine learning is also highly related to another similar topic, pattern recognition. These two terms can now be viewed as two facets of the same fields; however, machine learning grew out of computer science whereas pattern recognition has its origins in engineering [39]. Another similar topic is data mining which utilizes ML techniques in discovering patterns in large data and transforming raw data into information and decision. Within the field of data analytics, machine learning is used to devise complex models and algorithms that lend themselves to prediction which is known as predictive analytics in commercial applications. These analytical models allow researchers, data scientists, engineers, and analysts to “produce reliable, repeatable decisions and results” and uncover “hidden insights” through learning from historical relationships and trends in the data (i.e., input).

1.1.1 Classification of Machine Learning Techniques

Machine learning techniques are classified into three following categories, depending on the nature of the learning data or learning process:

1. **Supervised learning:** In this type of learning, there is a supervisor to teach machines in learning a concept. This means that the algorithm learns from labeled data (i.e., training data) which include example data and related target responses, i.e., input and output pairs. If we assume the ML algorithm as a system (e.g., a face identification), in the training phase of the system, we provide both input sample (e.g., an image from a face) and the corresponding output (e.g., the ID of the person to whom face belongs) to the system. The collecting of labeled data requires skilled human agents (e.g., a translator to translate a text from a language to another) or a physical experiment (e.g., determining whether there is rock or metal near to a sonar system of a submarine) that are costly and time-consuming. The supervised learning methods can be divided into classification and regression. When the number of classes of the data is limited (i.e., the output label of the data is a discrete variable) the learning is called classification (e.g., classifying an email to spam and not-spam classes); and when the output label is a continuous variable (e.g., the price of a stock index) the topic is called regression. Examples of the most widely used supervised learning algorithms are SVM [23, 24, 25], Artificial Neural Networks (e.g., Multi-layer perceptron [18], LSTM [26, 40]), linear and logistic regression [41], Naïve Bayes [42], decision trees [16] and K-Nearest Neighbor (KNN) [13, 14].
2. **Unsupervised learning:** In this case, there is not any supervision in the learning and the ML algorithm works on the unlabeled data. It means that the algorithm learns from plain examples without any associated response, leaving to the algorithm to determine the data patterns based on the similarities in the data. This type of algorithm tends to restructure the data and cluster them. From a system viewpoint, this kind of learning receives sample data as the input (e.g., the human faces) without the corresponding output and groups similar samples in the same clusters. The categorization of unlabeled data is commonly called clustering. Also, association as another type of unsupervised learning refers to methods which can discover rules that describe large portions of data, such as people that buy product X also tend to buy the other product Y. Dimensionality reduction as another type of unsupervised learning denotes the methods transform data from a high-dimensional space into a low-dimensional space. Examples of well-known unsupervised learning methods [43] are k-means, hierarchical clustering, density-based spatial clustering of applications with noise (DBSCAN), Neural Networks (e.g., Autoencoders, Self-organizing map), Expectation-Maximization (EM), and Principal Component Analysis (PCA).
3. **Reinforcement learning:** In this category of learning, an agent can learn from an action-reward mechanism by interacting with an environment just like the learning process of a human to play chess game by exercising and training by trial and error. Reinforcement algorithms are presented with examples and without labels

but receiving positive (e.g., reward) or negative (e.g., penalty) feedback from the environment. Two widely used reinforcement models are Markov Decision Process (MDP) and Q-learning [44].

A summary of machine learning types is summarized in Fig. 1.1. In addition to the mentioned three categories, there is another type called semi-supervised learning (that is also referred to as either transductive learning or inductive learning) which falls between supervised and unsupervised learning methods. It combines a small amount of labeled data with a large amount of unlabeled data for training.

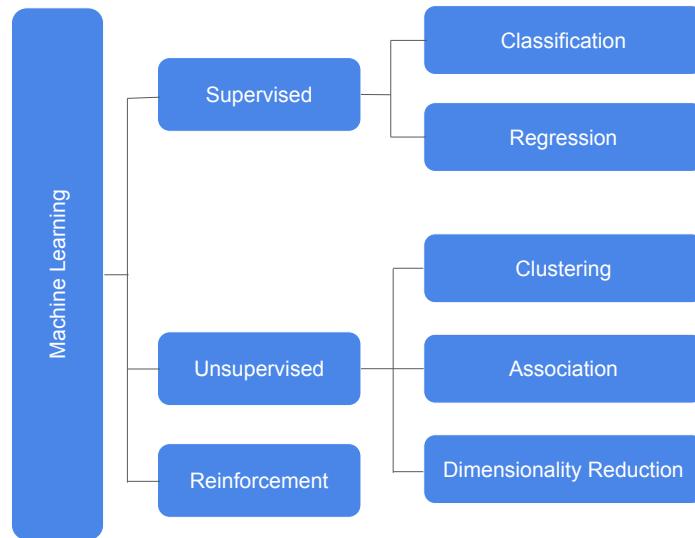


Fig. 1.1: Various types of machine learning

From another point of view, machine learning techniques are classified into the generative approach and discriminative approach. Generative models explicitly model the actual distribution of each class of data while discriminative models learn the (hard or soft) boundary between classes. From the statistical viewpoint, both of these approaches finally predict the conditional probability $P(\text{Class}|\text{Data})$ but both models learn different probabilities. In generative methods, joint distribution $P(\text{Class}, \text{Data})$ is learned and the prediction is performed according to this distribution. On the other side, discriminative models do predictions by estimating conditional probability $P(\text{Class}|\text{Data})$.

Examples of generative methods are deep generative models (DGMs) such as Variational Autoencoder (VAE) and GANs, Naïve Bayes, Markov random fields, and Hidden Markov Models (HMM). SVM is a discriminative method that learns the decision boundary like some other methods such as logistic regression, traditional neural networks such as multi-layer perceptron (MLP), KNN, and Conditional Random Fields (CRFs).

1.2 What is the Pattern?

In ML, we seek to design and build machines that can learn and recognize patterns, as is also called pattern recognition. To do this, the data, need to have regularity or arrangement, called pattern, to be learned by ML algorithms. The data may be created by humans such as stock price or a signature or have a natural nature such as speech signals or DNS. Therefore, a pattern includes elements that are repeated in a predictable manner. The patterns in natural data, e.g., speech signals, are often chaotic and stochastic and do not exactly repeat. There are various types of natural patterns include spirals, meanders, waves, foams, tilings, cracks, and those created by symmetries of rotation and reflection. Some types of patterns such as a geometric pattern in an image can be directly observed while abstract patterns in a huge amount of data or a language may become observable after analyzing the data using pattern discovery methods. In both cases, the underlying mathematical structure of a pattern can be analyzed by machine learning techniques which are mainly empowered by mathematical tools. The techniques can learn the patterns to predict or recognize them or can search them to find the regularities. Accordingly, if a dataset suffers from any regularities and repeatable templates, the modeling result of ML techniques will not be promising.

1.3 An Introduction to SVM with a Geometric Interpretation

Support vector machine (SVM), also known as support vector network, is a supervised learning approach used for classification and regression. Given a set of training labeled examples belonging to two classes, the SVM training algorithm builds a decision boundary between the samples of these classes. SVM does this in such a way that optimally discriminates between two classes by maximizing the margin between two data categories. For data samples in an N -dimensional space, SVM constructs an $N - 1$ dimensional separating hyperplane to discriminate two classes. To describe the SVM, assume a binary classification example in a 2-dimensional space, e.g., distinguishing a cat from a dog using the values of their height (x_1) and weight (x_2). An example of the training labeled samples for these classes in the feature space is given in Fig. 2(a), in which the samples for the dog are considered as the positive samples and the cat sample are presented as the negative ones. To do the classification, probably the primary intuitive approach is to draw a separative line between the positive and negative samples. However, as it is shown in Fig. 2(b), there are many possible lines to be the decision boundary between these classes. Now, the question is which line is better and should be chosen as the boundary?

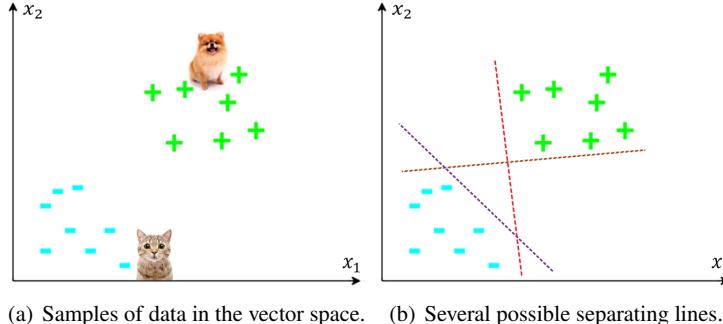


Fig. 1.2: (a) A binary classification example, (b) Some possible decision boundaries

Although all the lines given in Fig. 2(b) are a true answer to do the classification, neither of them seems the best fit. Alternatively, a line that is drawn between the two classes which have the maximum distance from both classes is the better choice. To do this, the data points that lie closest to the decision boundary are the most difficult samples to classify and they have a direct bearing on the optimum location of the boundary. These samples are called support vectors that are closer to the decision boundary and influence the position and orientation (see Fig. 1.3). According to these vectors, the maximum distance between the two classes is determined. This distance is called margin and a decision line that half this margin seems to be the optimum boundary. This line is such that the margin is maximized which is called the maximum-margin line between the two classes. SVM classifier attempts to find this optimal line.

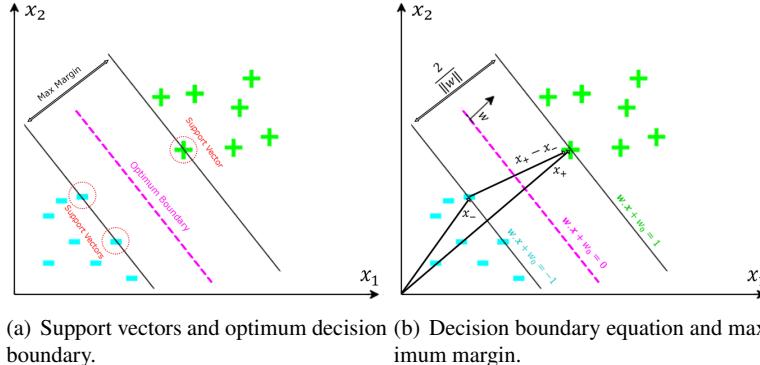


Fig. 1.3: (a) SVM optimum decision boundary, (b) Definition of the notations

In SVM, to construct the optimal decision boundary, a vector \mathbf{w} is considered to be perpendicular to the margin. Now, to classifying an unknown vector \mathbf{x} , we can

project it onto \mathbf{w} by computing $\mathbf{w} \cdot \mathbf{x}$ and determine on which side of the decision boundary \mathbf{x} lies by calculating $\mathbf{w} \cdot \mathbf{x} \geq t$ for a constant threshold t . It means that if the values of $\mathbf{w} \cdot \mathbf{x}$ are more than t , i.e., it is far away, sample \mathbf{x} is classified as a positive example. By assuming $t = -w_0$, the given decision rule can be given as $\mathbf{w} \cdot \mathbf{x} + w_0 \geq 0$. Now, the question is, how we can determine the values of \mathbf{w} and w_0 ? To do this, the following constraints are considered which means a sample is classified as positive if the value is equal or greater than 1, it is classified as negative if the value of -1 or less:

$$\mathbf{w} \cdot \mathbf{x}_+ + w_0 \geq 1, \quad \text{and} \quad \mathbf{w} \cdot \mathbf{x}_- + w_0 \leq -1. \quad (1.1)$$

These two equations can be integrated into an inequality, as in Eq. 1.2 by introducing a supplementary variable, y_i which is equal to +1 for positive samples and is equal to -1 for negative samples. This inequality is considered as equality, i.e., $y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1 = 0$, to define the main constraint of the problem which means the examples lying on the margins (i.e., *support vectors*) to be constrained to 0. This equation is equivalence to a line that is the answer to our problem. This decision boundary line in this example becomes a hyperplane in the general N -dimensional case.

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1 \geq 0. \quad (1.2)$$

To find the maximum margin that separating positive and negative examples, we need to know the width of the margin. To calculate the width of the margin, $(\mathbf{x}_+ - \mathbf{x}_-)$ need to be projected onto unit normalized $\frac{\mathbf{w}}{\|\mathbf{w}\|}$. Therefore, the width is computed as $(\mathbf{x}_+ - \mathbf{x}_-) \cdot \frac{\mathbf{w}}{\|\mathbf{w}\|}$ in which by using $y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1 = 0$ to substituting $\mathbf{w} \cdot \mathbf{x}_+ = 1 - w_0$ and $\mathbf{w} \cdot \mathbf{x}_- = 1 + w_0$ in that, the final value for width is obtained as $2\|\mathbf{w}\|$ (see Fig. 1.3). Finally, maximizing this margin is equivalent to Eq. 1.3 which is a quadratic function:

$$\min_{\mathbf{w}, w_0} \frac{1}{2} \|\mathbf{w}\|^2, \quad (1.3)$$

$$y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1 \geq 0.$$

This is a constrained optimization problem and can be solved by the Lagrange multiplier method. After writing the Lagrangian equation as in Eq. 1.4, and computing the partial derivative with respect to \mathbf{w} and setting it to zero results in $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$ and with respect to w_0 and setting it to zero gives $\sum_i \alpha_i y_i$. It means that \mathbf{w} is a linear combination of the samples. Using these values in Eq. 1.4 results in a Lagrangian equation in which the problem depends only on dot products of pairs of data samples. Also, $\alpha_i = 0$ for the training examples that are not support vectors that means these examples do not affect the decision boundary. Another interesting fact about this optimization problem is that it is a convex problem and it is guaranteed to always find a global optimum.

$$L(\mathbf{w}, w_0) = \frac{1}{2} \|\mathbf{w}\|^2 + \sum_i [y_i(\mathbf{w} \cdot \mathbf{x}_i + w_0) - 1] \alpha_i. \quad (1.4)$$

The above-described classification problem and its solution using the SVM assumes the data is linearly separable. However, in most real-life applications, this assumption is not correct and most problems are not classified simply using a linear boundary. The SVM decision boundary is originally linear [11] but has been extended to handle non-linear cases as well [23]. To do this, SVM proposes a method called kernel trick in which an input vector is transformed using a nonlinear function like $\phi(\cdot)$ into a higher-dimensional space. Then, in this new space, the maximum-margin linear boundary is found. It means that a non-linear problem is converted into a linearly-separable problem in the new higher-dimensional space without affecting the convexity of the problem. A simple example of this technique is given in Fig. 1.4 in which 1-dimensional data samples, x_i , are transformed into 2-dimensional space using $(x_i, x_i \times x_i)$ transform. In this case, the dot product of two samples, i.e., $x_i \cdot x_j$, in the optimization problem is replaced with $\phi(x_i) \cdot \phi(x_j)$. In practice, if we have a function like such that $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$, then we do not need to know the transformation $\phi(\cdot)$ and only function $K(\cdot)$, (which is called the *kernel function*) is required. Some common kernel functions are linear, polynomial, sigmoid, and radial basis functions (RBF).

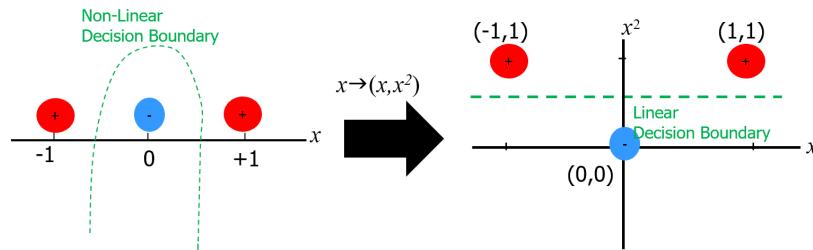


Fig. 1.4: Transforming data from a non-linear space into a linear higher-dimensional space

Although the kernel trick is a clever method to handle the non-linearity, the SVM still assumes that the data is linearly-separable in this transformed space. This assumption is not true in most real-world applications. Therefore, another type of SVM is proposed which is called *soft-margin* SVM [25]. The described SVM method up to now is known as *hard-margin* SVM. As given, hard-margin SVMs assume the data is linearly-separable without any errors, whereas soft-margin SVMs allow some misclassification and results in a more robust decision in non-linearly-separable data. Today, soft-margin SVMs are the most common SVM techniques in ML which

utilize so-called slack variables in the optimization problem to control the amount of misclassification.

1.4 History of SVMs

Vladimir Vapnik, the Russian statistician, is the main originator of the SVM technique. The primary work on the SVM algorithm was proposed by Vapnik and Lerner in 1963 [11] as the Generalized Portrait algorithm for pattern recognition. However, it was not the first algorithm for pattern recognition, and Fisher in 1936 had proposed a method for this purpose [45]. Also, Frank Rosenblatt had been proposed the perceptron linear classifier which was an early feedforward neural network [9]. One year after the primary work of Vapnik and Lerner, in 1964, Vapnik further developed the Generalized Portrait algorithm [46]. In this year, a geometrical interpretation of the kernels was introduced in [47] as inner products in a feature space. The kernel theory which is the main concept in the development of SVM and is called ‘kernel trick’ was previously proposed in [48]. In 1965, a large margin hyperplane in the input space was introduced in [49] which is another key idea of the SVM algorithm. At the same time, a similar optimization concept was used in pattern recognition by [50]. Another important research that defines the basic idea of the soft-margin concept in SVM was introduced by Smith in 1968 [51]. This idea was presented as the use of slack variables to overcome the problem of noisy samples that are not linearly separable. In the history of SVM development, the breakthrough work is the formulation of statistical learning framework or VC theory proposed by Vapnik and Chervonenkis in 1974 [53] which presents one of the most robust prediction methods. It is not surprising to say that the rising of SVM was in this decade and this reference has been translated from Russian into other languages such as German [54] and English [55]. The use of polynomial kernel in SVM was proposed by Poggio in 1975 [56] and the improvement of kernel techniques for regression was presented by Wahba in 1990 [57]. Studying the connection between neural networks and kernel regression was done by Poggio and Girosi in 1990 [58]. The improvement of the previous work on slack variables in [51] was done by Bennett and Mangasarian in 1992 [59]. Another main milestone in the development of SVM is in 1992 in which SVM has been presented in its today’s form by Vapnik and his colleagues Boser and Guyon [23]. In this work, the optimal margin classifier of linear classifiers (from Vapnik and Lerner in 1963 [11]) was extended to nonlinear cases by utilizing the kernel trick to maximum-margin hyperplanes [47]. In 1995, soft-margin of SVM classifiers to handle noisy and not linearly separable data was introduced using slack variables by Cortes and Vapnik [25]. In 1996, the algorithm was extended to the case of regression [60] which is called support-vector regression (SVR). The rapid growth of SVM and using this technique in various applications has been increased after 1995. Also, the theoretical aspects of SVM have been studied and it has been extended in other domains than the classification. The statistical bounds on the generalization of hard margin were given by Bartlett in 1998 [61] and it was presented

for soft margin and the regression case in 2000 by Shawe-Taylor and Cristianini [62]. The SVM was originally developed for supervised learning which has been extended to the unsupervised case in 2001 [30] called support-vector clustering. Another improvement of SVM was its extension from the binary classification into multiclass SVM by Duan and Keerthi in 2005 [63] by distinguishing between one of the labels and the rest (one-versus-all) or between every pair of classes (one-versus-one). In 2011, SVM was analyzed as a graphical model and it was shown that it admits a Bayesian interpretation using data augmentation technique [64]. Accordingly, a scalable version of the Bayesian SVM was developed in 2017 [65] enabling the application of Bayesian SVMs in big data applications. A summary of the related researches to SVM development is given in Table 1.1.

1.5 SVM Applications

Today, machine learning algorithms are on the rise and are widely used in real applications. Every year new techniques are proposed that overcome the current leading algorithms. Some of them are only little advances or combinations of existing algorithms and others are newly created and lead to astonishing progress. Although deep learning techniques are dominant in many real applications such as image processing (e.g., for image classification) and sequential data modeling (e.g., in natural language processing tasks such as machine translation), but these techniques require a huge amount of training data for success modeling. Large scale labeled data sets are not available in many applications in which other ML techniques (called classical ML methods) such as SVM, decision tree, and Bayesian family methods have higher performance than deep learning techniques. Furthermore, there is another fact in ML. Each task in ML applications can be solved using various methods, however, there is no single algorithm that will work well for all tasks. This fact is known as the No Free Lunch Theorem in ML [67]. Each task that we want to solve has its idiosyncrasies and there are various ML algorithms to suit the problem. Among the non-deep learning methods, SVM, as a well-known machine learning technique, is widely used in various classification and regression tasks today due to its high performance and reliability across a wide variety of problem domains and datasets [68]. Generally, SVM can be applied to any ML task in any application such as computer vision and image processing, natural language processing (NLP), medical applications, biometrics, and cognitive science. In the following, some common applications of SVM are reviewed.

- **ML Applications in Medical:** SVM is widely applied in medical applications including medical image processing (i.e., a cancer diagnosis in mammography [69]), bioinformatics (i.e., patients and gene classification) [70] and health signal analysis (i.e., electrocardiogram signal classification for detection for identifying heart anomalies [71], and electroencephalogram signal processing in psychology and neuroscience [72]). SVM which is among the successful methods in this field is used to diagnose and prognosis of various types of diseases. Also, SVM is

utilized for identifying the classification of genes, patients based on genes, and other biological problems [73].

- **Text Classification:** There are various applications for text classification including topic identification (i.e., categorization of a given document into predefined classes such as scientific, sport, or political), author identification/verification of a written document, spam, and fake news/email/comment detection, language identification (i.e., determining the language of a document), polarity detection (e.g., finding that a given comment in a social media or e-commerce website is positive or negative) and word sense disambiguation (i.e., determining the meaning of a disambiguated word in a sentence such as ‘bank’). SVM is a competitive method for the other ML techniques in this area [74, 75].
- **Image Classification:** The process of assigning a label to an image is generally known as image recognition which can be used in various fields such as in biometrics (e.g., face detection/identification/verification), medical (e.g., processing MRI and CT images for disease diagnosis), object recognition, remote-sensing classification (e.g., categorization of satellite images), automated image organization for social networks and websites, visual searches and image retrieval. Although deep learning techniques, especially CNNs lead this area for the representation and feature extraction, however, SVM is utilized commonly as the classifier [76, 77], both with classical image processing-based techniques and deep neural network methods [78]. The image recognition services are now generally offered by the AI teams of technology companies such as Microsoft, IBM, Amazon, and Google.
- **Steganography Detection:** Steganography is the practice of concealing a message in an appropriate multimedia carrier such as an image, an audio file, or a video file. This technique is used for secure communication in security-based organizations to concealing both the fact that a secret message is being sent and its contents. This method has the advantage over cryptography in which only the content of a message is protected. On the other hand, the act of detecting whether an image (or other files) is stego or not is called steganalysis. The analysis of an image to determine if it is a stego image or not is a binary classification problem in which SVM is widely used [79].

References

1. Samuel, A. L.: Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* **44**, 206–226 (2000)
2. Mitchell, T. M.: Machine learning, McGraw-Hill Higher Education, New York (1997)
3. Bayes, T.: An essay towards solving a problem in the doctrine of chances. *MD Comput.* **8**, 157 (1991)
4. Gagniuc, P. A.: Markov chains: from theory to implementation and experimentation. John Wiley & Sons, Hoboken NJ (2017)
5. Fisher, R. A.: The goodness of fit of regression formulae, and the distribution of regression coefficients. *J. R. Stat. Soc.* **85**, 597–612 (1922)
6. Hebb, D.: The organization of behavior. Wiley, New York (1949)

Table 1.1: A brief history of SVM development

| Decade | Year | Researcher(s) | SVM Development |
|--------|------|---|---|
| 1950 | 1950 | Aronszajn [48] | Introducing the ‘Theory of Reproducing Kernels’ |
| | 1963 | Vapnik and Lerner [11] | Introducing the Generalized Portrait algorithm (the algorithm implemented by support vector machines is a nonlinear generalization of the Generalized Portrait algorithm) |
| 1960 | 1964 | Vapnik [46] | Developing the Generalized Portrait algorithm |
| | 1964 | Aizerman, Braverman and Rozonoer [47] | Introducing the geometrical interpretation of the kernels as inner products in a feature space |
| | 1965 | Cover [49] | Discussing large margin hyperplanes in the input space and also sparseness |
| | 1965 | Mangasarian [50] | Studding optimization techniques for pattern recognition similar to large margin hyperplanes |
| | 1968 | Smith [51] | Introducing the use of slack variables to overcome the problem of noise and non-separability |
| 1970 | 1973 | Duda and Hart [52] | Discussing large margin hyperplanes in the input space |
| | 1974 | Vapnik and Chervonenkis [53] | Writing a book on ‘statistical learning theory’ (in Russian) which can be viewed as the starting of SVMs |
| | 1975 | Poggio [56] | Proposing the use of polynomial kernel in SVM |
| | 1979 | Vapnik and Chervonenkis [54] | Translating of Vapnik and Chervonenkis’s 1974 book to German |
| 1980 | 1982 | Vapnik [55] | Writing an English translation of his 1979 book |
| 1990 | 1990 | Poggio and Girosi [58] | Studying the connection between neural networks and kernel regression |
| | 1990 | Wahba [57] | Improving the kernel method for regressing |
| | 1992 | Bennett and Mangasarian [59] | Improving Smith’s 1968 work on slack variables |
| | 1992 | Boser, Guyon and Vapnik [23] | Presenting SVM in today’s form at the COLT 1992 conference |
| | 1995 | Cortes and Vapnik [25] | Introducing the soft margin classifier |
| | 1996 | Drucker et al. [60] | Extending the algorithm to the case of regression, called SVR |
| | 1997 | Muller et al. [66] | Extending SVM for time-series prediction |
| | 1998 | Bartlett [61] | Providing the statistical bounds on the generalization of hard margin |
| 2000 | 2000 | Shawe-Taylor and Cristianini [62] | Giving the statistical bounds on the generalization of soft margin and the regression case |
| | 2001 | Ben-Hur, Horn, Siegelmann and Vapnik [30] | Extending SVM to the unsupervised case |
| | 2005 | Duan and Keerthi [63] | Extending SVM from the binary classification into multiclass SVM |
| 2010 | 2011 | Polson and Scott [64] | Studding graphical model representation of SVM and its Bayesian interpretation |
| | 2017 | Florian, Galy-Fajou, Deutsch and Kloft [65] | Developing a scalable version of Bayesian SVM for big data applications |

7. McCulloch, W. S., Pitts, W.: A logical calculus of the ideas immanent in nervous activity. Bull. Math. Biol. **5**, 115–133 (1943)

8. Samuel, A. L.: Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* **3**, 210–229 (1959)
9. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. *Psychol. Rev.* **65**, 386–408 (1958)
10. Solomonoff, R. J.: A formal theory of inductive inference. Part II. *Inf. Control.* **7**, 224–254 (1964)
11. Vapnik, V.: Pattern recognition using generalized portrait method. *Autom. Remote. Control.* **24**, 774–780 (1963)
12. Widrow, B., Hoff, M. E.: Adaptive switching circuits (No. TR-1553-1). Stanford Univ Ca Stanford Electronics Labs (1960)
13. Sebestyen, G. S.: Decision-making processes in pattern recognition. Macmillan, New York (1962)
14. Nilsson, N. J.: Learning machines. McGraw-Hill, New York (1965)
15. Michie, D.: Experiments on the mechanization of game-learning Part I. Characterization of the model and its parameters. *Comput. J.* **6**, 232–236 (1963)
16. Morgan, J. N., Sonquist, J. A.: Problems in the analysis of survey data, and a proposal. *J. Am. Stat. Assoc.* **58**, 415–434 (1963)
17. Minsky, M., Papert, S. A.: Perceptrons: An introduction to computational geometry. MIT press, England (1969)
18. Rumelhart, D. E., Hinton, G. E., Williams, R. J.: Learning representations by back-propagating errors. *Nature* **323**, 533–536 (1986)
19. Hopfield, J. J.: Neurons with graded response have collective computational properties like those of two-state neurons. *Proc. Natl. Acad. Sci.* **81**, 3088–3092 (1984)
20. Watkins, C. J. C. H.: Learning from delayed rewards. PhD Thesis, University of Cambridge, England (1989)
21. Fukushima, K.: Neocognitron: A hierarchical neural network capable of visual pattern recognition. *Neural Netw.* **1**, 119–130 (1988)
22. Hinton, G. E.: Analyzing cooperative computation. 5th COGSCI, Rochester (1983)
23. Boser, B. E., Guyon, I. M., Vapnik, V. N.: A training algorithm for optimal margin classifiers. COLT92: 5th Annual Workshop Comput. Learn. Theory, Pennsylvania (1992)
24. Vapnik, V. N.: The nature of statistical learning theory. Springer-Verlag, New York (1995)
25. Cortes, C., Vapnik, V.: Support-vector networks. *Mach. Learn.* **20**, 273–297 (1995)
26. Hochreiter, S., Schmidhuber, J.: Long short-term memory. *Neural Comput.* **9**, 1735–1780 (1997)
27. Warwick K.: A brief history of Deep Blue, IBM's Chess Computer. Mental Floss (2017)
28. Ho, T. K.: Random decision forests. Proceedings of 3rd Int. Conf. Doc. Anal. Recognit., Montreal (1995)
29. Schapire, R. E.: The strength of weak learnability. *Mach. Learn.* **5**, 197–227 (1990)
30. Ben-Hur, A., Horn, D., Siegelmann, H. T., Vapnik, V.: Support vector clustering. *J. Mach. Learn. Res.* **2**, 125–137 (2001)
31. Hinton, G. E., Osindero, S., Teh, Y. W.: A fast learning algorithm for deep belief nets. *Neural Comput.* **18**, 1527–1554 (2006)
32. Goodfellow, I., Bengio, Y., Courville, A., Bengio, Y.: Deep learning. MIT Press, England (2016)
33. Liu, W., Wang, Z., Liu, X., Zeng, N., Liu, Y., Alsaadi, F. E.: A survey of deep neural network architectures and their applications. *Neurocomputing* **234**, 11–26 (2017)
34. Khan, A., Sohail, A., Zahoor, U., Qureshi, A. S.: A survey of the recent architectures of deep convolutional neural networks. *Artif. Intell. Rev.* **53**, 5455–5516 (2020)
35. Pan, Z., Yu, W., Yi, X., Khan, A., Yuan, F., Zheng, Y.: Recent progress on generative adversarial networks (GANs): A survey. *IEEE Access* **7**, 36322–36333 (2019)
36. Assael, Y. M., Shillingford, B., Whiteson, S., De Freitas, N.: Lipnet: End-to-end sentence-level lipreading. arXiv preprint arXiv:1611.01599 (2016)
37. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y.: Mastering the game of go without human knowledge. *Nature* **550**, 354–359 (2017)

38. Ferrucci, D., Levas, A., Bagchi, S., Gondek, D., Mueller, E. T.: Watson: beyond jeopardy!. *Artif. Intell.* **199**, 93–105 (2013)
39. Bishop, C. M.: Pattern recognition and machine learning. Springer, Singapore (2006)
40. Gers, F. A., Schmidhuber, J., Cummins, F.: Learning to forget: Continual prediction with LSTM. Ninth Int. Conf. Artif. Neural Netw., Edinburgh (1999)
41. Cramer, J. S.: The origins of logistic regression (Technical report). Tinbergen Institute, 167–178 (2002)
42. Hand, D. J., Yu, K.: Idiot's Bayes-not so stupid after all?. *Int. Stat. Rev.* **69**, 385–398 (2001)
43. Xu, R., Wunsch, D.: Survey of clustering algorithms. *IEEE Trans. Neural Netw. Learn. Syst.* **16**, 645–678 (2005)
44. Sutton, R. S., Barto, A. G.: Reinforcement learning: An introduction. MIT press, USA (2018)
45. Fisher, R. A.: The use of multiple measurements in taxonomic problems. *Ann. Eugen.* **7**, 179–188 (1936)
46. Vapnik V. N., Chervonenkis A. Y.: On a class of perceptrons. *Autom. Remote.* **25**, 103–109 (1964)
47. Aizerman, M.A., Braverman, E.M., Rozonoer, L.I.: Theoretical foundations of the potential function method in pattern recognition learning. *Autom. Remote.* **25**, 821–837 (1964)
48. Aronszajn, N.: Theory of reproducing kernels. *Trans. Am. Math. Soc.* **68**, 337–404 (1950)
49. Cover, T. M.: Geometrical and statistical properties of systems of linear inequalities with applications in pattern recognition. *IEEE Trans. Elect. Comput.* **3**, 326–334 (1965)
50. Mangasarian, O. L.: Linear and nonlinear separation of patterns by linear programming. *Oper. Res.* **13**, 444–452 (1965)
51. Smith, F. W.: Pattern classifier design by linear programming. *IEEE Trans. Comput.* **100**, 367–372 (1968)
52. Duda, R. O., Hart, P. E.: Pattern classification and scene analysis. Wiley, New York (1973)
53. Vapnik, V., Chervonenkis, A.: Theory of pattern recognition: Statistical problems of learning (Russian). Nauka, Moscow (1974)
54. Vapnik, V., Chervonenkis, A.: Theory of pattern recognition (German). Akademie-Verlag, Berlin (1979)
55. Vapnik, V. N.: Estimation of dependencies based on empirical data. Springer, New York (1982)
56. Poggio, T.: On optimal nonlinear associative recall. *Biol. Cybern.* **19**, 201–209 (1975)
57. Wahba, G.: Spline models for observational data. SIAM, Pennsylvania (1990)
58. Poggio, T., Girosi, F.: Networks for approximation and learning. *Proceedings of the IEEE* **78**, 1481–1497 (1990)
59. Bennett, K. P., Mangasarian, O. L.: Robust linear programming discrimination of two linearly inseparable sets. *Optim. Methods. Softw.* **1**, 23–34 (1992)
60. Drucker, H., Burges, C. J., Kaufman, L., Smola, A., Vapnik, V.: Support vector regression machines. *Adv. Neural Inf. Process Syst.* **9**, 155–161 (1996)
61. Bartlett, P. L.: The sample complexity of pattern classification with neural networks: the size of the weights is more important than the size of the network. *IEEE Trans. Inf. Theory* **44**, 525–536 (1998)
62. Shawe-Taylor, J., Cristianini, N.: Margin distribution and soft margin. In: Smola, A. J., Bartlett, P., Scholkopf, B., Schuurmans, D., (eds.) Advances in large margin classifiers, pp. 349–358. MIT Press, england (2000)
63. Duan, K. B., Keerthi, S. S.: Which is the best multiclass SVM method? An empirical study. *Int. workshop Multiple Classifier Syst.*, pp. 278–285. Springer, Heidelberg (2005)
64. Polson, N. G., Scott, S. L.: Data augmentation for support vector machines. *Bayesian Anal.* **6**, 1–23 (2011)
65. Wenzel, F., Galy-Fajou, T., Deutsch, M., Kloft, M.: Bayesian nonlinear support vector machines for big data. *European Conference on Mach. Learn. Knowl. Discov. Databases*, pp. 307–322. Springer, Cham (2017)
66. Müller, K. R., Smola, A. J., Rätsch, G., Schölkopf, B., Kohlmorgen, J., Vapnik, V.: Predicting time series with support vector machines. *Int. Conf. Artif. Neural Netw.*, pp. 999–1004. Springer, Heidelberg (1997)

67. Wolpert, D. H.: The lack of a priori distinctions between learning algorithms. *Neural Comput.* **8**, 1341–1390 (1996)
68. Cervantes, J., Garcia-Lamont, F., Rodríguez-Mazahua, L., Lopez, A.: A comprehensive survey on support vector machine classification: Applications, challenges and trends. *Neurocomputing* **408**, 189–215 (2020)
69. Azar, A. T., El-Said, S. A.: Performance analysis of support vector machines classifiers in breast cancer mammography recognition. *Neural. Comput. Appl.* **24**, 1163–1177 (2014)
70. Byvatov, E., Schneider, G.: Support vector machine applications in bioinformatics. *Appl. Bioinformatics* **2**, 67–77 (2003)
71. Melgani, F., Bazi, Y.: Classification of electrocardiogram signals with support vector machines and particle swarm optimization. *IEEE Trans. Inf. Technol. Biomed.* **12**, 667–677 (2008)
72. Li, S., Zhou, W., Yuan, Q., Geng, S., Cai, D.: Feature extraction and recognition of ictal EEG using EMD and SVM. *Comput. Biol. Med.* **43**, 807–816 (2013)
73. Pavlidis, P., Weston, J., Cai, J., Grundy, W. N.: Gene functional classification from heterogeneous data. *Proceedings of the fifth Annual Int. Conf. Comput. Biol.*, 249–255 (2001)
74. Joachims, T.: Transductive inference for text classification using support vector machines. *ICML 99: Proceedings of the Sixteenth Int. Conf. Mach. Learn.*, 200–209 (1999)
75. Aggarwal, C. C., Zhai, C.: A survey of text classification algorithms. In: Aggarwal, C. C., Zhai, C., (eds.) *Mining text data*, 163–222. Springer, Boston (2012)
76. Miranda, E., Aryuni, M., Irwansyah, E.: A survey of medical image classification techniques. *IEEE Int. Conf. Info. Mngmt. Tech.*, 56–61 (2016)
77. Tuia, D., Volpi, M., Copa, L., Kanevski, M., Munoz-Mari, J.: A survey of active learning algorithms for supervised remote sensing image classification. *IEEE J. Sel. Topics Signal Process.* **5**, 606–617 (2011)
78. Li, Y., Li, J., Pan, J. S.: Hyperspectral image recognition using SVM combined deep learning. *J. Internet Technol.* **20**, 851–859 (2019)
79. Li, B., He, J., Huang, J., Shi, Y. Q.: A survey on image steganography and steganalysis. *J. Inf. Hiding Multimedia Signal Process.* **2**, 142–172 (2011)

Chapter 2

Basics of SVM Method and Least Squares SVM

Kourosh Parand and Fatemeh Baharifard and Alireza Afzal Aghaei and Mostafa Jani

Abstract The learning process of support vector machine algorithms leads to solving a convex quadratic programming problem. Since this optimization problem has a unique solution and also satisfies the Karush-Kuhn-Tucker conditions, it can be solved very efficiently. In this chapter, the formulation of optimization problems which are arisen in the various form of support vector machine algorithms are discussed.

2.1 Linear SVM Classifiers

As mentioned in the previous chapter, the linear support vector machine method for categorizing separable data was introduced by Vapnik and Chervonenkis in 1964 [1]. This method finds the best discriminator hyperplane, which separates samples of two classes among a training data set. Consider $\mathcal{D} = \{(\mathbf{x}_i, y_i) | i = 1 \dots N, \mathbf{x}_i \in \mathbb{R}^d \text{ and } y_i \in \{-1, +1\}\}$ as a set of training data, where the samples in classes C_1 and C_2 have $+1$ and -1 labels, respectively.

In this section, the SVM method is explained for two cases. The first case occurs when the training samples are linearly separable and the goal is to find a linear

Kourosh Parand

Department of Statistics and Actuarial Science, University of Waterloo, Canada e-mail: k.parand1394@gmail.com

Fatemeh Baharifard

School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran
e-mail: fateme.baharifard@gmail.com

Alireza Afzal Aghaei

Department of Computer and Data Science, Faculty of Mathematical Sciences, Shahid Beheshti University, Tehran, Iran, e-mail: alirezaafzalaghaei@gmail.com

Mostafa Jani

Department of Computer and Data Science, Faculty of Mathematical Sciences, Shahid Beheshti University, Tehran, Iran e-mail: mostafa.jani@gmail.com

separator by the hard margin SVM method. The second is about the training samples which are not linearly separable (e.g., due to noise) and so have to use the soft margin SVM method.

2.1.1 Hard Margin SVM

Assume that the input data is linearly separable. Fig. 2.1 shows an example of this data as well as indicates that the separator hyperplane is not unique. The aim of the SVM method is to find a unique hyperplane that has a maximum distance to the closest points of both classes. The equation of this hyperplane can be considered as follows

$$\langle \mathbf{w}, \mathbf{x} \rangle + w_0 = \mathbf{w}^T \mathbf{x} + w_0 = \sum_{i=1}^d w_i x_i + w_0 = 0, \quad (2.1)$$

where w_i 's are unknown weights of the problem.

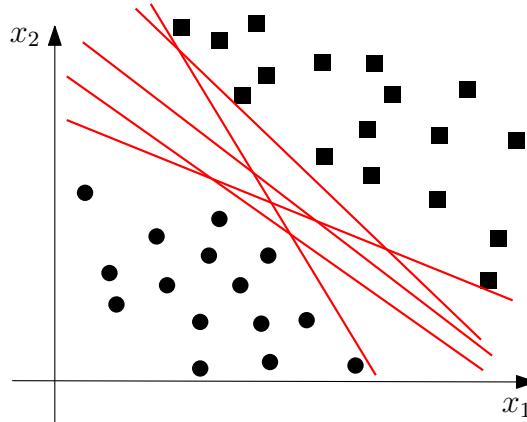


Fig. 2.1: Different separating hyperplanes for the same data set in a classification problem

The margin of a separator is the distance between the hyperplane and the closest training samples. The larger margin provides better generalization to unseen data. The hyperplane with the largest margin has equal distances to the closest samples of both classes. Consider the distance from the separating hyperplane to the nearest sample of each of classes C_1 and C_2 is equal to $\frac{D}{\|\mathbf{w}\|}$, where $\|\mathbf{w}\|$ is the Euclidean norm of \mathbf{w} . So, the margin will be equal to $\frac{2D}{\|\mathbf{w}\|}$. The purpose of the hard margin SVM method is to maximize this margin so that all training data is categorized correctly. Therefore, the following conditions must be satisfied

$$\begin{cases} \forall \mathbf{x}_i \in C_1 \ (y_i = +1) \rightarrow (\mathbf{w}^T \mathbf{x}_i + w_0) \geq D, \\ \forall \mathbf{x}_i \in C_2 \ (y_i = -1) \rightarrow (\mathbf{w}^T \mathbf{x}_i + w_0) \leq -D. \end{cases} \quad (2.2)$$

So, the following optimization problem is obtained to find the separator hyperplane, which is depicted in Fig. 2.2 for two-dimensional space

$$\begin{aligned} & \max_{D, \mathbf{w}, w_0} \frac{2D}{\|\mathbf{w}\|} \\ \text{s.t. } & (\mathbf{w}^T \mathbf{x}_i + w_0) \geq D, \quad \forall \mathbf{x}_i \in C_1, \\ & (\mathbf{w}^T \mathbf{x}_i + w_0) \leq -D, \quad \forall \mathbf{x}_i \in C_2. \end{aligned} \quad (2.3)$$

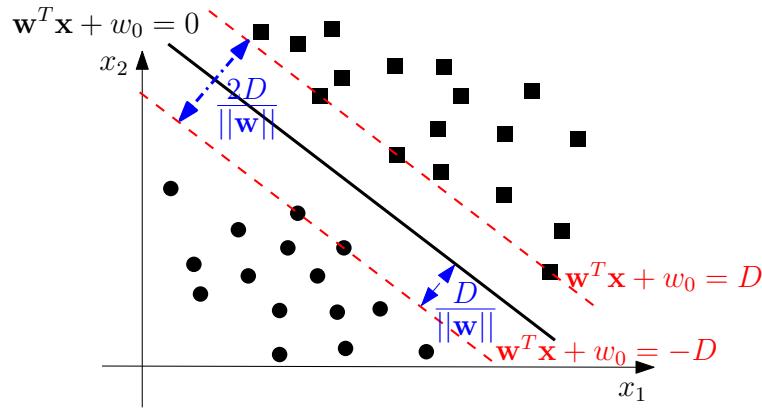


Fig. 2.2: Hard margin SVM method: a unique hyperplane with the maximum margin

One can set $\mathbf{w}' = \frac{\mathbf{w}}{D}$ and $w'_0 = \frac{w_0}{D}$ and combine the above two inequalities as an inequality $y_i(\mathbf{w}'^T \mathbf{x}_i + w'_0) \geq 1$ to have

$$\begin{aligned} & \max_{\mathbf{w}', w'_0} \frac{2}{\|\mathbf{w}'\|} \\ \text{s.t. } & y_i(\mathbf{w}'^T \mathbf{x}_i + w'_0) \geq 1, \quad i = 1, \dots, N. \end{aligned} \quad (2.4)$$

Moreover, in order to maximize the margin, can equivalently minimize $\|\mathbf{w}'\|$. This gives us the following primal problem for normalized vectors \mathbf{w} and w_0 (here $\|\mathbf{w}\|$ is substituted with $\|\mathbf{w}\|^2 = \mathbf{w}^T \mathbf{w}$ for simplicity)

$$\begin{aligned} & \min_{\mathbf{w}, w_0} \frac{1}{2} \mathbf{w}^T \mathbf{w} \\ \text{s.t. } & y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1, \quad i = 1, \dots, N. \end{aligned} \quad (2.5)$$

The Quadratic programming (QP) problem as a special case of nonlinear programming, is a problem of minimizing or maximizing a quadratic objective function of several variables, subject to some linear constraints defined on these variables. The advantage of QP problems is that there are some efficient computational methods to solve them [2, 3]. The general format of this problem is shown below. Here, Q is a real symmetric matrix.

$$\begin{aligned} \min_{\mathbf{x}} & \frac{1}{2} \mathbf{x}^T Q \mathbf{x} + \mathbf{c}^T \mathbf{x} \\ \text{s.t. } & \mathbf{A} \mathbf{x} \leq \mathbf{b}, \\ & E \mathbf{x} = \mathbf{d}. \end{aligned} \quad (2.6)$$

According to the above definition, the problem Eq. 2.5 can be considered as a convex QP problem (Q be an identity matrix, vector \mathbf{c} equals to zero and constraints are reformulated to become $A\mathbf{x} \leq \mathbf{b}$ form). If the problem has a feasible solution, it is a global minimum and can be obtained by an efficient method.

On the other hand, instead of solving the primal form of the problem, the dual form of it can be solved. The dual problem is often easier and helps us to have better intuition about the optimal hyperplane. More importantly, it enables us to take advantage of the kernel trick which will be explained later.

A common method for solving constrained optimization problems is using the technique of Lagrangian multipliers. In the Lagrangian multipliers method, a new function, namely the Lagrangian function is formed from the objective function and constraints, and the goal is to obtain the stationary point of this function. Consider the minimization problem as follows

$$\begin{aligned} p^* = \min_{\mathbf{x}} & f(\mathbf{x}) \\ \text{s.t. } & g_i(\mathbf{x}) \leq 0, \quad i = 1, \dots, m, \\ & h_i(\mathbf{x}) = 0, \quad i = 1, \dots, p. \end{aligned} \quad (2.7)$$

The Lagrangian function of this problem is

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\lambda}) = f(\mathbf{x}) + \sum_{i=1}^m \alpha_i g_i(\mathbf{x}) + \sum_{i=1}^p \lambda_i h_i(\mathbf{x}), \quad (2.8)$$

where $\boldsymbol{\alpha} = [\alpha_1, \dots, \alpha_m]$ and $\boldsymbol{\lambda} = [\lambda_1, \dots, \lambda_p]$ are Lagrangian multipliers vectors. According to the following equation, by setting $\alpha_i \geq 0$, the maximum of $\mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\lambda})$ is equivalent to $f(\mathbf{x})$

$$\max_{\alpha_i \geq 0, \lambda_i} \mathcal{L}(\mathbf{x}, \boldsymbol{\alpha}, \boldsymbol{\lambda}) = \begin{cases} \infty & \forall g_i(\mathbf{x}) > 0, \\ \infty & \forall h_i(\mathbf{x}) \neq 0, \\ f(\mathbf{x}) & \text{otherwise.} \end{cases} \quad (2.9)$$

Therefore, the problem Eq. 2.7 can be written as

$$p^* = \min_x \max_{\alpha_i \geq 0, \lambda_i} \mathcal{L}(x, \alpha, \lambda). \quad (2.10)$$

The dual form of Eq. 2.10, will be obtained by swapping the order of max and min

$$d^* = \max_{\alpha_i \geq 0, \lambda_i} \min_x \mathcal{L}(x, \alpha, \lambda). \quad (2.11)$$

The weak duality is always held and so $d^* \leq p^*$. In addition, because the primal problem is convex, the strong duality is also established and $d^* = p^*$. So, the primal optimal objective and the dual optimal objective are equal and instead of solving the primal problem, the dual problem can be solved.

Here, the dual formulation of the problem Eq. 2.5 is discussed. Combining the objective function and the constraints, give us

$$\mathcal{L}(\mathbf{w}, w_0, \alpha) = \frac{1}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^N \alpha_i (1 - y_i (\mathbf{w}^T \mathbf{x}_i + w_0)), \quad (2.12)$$

which leads to the following optimization problem

$$\min_{\mathbf{w}, w_0} \max_{\alpha_i \geq 0} \left\{ \frac{1}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^N \alpha_i (1 - y_i (\mathbf{w}^T \mathbf{x}_i + w_0)) \right\}. \quad (2.13)$$

Corresponding to the strong duality, the following dual optimization problem can be examined to find the optimal solution of the above problem

$$\max_{\alpha_i \geq 0} \min_{\mathbf{w}, w_0} \left\{ \frac{1}{2} \|\mathbf{w}\|^2 + \sum_{i=1}^N \alpha_i (1 - y_i (\mathbf{w}^T \mathbf{x}_i + w_0)) \right\}. \quad (2.14)$$

The solution is characterized by the saddle point of the problem. To find

$$\min_{\mathbf{w}, w_0} \mathcal{L}(\mathbf{w}, w_0, \alpha),$$

taking the first-order partial derivative of \mathcal{L} with respect to \mathbf{w} and w_0 obtain

$$\begin{cases} \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, w_0, \alpha) = 0, & \rightarrow \mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i, \\ \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \alpha)}{\partial w_0} = 0, & \rightarrow \sum_{i=1}^N \alpha_i y_i = 0. \end{cases} \quad (2.15)$$

In these equations, w_0 has been removed and a global constraint has been set for α . By substituting \mathbf{w} from the above equation in the Lagrangian function of Eq. 2.12, have

$$\mathcal{L}(\alpha) = \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j. \quad (2.16)$$

So, the dual form of problem Eq. 2.5 is as follows

$$\begin{aligned} \max_{\alpha} & \left\{ \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \right\} \\ \text{s.t. } & \sum_{i=1}^N \alpha_i y_i = 0, \\ & \alpha_i \geq 0, \quad i = 1, \dots, N. \end{aligned} \quad (2.17)$$

The equivalent of the problem Eq. 2.17 can be considered as follows, which is a quadratic programming due to Eq. 2.6

$$\begin{aligned} \min_{\alpha} & \frac{1}{2} \alpha^T \begin{bmatrix} y_1 y_1 \mathbf{x}_1^T \mathbf{x}_1 & \dots & y_1 y_N \mathbf{x}_1^T \mathbf{x}_N \\ \vdots & \ddots & \vdots \\ y_N y_1 \mathbf{x}_N^T \mathbf{x}_1 & \dots & y_N y_N \mathbf{x}_N^T \mathbf{x}_N \end{bmatrix} \alpha + (-\mathbf{1})^T \alpha \\ \text{s.t. } & -\alpha \leq \mathbf{0}, \\ & \mathbf{y}^T \alpha = \mathbf{0}. \end{aligned} \quad (2.18)$$

After finding α by QP process, vector of \mathbf{w} can be calculated from equality $\mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i$ and the only unknown variable that remains is w_0 . For computing w_0 , support vectors must be introduced first. To express the concept of support vectors, need to explain the Karush-Kuhn-Tucker (KKT) conditions [4, 5] beforehand.

The necessary conditions for the optimal solution of nonlinear programming are called the Karush Kuhn Tucker (KKT) conditions. In fact, if there exists some saddle point $(\mathbf{w}^*, w_0^*, \alpha^*)$ of $\mathcal{L}(\mathbf{w}, w_0, \alpha)$, then it satisfies the following KKT conditions

$$\begin{cases} 1. \quad y_i (\mathbf{w}^{*T} \mathbf{x}_i + w_0^*) \geq 1, \quad i = 1, \dots, N, \\ 2. \quad \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, w_0, \alpha)|_{\mathbf{w}^*, w_0^*, \alpha^*} = 0, \\ \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \alpha)}{\partial w_0}|_{\mathbf{w}^*, w_0^*, \alpha^*} = 0, \\ 3. \quad \alpha_i^* (1 - y_i (\mathbf{w}^{*T} \mathbf{x}_i + w_0^*)) = 0, \quad i = 1, \dots, N, \\ 4. \quad \alpha_i^* \geq 0, \quad i = 1, \dots, N. \end{cases} \quad (2.19)$$

The first condition indicates the feasibility of the solution and states that the conditions at the optimal point should not be violated. The second condition ensures that there is no direction that can both improve the objective function and be feasible. The third condition is related to the complementary slackness, which together with the fourth condition indicates that the Lagrangian multipliers of the inequality constraints are zero and the Lagrangian multipliers of the equality constraints are positive. In other words, for active constrain $y_i (\mathbf{w}^{*T} \mathbf{x}_i + w_0) = 1$, α_i can be greater than zero and the corresponding \mathbf{x}_i is defined as a support vector. But for inactive constraints $y_i (\mathbf{w}^{*T} \mathbf{x}_i + w_0) > 1$, $\alpha_i = 0$ and \mathbf{x}_i is not a support vector (see Fig. 2.3).

If define a set of support vectors as $SV = \{\mathbf{x}_i | \alpha_i > 0\}$ and consequently define $S = \{i | \mathbf{x}_i \in SV\}$, the direction of the hyper-plane which is related to \mathbf{w} can be found as follows

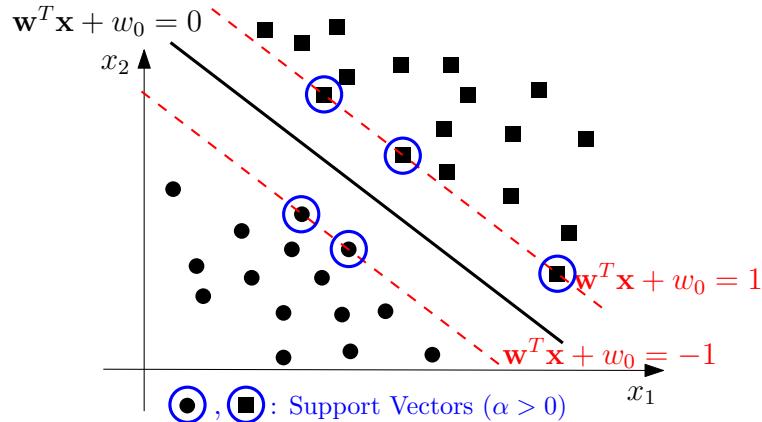


Fig. 2.3: The support vectors in the hard margin SVM method

$$\mathbf{w} = \sum_{s \in S} \alpha_s y_s \mathbf{x}_s. \quad (2.20)$$

Moreover, any sample whose Lagrangian multiplier is greater than zero is on the margin and can be used to compute w_0 . Using the sample \mathbf{x}_{s_j} that $s_j \in S$ for which the equality $y_{s_j}(\mathbf{w}^T \mathbf{x}_{s_j} + w_0) = 1$ is established, have

$$w_0 = y_{s_j} - \mathbf{w}^T \mathbf{x}_{s_j}. \quad (2.21)$$

By assuming that the linear classifier is $y = \text{sign}(w_0 + \mathbf{w}^T \mathbf{x})$, new samples can be classified using only support vectors of the problem. Consider a new sample \mathbf{x} , thus the label of this sample (\hat{y}) can be computed as follows

$$\begin{aligned} \hat{y} &= \text{sign}(w_0 + \mathbf{w}^T \mathbf{x}) \\ &= \text{sign}\left(y_{s_j} - \left(\sum_{s \in S} \alpha_s y_s \mathbf{x}_s\right)^T \mathbf{x}_{s_j} + \left(\sum_{s \in S} \alpha_s y_s \mathbf{x}_s\right)^T \mathbf{x}\right) \\ &= \text{sign}\left(y_{s_j} - \sum_{s \in S} \alpha_s y_s \mathbf{x}_s^T \mathbf{x}_{s_j} + \sum_{s \in S} \alpha_s y_s \mathbf{x}_s^T \mathbf{x}\right). \end{aligned} \quad (2.22)$$

2.1.2 Soft Margin SVM

Soft margin SVM is a method for obtaining a linear classifier for some training samples which are not actually linearly separable. The overlapping classes or separable classes that include some noise data are examples of these problems. In these cases, the hard margin SVM does not work well or even does not get the right answer. One solution for these problems is trying to minimize the number of misclassified

points as well as maximize the number of samples that are categorized correctly. But this counting solution falls into the category of NP-complete problems [6]. So, an efficient solution is defining a continuous problem that is solvable deterministically in a polynomial time by some optimization techniques. The extension of the hard margin method, called the soft margin SVM method was introduced by Cortes and Vapnik [6] for this purpose.

In the soft margin method, the samples are allowed to violate the conditions, while the total amount of these violations should not be increased. In fact, the soft margin method tries to maximize the margin while minimizes the total violations. Therefore, for each sample \mathbf{x}_i , a slack variable $\xi_i \geq 0$, which indicates the amount of violation of it from the correct margin, should be defined and the inequality of this sample should be relaxed to

$$y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1 - \xi_i. \quad (2.23)$$

Moreover, the total violation is $\sum_{i=1}^N \xi_i$, which should be minimized. So, the primal optimization problem to obtain the separator hyperplane becomes

$$\begin{aligned} & \min_{\mathbf{w}, w_0, \{\xi_i\}_{i=1}^N} \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^N \xi_i \\ & \text{s.t. } y_i(\mathbf{w}^T \mathbf{x}_i + w_0) \geq 1 - \xi_i, \quad i = 1, \dots, N, \\ & \quad \xi_i \geq 0, \end{aligned} \quad (2.24)$$

where C is a regularization parameter. A large amount for C leads to ignoring conditions hardly so if $C \rightarrow \infty$, this method is equivalent to hard margin SVM. Conversely, if the amount of C is small, conditions are easily relaxed and a large margin will be achieved. Therefore, the appropriate value of C should be used in proportion to the problem and the purpose.

If sample \mathbf{x}_i is categorized correctly, but located inside the margin, then $0 < \xi_i < 1$. Otherwise $\xi_i > 1$, which is because sample \mathbf{x}_i is in the wrong category and is a misclassified point (see Fig. 2.4).

As can be seen in Eq. 2.24, this problem is still a convex QP and unlike the hard margin method, it has always obtained a feasible solution.

Here, the dual formulation of the soft margin SVM is discussed. The Lagrange formulation is

$$\mathcal{L}(\mathbf{w}, w_0, \xi, \alpha, \beta) = \frac{1}{2} \|\mathbf{w}\|^2 + C \sum_{i=1}^N \xi_i + \sum_{i=1}^N \alpha_i (1 - \xi_i - y_i(\mathbf{w}^T \mathbf{x}_i + w_0)) - \sum_{i=1}^N \beta_i \xi_i, \quad (2.25)$$

where α_i 's and β_i 's are the Lagrangian multipliers. The Lagrange formulation should be minimized with respect to \mathbf{w} , w_0 , and ξ_i 's while be maximized with respect to positive Lagrangian multipliers α_i 's and β_i 's. To find $\min_{\mathbf{w}, w_0, \xi} \mathcal{L}(\mathbf{w}, w_0, \xi, \alpha, \beta)$, we have

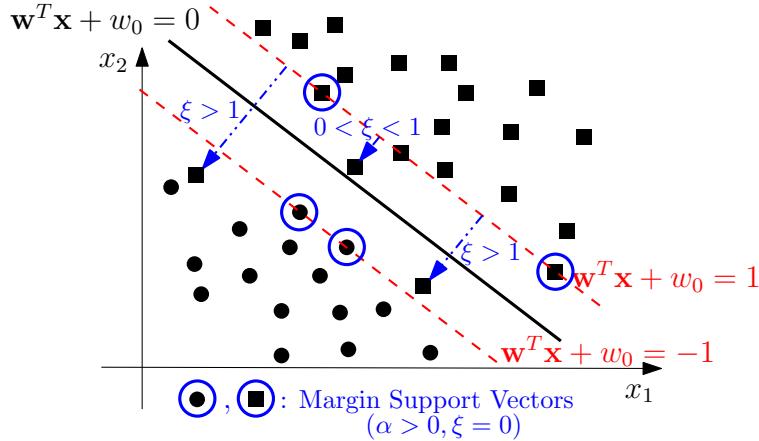


Fig. 2.4: Hard margin SVM method: types of support vectors and their corresponding ξ values

$$\begin{cases} \nabla_{\mathbf{w}} \mathcal{L}(\mathbf{w}, w_0, \xi, \alpha, \beta) = 0 & \rightarrow \mathbf{w} = \sum_{i=1}^N \alpha_i y_i \mathbf{x}_i, \\ \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \alpha, \beta)}{\partial w_0} = 0 & \rightarrow \sum_{i=1}^N \alpha_i y_i = 0, \\ \frac{\partial \mathcal{L}(\mathbf{w}, w_0, \xi, \alpha, \beta)}{\partial \xi_i} = 0 & \rightarrow C - \alpha_i - \beta_i = 0. \end{cases} \quad (2.26)$$

By substituting \mathbf{w} from the above equation in the Lagrangian function of Eq. 2.25, the same equation as Eq. 2.16 is achieved. But, here two constraints on α are created. One is the same as before and the other is $0 \leq \alpha_i \leq C$. It should be noted that β_i does not appear in $\mathcal{L}(\alpha)$ of Eq. 2.16 and just need to consider that $\beta_i \geq 0$. Therefore, the condition $C - \alpha_i - \beta_i = 0$ can be replaced by condition $0 \leq \alpha_i \leq C$. So, the dual form of problem Eq. 2.24 becomes as follows

$$\begin{aligned} \max_{\alpha} & \left\{ \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \mathbf{x}_i^T \mathbf{x}_j \right\} \\ \text{s.t.} & \sum_{i=1}^N \alpha_i y_i = 0, \\ & 0 \leq \alpha_i \leq C, \quad i = 1, \dots, N. \end{aligned} \quad (2.27)$$

After solving the above QP problem, \mathbf{w} and w_0 can be obtained based on Eqs. (2.20) and (2.21), respectively, while still defined $S = \{i | \mathbf{x}_i \in SV\}$. As mentioned before, the set of support vectors (SV) is obtained based on the complementary slackness criterion of $\alpha_i^* (1 - y_i (\mathbf{w}^* \mathbf{x}_i + w_0^*)) = 0$ for sample \mathbf{x}_i . Another complementary slackness condition of Eq. 2.24, is $\beta_i^* \xi_i = 0$. According to it, the support vectors are divided into two categories, the margin support vectors and the non-margin support vectors:

- If $0 < \alpha_i < C$, based on condition $C - \alpha_i - \beta_i = 0$, we have $\beta_i \neq 0$ and so ξ_i should be zero. Therefore, $y_i(\mathbf{w}^T \mathbf{x}_i + w_0) = 1$ is hold and sample \mathbf{x}_i located on the margin and be a margin support vector.
- If $\alpha_i = C$, we have $\beta_i = 0$ and so ξ_i can be greater than zero. So, $y_i(\mathbf{w}^T \mathbf{x}_i + w_0) < 1$ and sample \mathbf{x}_i is on or over the margin and be a non-margin support vector.

Also, the classification formula of a new sample in the soft margin SVM is the same as hard margin SVM which is discussed in Eq. 2.22.

2.2 Nonlinear SVM Classifiers

Some problems have non-linear decision surfaces and therefore linear methods do not provide a suitable answer for them. The nonlinear SVM classifier was introduced by Vapnik [7] for these problems. In this method, the input data is mapped to a new feature space by a nonlinear function and a hyperplane is found in the transformed feature space. By applying reverse mapping, the hyperplane becomes a curve or a surface in the input data space (see Fig. 2.5).

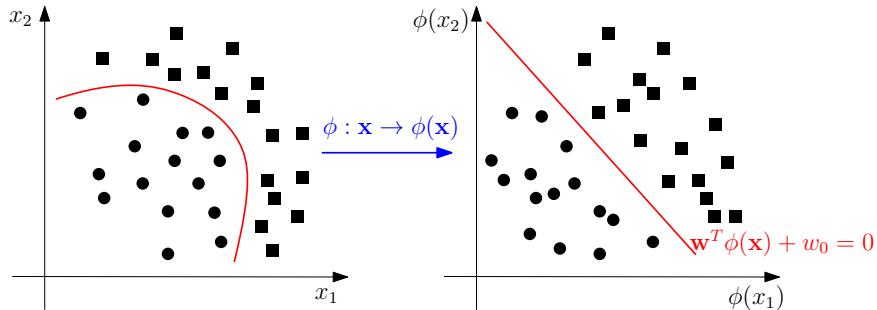


Fig. 2.5: Mapping input data to a high dimensional feature space to have a linear separator hyperplane in the transformed space

Applying a transformation $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^k$ on the input space, sample $\mathbf{x} = [x_1, \dots, x_d]$ can be represented by $\phi(\mathbf{x}) = [\phi_1(\mathbf{x}), \dots, \phi_k(\mathbf{x})]$ in the transformed space, where $\phi_i(\mathbf{x}) : \mathbb{R}^d \rightarrow \mathbb{R}$. The primal problem of soft-margin SVM in the transformed space is as follows

$$\begin{aligned} \min_{\mathbf{w}, w_0, \{\xi_i\}_{i=1}^N} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^N \xi_i \\ \text{s.t.} \quad & y_i(\mathbf{w}^T \phi(\mathbf{x}_i) + w_0) \geq 1 - \xi_i, \quad i = 1, \dots, N, \\ & \xi_i \geq 0, \end{aligned} \tag{2.28}$$

and $\mathbf{w} \in \mathbb{R}^k$ are the separator parameters that should be obtained.

The dual problem Eq. 2.27 also changes in a transformed space, as follows

$$\begin{aligned} \max_{\alpha} & \left\{ \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \right\} \\ \text{s.t. } & \sum_{i=1}^N \alpha_i y_i = 0, \\ & 0 \leq \alpha_i \leq C, \quad i = 1, \dots, N. \end{aligned} \quad (2.29)$$

In this case, the classifier is defined as

$$\hat{y} = \text{sign}(w_0 + \mathbf{w}^T \phi(\mathbf{x})), \quad (2.30)$$

where $\mathbf{w} = \sum_{\alpha_i > 0} \alpha_i y_i \phi(\mathbf{x}_i)$ and $w_0 = y_{s_j} - \mathbf{w}^T \phi(\mathbf{x}_{s_j})$ such that \mathbf{x}_{s_j} is a sample from the support vectors set.

The challenge of this method is choosing a suitable transformation. If the proper mapping is used, the problem can be modeled with good accuracy in the transformed space. But, if $k \gg d$, then there are more parameters to learn and this takes more computational time. In this case, it is better to use the kernel trick which is mentioned in the next section. The kernel trick is a technique to use feature mapping without explicitly applying it to the input data.

2.2.1 Kernel Trick and Mercer Condition

In the kernel-based approach, a linear separator in a high dimensional space is obtained without applying the transformation function to the input data. In the dual form of the optimization problem Eq. 2.29 only the dot product of each pair of training samples exists. By computing the value of these inner products, the calculation of $\alpha = [\alpha_1, \dots, \alpha_N]$ only remains. So, unlike the primal problem Eq. 2.28, learning k parameters is not necessary here. The importance of this is well seen when $k \gg d$.

Let define $K(\mathbf{x}, \mathbf{t}) = \phi(\mathbf{x})^T \phi(\mathbf{t})$ as the kernel function. The basis of the kernel trick is the calculation of $K(\mathbf{x}, \mathbf{t})$ without mapping samples \mathbf{x} and \mathbf{t} to the transformed space. For example, consider a two-dimensional input data set and define a kernel function as follows:

$$\begin{aligned} K(\mathbf{x}, \mathbf{t}) &= (1 + \mathbf{x}^T \mathbf{t})^2 \\ &= (1 + x_1 t_1 + x_2 t_2)^2 \\ &= 1 + 2x_1 t_1 + 2x_2 t_2 + x_1^2 t_1^2 + x_2^2 t_2^2 + 2x_1 t_1 x_2 t_2. \end{aligned} \quad (2.31)$$

As can be seen, the expansion of this kernel function equals to the inner product of the following second order ϕ 's

$$\begin{aligned}\phi(\mathbf{x}) &= [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2], \\ \phi(\mathbf{t}) &= [1, \sqrt{2}t_1, \sqrt{2}t_2, t_1^2, t_2^2, \sqrt{2}t_1t_2].\end{aligned}\quad (2.32)$$

So, we can substitute the dot product $\phi(\mathbf{x})^T\phi(\mathbf{t})$ with the kernel function $K(\mathbf{x}, \mathbf{t}) = (1 + \mathbf{x}^T\mathbf{t})^2$ without directly calculating $\phi(\mathbf{x})$ and $\phi(\mathbf{t})$.

The polynomial kernel function can similarly be generalized to d -dimensional feature space $\mathbf{x} = [x_1, \dots, x_d]$ where ϕ 's are polynomials of order M as follows

$$\begin{aligned}\phi(\mathbf{x}) &= [1, \sqrt{2}x_1, \dots, \sqrt{2}x_d, x_1^2, \dots, x_d^2, \sqrt{2}x_1x_2, \dots, \\ &\quad \sqrt{2}x_1x_d, \sqrt{2}x_2x_3, \dots, \sqrt{2}x_{d-1}x_d]^T.\end{aligned}$$

The following polynomial kernel function can indeed be efficiently computed with a cost proportional to d (the dimension of \mathbf{x}) instead of k (the dimension of $\phi(\mathbf{x})$)

$$K(\mathbf{x}, \mathbf{t}) = (1 + \mathbf{x}^T\mathbf{t})^M = (1 + x_1t_1 + x_2t_2 + \dots, x_dt_d)^M. \quad (2.33)$$

In many cases, the inner product in the embedding space can be computed efficiently by defining a kernel function. Some common kernel functions are listed below

- $K(\mathbf{x}, \mathbf{t}) = \mathbf{x}^T\mathbf{t}$,
- $K(\mathbf{x}, \mathbf{t}) = (\mathbf{x}^T\mathbf{t} + 1)^M$,
- $K(\mathbf{x}, \mathbf{t}) = \exp(-\frac{\|\mathbf{x}-\mathbf{t}\|^2}{\gamma})$,
- $K(\mathbf{x}, \mathbf{t}) = \tanh(a\mathbf{x}^T\mathbf{t} + b)$.

These functions are known as linear, polynomial, Gaussian, and sigmoid kernels, respectively [8].

A valid kernel corresponds to an inner product in some feature space. A necessary and sufficient condition to check the validity of a kernel function is the Mercer condition [9]. This condition states that any symmetric positive definite matrix can be regarded as a kernel matrix. By restricting a kernel function to a set of points $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}$, the corresponding kernel matrix $K_{N \times N}$ is a matrix that the element in row i and column j is $K(\mathbf{x}_i, \mathbf{x}_j)$ as follows

$$K = \begin{bmatrix} K(\mathbf{x}_1, \mathbf{x}_1) & \dots & K(\mathbf{x}_N, \mathbf{x}_N) \\ \vdots & \ddots & \vdots \\ K(\mathbf{x}_N, \mathbf{x}_1) & \dots & K(\mathbf{x}_N, \mathbf{x}_N) \end{bmatrix}. \quad (2.34)$$

In other words, a real-valued function $K(\mathbf{x}, \mathbf{t})$ satisfies Mercer condition if for any square integrable function $g(\mathbf{x})$, have

$$\int \int g(\mathbf{x})K(\mathbf{x}, \mathbf{t})g(\mathbf{t})d\mathbf{x}d\mathbf{t} \geq 0. \quad (2.35)$$

Therefore, using a valid and suitable kernel function, the optimization problem Eq. 2.29 becomes the following problem

$$\begin{aligned} \max_{\alpha} & \left\{ \sum_{i=1}^N \alpha_i - \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \right\} \\ \text{s.t. } & \sum_{i=1}^N \alpha_i y_i = 0, \\ & 0 \leq \alpha_i \leq C, \quad i = 1, \dots, N, \end{aligned} \quad (2.36)$$

which is still a convex QP problem and is solved to find α_i 's. Moreover, for classifying new data, the similarity of the input sample \mathbf{x} is compared with all training data corresponding to the support vectors by the following formula

$$\hat{y} = \text{sign}\left(w_0 + \sum_{\alpha_i > 0} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})\right), \quad (2.37)$$

where

$$w_0 = y_{s_j} - \sum_{\alpha_i > 0} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_{s_j}), \quad (2.38)$$

such that \mathbf{x}_{s_j} is an arbitrary sample from the support vectors set.

Now it is important to mention some theorems about producing new kernels based on known kernel functions.

Theorem 2.1 A non-negative linear combination of some valid kernel functions is also a valid kernel function [10].

Proof Let's K_1, \dots, K_m satisfy the Mercer condition, we are going to show, $K_{new} = \sum_{i=1}^m a_i K_i$ where $a_i \geq 0$, satisfies the Mercer condition too. So, we have:

$$\begin{aligned} & \int \int g(\mathbf{x}) K_{new}(\mathbf{x}, \mathbf{t}) g(\mathbf{t}) d\mathbf{x} dt = \\ & \int \int g(\mathbf{x}) \sum_{i=1}^m a_i K_i(\mathbf{x}, \mathbf{t}) g(\mathbf{t}) d\mathbf{x} dt = \\ & \sum_{i=1}^m a_i \int \int g(\mathbf{x}) K_i(\mathbf{x}, \mathbf{t}) g(\mathbf{t}) d\mathbf{x} dt \geq 0. \end{aligned}$$

Theorem 2.2 The product of some valid kernel functions is also a valid kernel function [10].

Proof A similar way can be applied to this theorem. \square

The first implication of theorem 2.1 and theorem 2.2 is if K be a valid Mercer kernel function, then any polynomial function with positive coefficients of K , makes also a valid Mercer kernel function. On the other hand, the $\exp(K)$ also makes a valid Mercer kernel function (i.e. consider Maclaurin expansion of $\exp()$ for the proof) [11, 12].

2.3 SVM Regressors

The support vector machine model for classification tasks can be modified for the regression problems. This can be achieved by applying the ϵ -insensitive loss function to the model [13]. This loss function is defined as

$$\text{loss}(y, \hat{y}; \epsilon) = |y - \hat{y}|_\epsilon = \begin{cases} 0 & |y - \hat{y}| \leq \epsilon, \\ |y - \hat{y}| - \epsilon & \text{otherwise,} \end{cases}$$

where $\epsilon \in \mathbb{R}^+$ is a hyper-parameter which specifies the ϵ -tube. Every predicted point outside this tube, will incorporate in the model with a penalty. The unknown function is also defined as

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + w_0,$$

where $\mathbf{x} \in \mathbb{R}^d$ and $y \in \mathbb{R}$. The formulation of the primal form of the support vector regression model is constructed as

$$\begin{aligned} \min_{\mathbf{w}, w_0, \xi, \xi^*} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + C \sum_{i=1}^N (\xi_i + \xi_i^*) \\ \text{s.t.} \quad & y_i - \mathbf{w}^T \phi(\mathbf{x}_i) - w_0 \leq \epsilon + \xi_i, \quad i = 1, \dots, N, \\ & \mathbf{w}^T \phi(\mathbf{x}_i) + w_0 - y_i \leq \epsilon + \xi_i^*, \quad i = 1, \dots, N, \\ & \xi_i, \xi_i^* \geq 0, \quad i = 1, \dots, N. \end{aligned} \tag{2.39}$$

Using the Lagrangian multipliers, the dual form of this optimization problem leads to

$$\begin{aligned} \max_{\alpha, \alpha^*} \quad & -\frac{1}{2} \sum_{i,j=1}^N (\alpha_i - \alpha_i^*) (\alpha_j - \alpha_j^*) K(\mathbf{x}_i, \mathbf{x}_j) \\ & - \epsilon \sum_{i=1}^N (\alpha_i + \alpha_i^*) + \sum_{i=1}^N y_i (\alpha_i - \alpha_i^*) \\ \text{s.t.} \quad & \sum_{i=1}^N (\alpha_i - \alpha_i^*) = 0, \\ & \alpha_i, \alpha_i^* \in [0, C]. \end{aligned}$$

The bias variable w_0 follows the KKT conditions

$$w_0 = \frac{1}{|S|} \sum_{s \in S} \left[y_s - \epsilon - \sum_{i=1}^N (\alpha_i - \alpha_i^*) K(\mathbf{x}_s, \mathbf{x}_i) \right],$$

where $|S|$ is the cardinality of support vectors set. The unknown function $y(\mathbf{x})$ in the dual form can be computed as

$$y(\mathbf{x}) = \sum_{i=1}^N (\alpha_i - \alpha_i^*) K(\mathbf{x}, \mathbf{x}_i) + w_0.$$

2.4 LS-SVM Classifiers

The least-squares support vector machine (LS-SVM) is a modification of SVM formulation for machine learning tasks which was originally proposed by Suykens and Vandewalle [14]. The LS-SVM replaces the inequality constraints of SVM's primal model with equality ones. Also, the slack variables loss function changes to the squared error loss function. Taking advantage of these changes, the dual problem leads to a system of linear equations. Solving this system of linear equations can be more computationally efficient than a quadratic programming problem in some cases. Although this reformulation preserves the kernel trick property, the sparseness of the model is lost. Here, the formulation of LS-SVM for two-class classification tasks will be described.

Same as support vector machines, the LS-SVM considers the separating hyperplane in a feature space:

$$y(\mathbf{x}) = \text{sign} [\mathbf{w}^T \phi(\mathbf{x}) + w_0].$$

Then, the primal form is formulated as

$$\begin{aligned} \min_{\mathbf{w}, e, w_0} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \sum_{i=1}^N e_i^2 \\ \text{s.t.} \quad & y_i [\mathbf{w}^T \phi(\mathbf{x}_i) + w_0] = 1 - e_i, \quad i = 1, \dots, N. \end{aligned}$$

where γ is the regularization parameter and e_i is the slack variables which can be positive or negative. The corresponding Lagrangian function is:

$$\mathcal{L}(\mathbf{w}, w_0, e, \alpha) = \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \sum_{i=1}^N e_i^2 - \sum_{i=1}^N \alpha_i \{y_i [\mathbf{w}^T \phi(\mathbf{x}_i) + w_0] - 1 + e_i\}.$$

The optimality conditions yields

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 \rightarrow \mathbf{w} = \sum_{i=1}^N \alpha_i y_i \phi(\mathbf{x}_i), \\ \frac{\partial \mathcal{L}}{\partial w_0} = 0 \rightarrow \sum_{i=1}^N \alpha_i y_i = 0, \\ \frac{\partial \mathcal{L}}{\partial e_i} = 0 \rightarrow \alpha_i = \gamma e_i, \quad i = 1, \dots, N, \\ \frac{\partial \mathcal{L}}{\partial \alpha_i} = 0 \rightarrow y_i [\mathbf{w}^T \phi(\mathbf{x}_i) + w_0] - 1 + e_i = 0, \quad i = 1, \dots, N, \end{cases} \quad (2.40)$$

which can be written in the matrix form

$$\begin{bmatrix} 0 & y^T \\ y & \Omega + I/\gamma \end{bmatrix} \begin{bmatrix} w_0 \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ 1_v \end{bmatrix},$$

where

$$\begin{aligned} Z^T &= [\phi(\mathbf{x}_1)^T y_1; \dots; \phi(\mathbf{x}_N)^T y_N], \\ y &= [y_1; \dots; y_N], \\ 1_v &= [1; \dots; 1], \\ e &= [e_1; \dots; e_N], \\ \alpha &= [\alpha_1; \dots; \alpha_N], \end{aligned}$$

and

$$\begin{aligned} \Omega_{i,j} &= y_i y_j \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) \\ &= y_i y_j K(\mathbf{x}_i, \mathbf{x}_j), \quad i, j = 1, \dots, N. \end{aligned}$$

The classifier in the dual form takes the form

$$y(\mathbf{x}) = \text{sign} \left[\sum_{i=1}^N \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) + w_0 \right],$$

where $K(\mathbf{x}, \mathbf{t})$ is a valid kernel function.

2.5 LS-SVM Regressors

The LS-SVM for function estimation, known as LS-SVR, is another type of LS-SVM which deals with regression problems [15]. The LS-SVR primal form can be related to the ridge regression model, but for a not-explicitly defined feature map, the dual form with a kernel trick sense can be constructed. The LS-SVM for function estimation, considers the unknown function as

$$y(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + w_0,$$

and then formulates the primal optimization problem as:

$$\begin{aligned} \min_{\mathbf{w}, e, w_0} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + \gamma \frac{1}{2} \sum_{i=1}^N e_i^2 \\ \text{s.t.} \quad & y_i = \mathbf{w}^T \phi(\mathbf{x}_i) + w_0 + e_i, \quad i = 1, \dots, N. \end{aligned}$$

By applying the Lagrangian function and computing the optimality conditions, the dual form of the problem takes the form

$$\begin{bmatrix} 0 & 1^T \\ 1 & \Omega + I/\gamma \end{bmatrix} \begin{bmatrix} w_0 \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix},$$

where

$$\begin{aligned}y &= [y_1; \dots; y_N], \\1_v &= [1; \dots; 1], \\ \alpha &= [\alpha_1; \dots; \alpha_N], \\ \Omega_{i,j} &= \phi(\mathbf{x}_i)^T \phi(\mathbf{x}_j) = K(\mathbf{x}_i, \mathbf{x}_j).\end{aligned}$$

Also, the unknown function in the dual form can be described as

$$y(\mathbf{x}) = \sum_{i=1}^N \alpha_i K(\mathbf{x}_i, \mathbf{x}) + w_0,$$

with a valid kernel function $K(\mathbf{x}, \mathbf{t})$.

References

1. Vapnik, V., Chervonenkis, A.: A note one class of perceptrons. *Autom. Remote. Control.* **44**, 103–109 (1964)
2. Frank, M., Wolfe, P.: An algorithm for quadratic programming. *Nav. Res. Logist. Q.* **3**, 95–110 (1956)
3. Murty, K. G., Yu, F. T.: Linear complementarity, linear and nonlinear programming. Helderman-Verlag, Berlin (1988)
4. Karush, W.: Minima of functions of several variables with inequalities as side constraints. M. Sc. Dissertation. Dept. of Mathematics, Univ. of Chicago (1939)
5. Kuhn, H. W., Tucker, A. W.: Nonlinear programming. Berkeley Symposium on Mathematical Statistics and Probability, University of California Press, Berkeley (1951)
6. Cortes, C., Vapnik, V.: Support-vector networks. *Machine learning* **20**, 273–297 (1995)
7. Vapnik, V.: The nature of statistical learning theory. Springer, Berlin (2000)
8. Cheng, K., Lu, Z., Wei, Y., Shi, Y., Zhou, Y.: Mixed kernel function support vector regression for global sensitivity analysis. *Mech. Syst. Signal Process.* **96**, 201–214 (2017)
9. Mercer, J.: XVI. Functions of positive and negative type, and their connection the theory of integral equations. *Philosophical transactions of the royal society of London. Series A, containing papers of a mathematical or physical character.* **209**, 415–446 (1909)
10. Zanaty, E. A., Afifi, A.: Support vector machines (SVMs) with universal kernels. *Appl Artif. Intell.* **25**, 575–589 (2011)
11. Genton, M. G.: Classes of kernels for machine learning: a statistics perspective. *J. Mach. Learn. Res.* **2**, 299–312 (2001)
12. Shawe-Taylor, J., Cristianini, N.: Kernel methods for pattern analysis. Cambridge university press, UK (2004)
13. Drucker, H., Burges, C. JC., Kaufman, L., Smola, A. J., Vapnik, V.: Support vector regression machines. *Adv. Neural Inf. Process. Syst.* **9**, 155–161 (1997)
14. Suykens, J. AK., Vandewalle, J.: Least squares support vector machine classifiers. *Neural Process. Lett.* **9** 293—300 (1999)
15. Suykens, J. AK., Gestel, T. V., Brabanter, J. D., Moor, B. D., Vandewalle, J.: Least Squares Support Vector Machines. World Scientific, Singapore (2002)

Part II

Special Kernel Classifiers

Chapter 3

Fractional Chebyshev Kernel Functions: Theory and Application

Amir Hosein Hadian Rasanan and Sherwin Nedaei Janbesaraei and Dumitru Baleanu

Abstract Orthogonal functions have many useful properties and can be used for different purposes in machine learning. One of the main applications of the orthogonal functions is producing powerful kernel functions for the support vector machine algorithm. Maybe the simplest orthogonal function that can be used for producing kernel functions, is the Chebyshev polynomials. In this chapter, after reviewing some essential properties of Chebyshev polynomials and fractional Chebyshev functions, various Chebyshev kernel functions are presented, and fractional Chebyshev kernel functions are introduced. Finally, the performance of the various Chebyshev kernel functions is illustrated on two sample datasets.

3.1 Introduction

As mentioned in the previous chapters, the kernel function plays a crucial role in the performance of the SVM algorithms. In the literature of the SVM, various kernel functions have been developed and applied on several data sets [15, 11, 16, 47, 49], but each kernel function has its own benefits and limitations [14, 15]. The radial basis function (RBF) [18, 19] and polynomial kernel functions [54, 55] perhaps are the most popular ones, because they are easy to learn, have acceptable performance in pattern classification, and are very computationally efficient. However, there are

Amir Hosein Hadian Rasanan

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran, e-mail: amir.h.hadian@gmail.com

Sherwin Nedaei Janbesaraei

School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran, e-mail: sherwin.nedaei@gmail.com

Dumitru Baleanu

Department of Mathematics, Cankaya University, Ankara, 06530, Turkey e-mail: dumitru@cankaya.edu.tr

many other examples where the performance of those kernels is not satisfactory [48]. One of the well-established alternatives to these two kernels is the orthogonal kernel functions which have many useful properties embedded in their nature. These orthogonal functions are very useful in various fields of science as well as machine learning [53, 49, 46]. It can be said that the simplest family of these functions is Chebyshev. This family of orthogonal functions has been used in different cases such as signal and image processing [1], digital filtering [2], spectral graph theory [30], astronomy [38], numerical analysis [3, 4, 5, 6, 7, 8, 9], and machine learning [34]. On the other hand, in the literature of numerical analysis and scientific computing, the Chebyshev polynomials have been used for solving various problems in fluid dynamics [33], theoretical physics [32], control [39], and finance [40, 41]. One of the exciting applications of the Chebyshev polynomials has been introduced by Mall and Chakraverty [37], where they used the Chebyshev polynomials as an activation function of functional link neural network. The functional link neural network is a kind of single-layer neural network which utilizes orthogonal polynomials as the activation function. This framework based on Chebyshev polynomials has been used for solving various types of equations such as ordinary, partial, or system of differential equations [36, 35, 29].

Chebyshev Polynomials are named after Russian mathematician **Pafnuty Lvovich Chebyshev** (1821-1894). P.L. Chebyshev, the "extraordinary Russian mathematician", had some significant contributions to mathematics during his career. His papers on the calculation of equations root, multiple integrals, the convergence of Taylor series, theory of probability, and Poisson's weak law of large numbers, and also, integration by means of logarithms brought to him worldwide fame. Chebyshev wrote an important book titled "Teoria sravnenny" which was submitted for his doctorate in 1849 won a prize from *Académie des Sciences* of Paris. It was 1854 when he introduced the famous Chebyshev orthogonal Polynomials. He was appointed to a professorship at St Petersburg University officially, for 22 years. Finally, he died in St Petersburg on 26 November 1894^a.

^a For more information about **Chebyshev** and his contribution in orthogonal polynomials visit: <http://mathshistory.st-andrews.ac.uk/Biographies/Chebyshev.html>

Utilizing the properties of orthogonal functions in machine learning has always been attractive for researchers. The first Chebyshev kernel function is introduced in 2006 by Ye et al. [10]. Thereafter, the use of orthogonal Chebyshev kernel has been under investigation by many researchers [27, 11, 21, 22]. In 2011, Ozer et al. proposed a set of generalized Chebyshev kernels which made it possible for the input to be a vector rather than a single variable [11]. The generalized Chebyshev kernel made it easy to have modified Chebyshev kernels in which weight function can be an exponential function; for example, Ozer et al. used the Gaussian kernel function specifically. In fact, this brings more nonlinearity to the kernel function. Hence, the combination of Chebyshev kernel with other well-known kernels has been

investigated too, for instance, the Chebyshev-Gaussian and Chebyshev-Wavelet [21] kernel functions are bases on this combination. Also, Jinwei Zhao et al. [22] proposed the unified Chebyshev polynomials, a new sequence of orthogonal polynomials and consequently unified Chebyshev kernel through combining Chebyshev polynomials of the first and second kind, which has shown to have excellent generalization performance and prediction accuracy.

The classical orthogonal polynomials proved to be highly efficient in many applied problems. The uniform proximity and orthogonality, have attracted researchers in the kernel learning domain. In fact, **Vladimir Vapnik**, first introduced the approximation of real-valued functions using n-dimensional Hermit polynomials of classical orthogonal polynomials (Statistical Learning Theory, Wiley, USA, 1998). Then, a new perspective opened, later in 2006, Ye and colleagues [10] constructed an orthogonal Chebyshev kernel based on the Chebyshev polynomials of the first kind. Based on what is said in [10], "As Chebyshev polynomial has the best uniform proximity and its orthogonality promises the minimum data redundancy in feature space, it is possible to represent the data with fewer support vectors.".

One of the recent improvements in the field of special functions is the development of fractional orthogonal polynomials [17]. The fractional orthogonal functions can be obtained by some nonlinear transformation function and have much better performance in function approximation [42, 43, 44]. However, these functions have not been used as a kernel. Thus, this chapter first aims to present essential backgrounds for Chebyshev polynomial and its fractional version, then bring all the Chebyshev kernel functions together, and finally introduce and examine the fractional Chebyshev kernel functions.

This chapter is organized as follows. The basic definitions and properties of orthogonal Chebyshev polynomials and the fractional Chebyshev functions are presented in Section 3.2. Then, the ordinary Chebyshev kernel function and two previously proposed kernels based on these polynomials are discussed and the novel fractional Chebyshev kernel functions are introduced in Section 3.3. In Section 3.4, the results of experiments of both the ordinary Chebyshev kernel and the fractional one are covered and then a comparison between the obtained accuracy results of the mentioned kernels and RBF and polynomial kernel functions in the SVM algorithm on well-known datasets are exhibited to specify the validity and efficiency of the fractional kernel functions. Finally, in Section 3.5 conclusion remarks of this chapter are presented.

3.2 Preliminaries

Definition and basic properties of Chebyshev orthogonal polynomials of the first kind are presented in this section. In addition to the basics of these polynomials, the fractional form of this family is presented and discussed.

3.2.1 Properties of Chebyshev polynomials

There are four kinds of Chebyshev polynomials, but the focus of this chapter is on introducing the first kind of Chebyshev polynomials which is denoted by $T_n(x)$. The interested readers can investigate the other types of Chebyshev polynomials in [31].

The power of polynomials originally comes from their relation with trigonometric functions (sine and cosine) that are very useful in describing all kinds of natural phenomena [23]. Since $T_n(x)$ is a polynomial, then it is possible to define $T_n(x)$ using trigonometric relations.

Let assume z be a complex number over a unit circle $|z| = 1$, in which θ is the argument of z and $\theta \in [0, 2\pi]$, in other words:

$$x = \Re z = \frac{1}{2}(z + z^{-1}) = \cos \theta \quad \in [-1, 1], \quad (3.1)$$

where \Re is the real axis of a complex number.

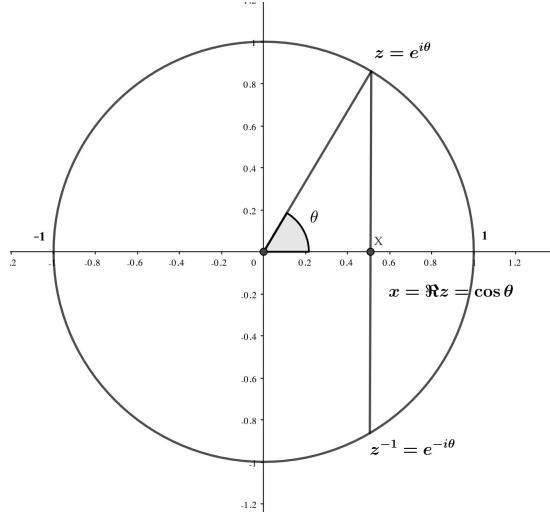
Considering Chebyshev polynomials of the first kind is denoted by $T_n(x)$, so one can define [50, 51, 52, 13]:

$$T_n(x) = \Re z^n = \frac{1}{2}(z^n + z^{-n}) = \cos n\theta. \quad (3.2)$$

Therefore, the n -th order of Chebyshev polynomial can be obtained by:

$$T_n(x) = \cos n\theta, \quad \text{where } x = \cos \theta. \quad (3.3)$$

The relation between x , z and θ is illustrated in Fig. 3.1:

Fig. 3.1: The plot of the relation between x , z , and θ

Based on definition 3.2, the Chebyshev polynomials for some n are defined as follows:

$$\Re z^0 = \frac{1}{2}(z^0 + z^0), \quad \Rightarrow \quad T_0(x) = 1, \quad (3.4)$$

$$\Re z^1 = \frac{1}{2}(z^1 + z^{-1}), \quad \Rightarrow \quad T_1(x) = x, \quad (3.5)$$

$$\Re z^2 = \frac{1}{2}(z^2 + z^{-2}), \quad \Rightarrow \quad T_2(x) = 2x^2 - 1, \quad (3.6)$$

$$\Re z^3 = \frac{1}{2}(z^3 + z^{-3}), \quad \Rightarrow \quad T_3(x) = 4x^3 - 3x, \quad (3.7)$$

$$\Re z^4 = \frac{1}{2}(z^4 + z^{-4}), \quad \Rightarrow \quad T_4(x) = 8x^4 - 8x^2 + 1. \quad (3.8)$$

Consequently, by considering this definition for $T_{n+1}(x)$ as follow:

$$\begin{aligned} T_{n+1}(x) &= \frac{1}{2}(z^{n+1} + z^{-n-1}) \\ &= \frac{1}{2}(z^n + z^{-n})(z + z^{-1}) - \frac{1}{2}(z^{n-1} + z^{1-n}) \\ &= 2 \cos(n\theta) \cos(\theta) - \cos((n-1)\theta), \end{aligned} \quad (3.9)$$

the following recursive relation can be obtained for the Chebyshev polynomials:

$$\begin{aligned} T_0(x) &= 1, \\ T_1(x) &= x, \\ T_n(x) &= 2xT_{n-1}(x) - T_{n-2}(x), \quad n \geq 2. \end{aligned} \quad (3.10)$$

Thus, any order of the Chebyshev polynomials can be generated using this recursive formula.

Additional to the recursive formula, the Chebyshev polynomials can be obtained directly by the following expansion for $n \in \mathbb{Z}^+$ [50, 51, 52, 13]:

$$T_n(x) = \sum_{k=0}^{\lfloor \frac{1}{2}n \rfloor} (-1)^k \frac{n!}{(2k)!(n-2k)!} (1-x^2)^k x^{n-2k}. \quad (3.11)$$

The other way to obtain the Chebyshev polynomials is by solving their Sturm-Liouville differential equation.

Theorem 3.1 [50, 51, 52, 13] $T_n(x)$ is the solution of the following second-order linear Sturm-Liouville differential equation:

$$(1-x^2) \frac{d^2y}{dx^2} - x \frac{dy}{dx} + n^2 y = 0, \quad (3.12)$$

where $-1 < x < 1$ and n is an integer number.

Proof By considering $\frac{d}{dx} \cos^{-1}(x) = -\frac{1}{\sqrt{1-x^2}}$, we have

$$\begin{aligned} \frac{d}{dx} T_n(x) &= \frac{d}{dx} \cos(n \cos^{-1} x), \\ &= n \cdot \frac{1}{\sqrt{1-x^2}} \sin(n \cos^{-1} x). \end{aligned} \quad (3.13)$$

In a similar way we can write that

$$\begin{aligned} \frac{d^2}{dx^2} T_n(x) &= \frac{d}{dx} \left\{ \frac{n}{\sqrt{1-x^2}} \sin(n \cos^{-1} x) \right\}, \\ &= \frac{nx}{\sqrt[3]{(1-x^2)^2}} \sin(n \cos^{-1} x) + \frac{n}{\sqrt{1-x^2}} \cos(n \cos^{-1} x) \cdot \frac{-n}{\sqrt{1-x^2}}, \\ &= \frac{nx}{\sqrt[3]{1-x^2}} \sin(n \cos^{-1} x) - \frac{n^2}{1-x^2} \cos(n \cos^{-1} x). \end{aligned}$$

So, by replacing the derivatives with their obtained formula in 3.12, the following is yielded:

$$(1-x^2) \frac{d^2 T_n(x)}{dx^2} - x \frac{dT_n(x)}{dx} + n^2 T_n(x) = \frac{nx}{\sqrt{1-x^2}} \sin(n \cos^{-1} x) - n^{-2} \cos(n \cos^{-1} x), \\ - \frac{nx}{\sqrt{1-x^2}} \sin(n \cos^{-1}) + n^2 \cos(n \cos^{-1} x), \\ = 0,$$

which yields that $T_n(x)$ is a solution to 3.12. \square

There are several ways to use these polynomials in Python such as using the exact formula 3.3, using the NumPy package ¹, and also implementing the recursive formula 3.10. So, the symbolic Python implementation of this recursive formula using the *sympy* library is:

Program Code

```
import sympy

x = sympy.Symbol("x")

def Tn(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    elif n >= 2:
        return (2 * x * Tn(x, n - 1)) - Tn(x, n - 2)

sympy.expand(sympy.simplify(Tn(x, 3)))
```

$$> 4x^3 - 3x$$

In the above code, the *3rd order* of first-kind Chebyshev polynomial is generated which is equal to $4x^3 - 3x$.

In order to explore the behavior of the Chebyshev polynomials, there are some useful properties that are available. The first property of Chebyshev polynomials is their orthogonality.

Theorem 3.2 [50, 51, 52] $\{T_n(x)\}$ forms a sequence of orthogonal polynomials which are orthogonal to each other over the interval $[-1, 1]$, with respect to the following weight function:

$$w(x) = \frac{1}{\sqrt{1-x^2}}, \quad (3.14)$$

so it can be concluded that

$$\int_{-1}^1 T_n(x) T_m(x) w(x) dx = \frac{\pi c_n}{2} \delta_{n,m}, \quad (3.15)$$

¹ <https://numpy.org/doc/stable/reference/routines.polynomials.chebyshev.html>

where $c_0 = 2$, $c_n = 1$ for $n \geq 1$, and $\delta_{n,m}$ is the Kronecker delta function.

Proof Suppose $x = \cos \theta$ and $n \neq m$, so we have:

$$\int_{-1}^1 T_n(x)T_m(x) \frac{1}{\sqrt{1-x^2}} dx, \quad (3.16)$$

$$= \int_{-\pi}^{\pi} T_n(\cos \theta)T_m(\cos \theta) \frac{1}{\sqrt{1-\cos^2 \theta}} (-\sin \theta) d\theta, \quad (3.17)$$

$$= \int_0^\pi \cos n\theta \cos m\theta d\theta,$$

$$= \int_0^\pi \frac{1}{2} [\cos(n+m)\theta + \cos(n-m)\theta] d\theta,$$

$$= \frac{1}{2} \left[\frac{1}{n+m} \sin(n+m)\theta + \frac{1}{m-n} \sin(m-n)\theta \right] \Big|_0^\pi = 0.$$

In case of $n = m \neq 0$, we have

$$\begin{aligned} & \int_{-1}^1 T_n(x)T_m(x) \frac{1}{\sqrt{1-x^2}} dx, \\ &= \int_0^\pi \cos n\theta \cos n\theta d\theta, \\ &= \int_0^\pi \cos^2 n\theta d\theta = \int_0^\pi \frac{1}{2} (1 + \cos 2n\theta) d\theta, \\ &= \frac{1}{2} \left[\theta + \frac{-1}{2n} \sin 2n\theta \right] \Big|_0^\pi = \frac{\pi}{2}. \end{aligned} \quad (3.18)$$

Also in the case where $n = m = 0$, we can write that

$$\int_{-1}^1 T_n(x)T_m(x) \frac{1}{\sqrt{1-x^2}} dx = \int_0^\pi 1 d\theta = \pi. \quad (3.19)$$

□

The orthogonality of these polynomials provides an efficient framework for computing the kernel matrix which will be discussed later. Computationally efficiency is not the only property of the Chebyshev polynomials, but there is even more about it. For example, the maximum value for $T_n(x)$ is no more than 1 since for $-1 \leq x \leq 1$, $T_n(x)$ is defined as the cosine of an argument. This property prevents overflow during computation. Besides, the Chebyshev polynomial of order $n > 0$ has exactly n roots and $n+1$ local extremum in the interval $[-1, 1]$, which two extremums are endpoints at ± 1 points [23]. So, for $k = 1, 2, \dots, n$, roots of $T_n(x)$ are obtained as follows:

$$x_k = \cos\left(\frac{(2k-1)\pi}{2n}\right). \quad (3.20)$$

Note that $x = 0$ is a root of $T_n(x)$ for all odd orders n , other roots are symmetrically placed on either side of $x = 0$. For $k = 0, 1, \dots, n$ extremums of Chebyshev polynomial of first kind, can be found using

$$x = \cos\left(\frac{\pi k}{n}\right). \quad (3.21)$$

Chebyshev polynomials are even and odd symmetrical, meaning even orders only have the even powers of x and odd orders only the odd powers of x . Therefore we have

$$T_n(x) = (-1)^n T_n(-x) = \begin{cases} T_n(-x), & n \text{ even} \\ -T_n(-x), & n \text{ odd} \end{cases}. \quad (3.22)$$

Additionally, there exist some other properties of the Chebyshev polynomials at the boundaries:

- $T_n(1) = 1$,
- $T_n(-1) = (-1)^n$,
- $T_{2n}(0) = (-1)^n$,
- $T_{2n+1}(0) = 0$.

Fig. 3.2 demonstrates Chebyshev polynomials of the first kind up to 6th order.

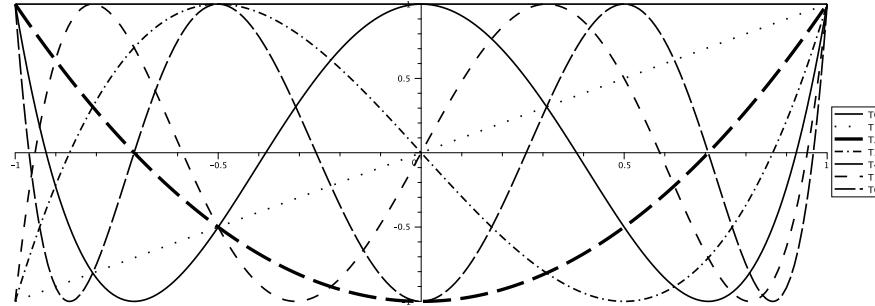


Fig. 3.2: Chebyshev polynomials of first kind up to 6th order

3.2.2 Properties of fractional Chebyshev functions

Fractional Chebyshev functions are a general form of the Chebyshev polynomials. The main difference between the Chebyshev polynomial and the fractional Chebyshev functions is that the order of x can be any positive real number in the latter. This generalization seems to improve the approximation power of the Chebyshev polynomials. To introduce the fractional Chebyshev functions, first, it is needed to

introduce a mapping function which is used to define fractional Chebyshev function of order α over a finite interval $[a, b]$, which is [20]:

$$x' = 2\left(\frac{x-a}{b-a}\right)^\alpha - 1. \quad (3.23)$$

By utilizing this transformation the fractional Chebyshev functions can be obtained as follows [20]:

$$FT_n^\alpha(x) = T_n(x') = T_n\left(2\left(\frac{x-a}{b-a}\right)^\alpha - 1\right), \quad a \leq x \leq b, \quad (3.24)$$

where $\alpha \in \mathbb{R}^+$ is the "fractional order" of function which is chosen relevant to the context. Considering 3.10, the recursive form of fractional Chebyshev functions defines as [20]:

$$\begin{aligned} FT_0^\alpha(x) &= 1, & FT_1^\alpha(x) &= 2\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\ FT_n^\alpha(x) &= 2\left(2\left(\frac{x-a}{b-a}\right)^\alpha - 1\right)FT_{n-1}^\alpha(x) - FT_{n-2}^\alpha(x), & n &\geq 1. \end{aligned} \quad (3.25)$$

The formulation of some of these functions are presented here:

$$\begin{aligned} FT_0^\alpha(x) &= 1, \\ FT_1^\alpha(x) &= 2\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\ FT_2^\alpha(x) &= 8\left(\frac{x-a}{b-a}\right)^{2\alpha} - 8\left(\frac{x-a}{b-a}\right)^\alpha + 1, \\ FT_3^\alpha(x) &= 32\left(\frac{x-a}{b-a}\right)^{3\alpha} - 48\left(\frac{x-a}{b-a}\right)^{2\alpha} + 18\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\ FT_4^\alpha(x) &= 128\left(\frac{x-a}{b-a}\right)^{4\alpha} - 256\left(\frac{x-a}{b-a}\right)^{3\alpha} + 160\left(\frac{x-a}{b-a}\right)^{2\alpha} - 32\left(\frac{x-a}{b-a}\right)^\alpha + 1. \end{aligned} \quad (3.26)$$

The readers can use the following Python code to generate any order of *Chebyshev Polynomials of Fractional Order* symbolically:

Program Code

```
import sympy
x = sympy.Symbol("x")
eta = sympy.Symbol(r'\eta')
alpha = sympy.Symbol(r'\alpha')
a = sympy.Symbol("a")
b = sympy.Symbol("b")
x=sympy.sympify(2*((x-a)/(b-a))**alpha -1)

def FTn(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
```

```
elif n >= 2:  
    return (2 * x*(FTn(x, n - 1))) - FTn(x, n - 2)
```

For example the 5th order can be generated as:

Program Code

```
sympy.expand(sympy.simplify(FTn(x, 5)))
```

$$> 512\left(\frac{x-a}{b-a}\right)^{5\alpha} - 1280\left(\frac{x-a}{b-a}\right)^{4\alpha} + 1120\left(\frac{x-a}{b-a}\right)^{3\alpha} - 400\left(\frac{x-a}{b-a}\right)^{2\alpha} + 50\left(\frac{x-a}{b-a}\right)^\alpha - 1$$

Due to Chebyshev polynomials are orthogonal respecting the weight function $\frac{1}{\sqrt{1-x^2}}$ over $[-1, 1]$, therefore using $x' = 2\left(\frac{x-a}{b-a}\right)^\alpha - 1$ one can conclude that fractional Chebyshev polynomials also are orthogonal over a finite interval respecting the following weight function:

$$w(x) = \frac{1}{\sqrt{1 - (2\left(\frac{x-a}{b-a}\right)^\alpha - 1)^\alpha}}. \quad (3.27)$$

Therefore we can define the orthogonality relation of the fractional Chebyshev polynomial as:

$$\int_{-1}^1 T_n(x') T_m(x') w(x') dx' = \int_a^b FT_n^\alpha(x) FT_m^\alpha(x) w(x) dx = \frac{\pi}{2} c_n \delta_{mn}, \quad (3.28)$$

where $c_0 = 2$, $c_n = 1$ for $n \geq 1$, and δ_{mn} is the Kronecker delta function.

The fractional Chebyshev function $FT_n^\alpha(x)$ has exactly n distinct zeros over finite interval $(0, \eta)$ which are:

$$x_k = \left(\frac{1 - \cos\left(\frac{(2k-1)\pi}{2n}\right)}{2} \right)^{\frac{1}{\alpha}}, \quad \text{for } k = 1, 2, \dots, n. \quad (3.29)$$

Moreover, the derivative of the fractional Chebyshev function has exactly $n - 1$ of distinct zeros over the finite interval $(0, \eta)$, which are the *Extrema*:

$$x'_k = \left(\frac{1 - \cos\left(\frac{k\pi}{n}\right)}{2} \right)^{\frac{1}{\alpha}}, \quad \text{for } k = 1, 2, \dots, n - 1. \quad (3.30)$$

Fig. 3.3 shows the fractional Chebyshev functions of the first kind up to 6-th order where $a = 0$, $b = 5$, and α is 0.5.

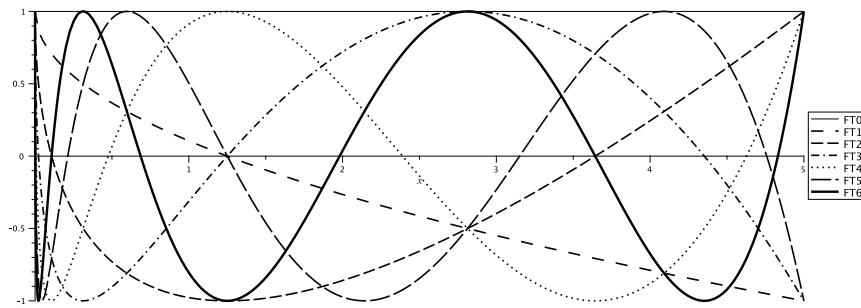


Fig. 3.3: Fractional Chebyshev functions of the first kind up to 6-th order where $a = 0$, $b = 5$, and $\alpha = 0.5$

Also, Fig. 3.4 depicts the fractional Chebyshev functions of the first kind of order 5 for different values of α while η is fixed at 5.

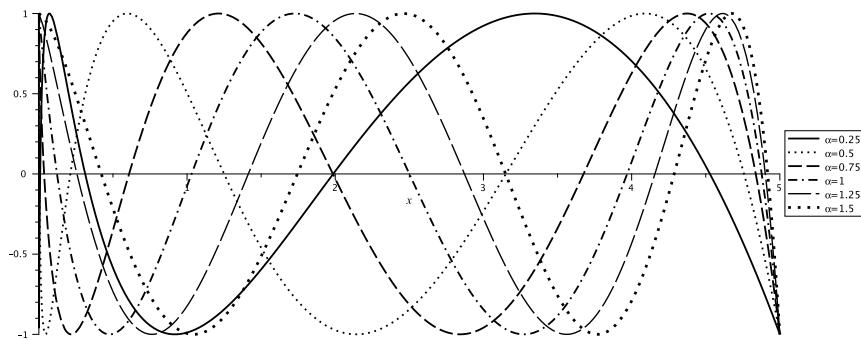


Fig. 3.4: Fractional Chebyshev functions of the first kind of order 5, for different α values

3.3 Chebyshev kernel functions

Following the Chebyshev polynomial principles and their fractional type discussed in the previous sections, this section presents the formulation of Chebyshev kernels. Therefore, in the following, first the ordinary Chebyshev kernel function, then some other versions of this kernel function, and finally the fractional Chebyshev kernels will be explained.

3.3.1 Ordinary Chebyshev kernel function

Many kernels constructed from orthogonal polynomials have been proposed for SVM and other kernel-based learning algorithms [10, 48, 49]. Some characteristics of such kernels have attracted attention such as lower data redundancy in feature space or on some occasions fewer support vectors these kernels need during the fitting procedure which thereby leads to less execution time [24]. On the other hand, orthogonal kernels have shown superior performance in classification problems than traditional kernels like RBF and polynomial [48, 11, 47].

Now, using the fundamental definition of the orthogonal kernel we will formulate the Chebyshev kernel. As we know, the unweighted orthogonal polynomial kernel function for SVM, for scalar inputs is defined as:

$$K(x, z) = \sum_{i=0}^n T_i(x)T_i(z), \quad (3.31)$$

where $T(\cdot)$ denotes the evaluation of the polynomial. x, z are kernel's input arguments, and n is the highest polynomial order. In most applications of the real world, input data is a multidimensional vector, hence two approaches have been proposed to extend one-dimensional polynomial kernel to multi-dimensional vector input [11, 12]:

1. Pairwise

According to Vapnik's theorem [25]:

"Let a multidimensional set of functions be defined by the basis functions that are the tensor products of the coordinate-wise basis functions. Then the kernel that defines the inner product in the n-dimensional basis is the product of one-dimensional kernels."

The multidimensional orthogonal polynomial kernel is constructed as a tensor product of one-dimensional kernel as:

$$K(x, z) = \prod_{j=1}^m K_j(x_j, z_j). \quad (3.32)$$

In this approach, the function evaluation of element pair of each input vectors x and z are multiplied in pairs, which means multiplying the corresponding elements of x and z and then overall kernel output is the multiplication of all outcomes of the previous step. Therefore, for an m-dimensional input vector $x \in \mathbb{R}^m$, given $x = \{x_1, x_2, \dots, x_m\}$, $z \in \mathbb{R}^m$, and $z = \{z_1, z_2, \dots, z_m\}$, the unweighted Chebyshev kernel is defined as:

$$K_{Cheb}(x, z) = \prod_{j=1}^m K_j(x_j, z_j) = \prod_{j=1}^m \sum_{i=0}^n T_i(x_j)T_i(z_j), \quad (3.33)$$

where $\{T_i(\cdot)\}$ are Chebyshev polynomials. Simply by multiplying the weight function defined in 3.14 which the orthogonality of the Chebyshev polynomial of first-kind is defined with (see 3.15), one can construct the pairwise orthogonal Chebyshev kernel function as:

$$K_{Cheb}(x, z) = \frac{\sum_{i=0}^n T_i(x)T_i(z)}{\sqrt{1-xz}}, \quad (3.34)$$

where x and z are scalar valued inputs and for vector input we have:

$$K_{Cheb}(\mathbf{x}, \mathbf{z}) = \prod_{j=1}^m \frac{\sum_{i=0}^n T_i(x_j)T_i(z_j)}{\sqrt{1-x_jz_j}}, \quad (3.35)$$

where m is a dimension of vectors \mathbf{x} and \mathbf{z} .

However this method was first proposed and then, many kernels have been constructed using a pairwise method [10, 12, 26, 57], but kernels of this method suffer from two drawbacks. First, the time complexity, or better to say computational complexity is considerably high, thus, this can be a serious problem when input vectors are of high dimension [11] Second, those kernels constructed on pairwise method lead to poor generalization if one of the kernels ($T_i(x)$ or $T_i(z)$) is close to zero, at the time when two vectors \mathbf{x} and \mathbf{z} are quite similar to each other [27].

2. Vectorized

In this method proposed by Ozer et al. [27] the generalization problem of *pairwise* is tackled by applying the input vectors as a whole rather than per element and this is done by means of evaluation of the inner product of input vectors. Based on the fact that the inner product of two vectors x and z is defined as $\langle x, z \rangle = xz^T$ and considering Eq. 3.31, one can construct the *unweighted Generalized Chebyshev kernel* as [27, 11]:

$$K(\mathbf{x}, \mathbf{z}) = \sum_{i=0}^n T_i(\mathbf{x})T_i^T(\mathbf{z}) = \sum_{i=0}^n \langle T_i(x)T_i(z) \rangle. \quad (3.36)$$

In a similar fashion to Eq. 3.35, the *Generalized Chebyshev Kernel* is defined as [27, 11]:

$$K_{G_Cheb}(\mathbf{x}, \mathbf{z}) = \frac{\sum_{i=0}^n \langle T_i(x), T_i(z) \rangle}{\sqrt{m - \langle x, z \rangle}}, \quad (3.37)$$

where m is the dimensional of input vectors \mathbf{x} and \mathbf{z} .

Note that as the Chebyshev polynomials are orthogonal within $[-1, 1]$, the input data should be normalized according to:

$$x^{new} = \frac{2(x - Min)}{Max - Min} - 1, \quad (3.38)$$

where *Min* and *Max* are minimum and maximum values of the input data, respectively. It is clear that if the input data is a data set, normalization should be done column-wised.

3.3.1.1 Validation of Chebyshev Kernel

According to *Mercer Theorem* introduced at 2.2.1 a valid kernel function needs to be positive semi-definite or equivalently must satisfy the necessary and sufficient conditions of Mercer's Theorem. Here the validity of the generalized Chebyshev kernel will be proved.

Theorem 3.3 [11] *The generalized Chebyshev kernel introduced at Eq. 3.37 is a valid Mercer kernel.*

Proof As Mercer's theorem states, any SVM kernel as a valid kernel must be non-negative, in a precise way:

$$\iint K(x, z) f(x) f(z) dx dz \geq 0. \quad (3.39)$$

The multiplication of two valid kernels is also a valid kernel. Therefore one can express any order of the Chebyshev kernel as a product of two kernel functions:

$$K(x, z) = k_{(1)}(x, z) k_{(2)}(x, z), \quad (3.40)$$

where

$$k_{(1)}(x, z) = \sum_{j=0}^n T_j(x) T_j^T(z) = T_0(x) T_0^T(z) + T_1(x) T_1^T(z) + \dots + T_n(x) T_n^T(z), \quad (3.41)$$

and

$$k_{(2)}(x, z) = \frac{1}{\sqrt{m - \langle x, z \rangle}}. \quad (3.42)$$

Considering $f : \mathbb{R}^m \rightarrow \mathbb{R}$ and assuming that each element is independent of the other elements of the kernel, we can evaluate the Mercer condition for $k_{(1)}(\mathbf{x}, \mathbf{z})$ as follows

$$\begin{aligned}
\iint k_{(1)}(x, z) f(x) f(z) dx dz &= \iint \sum_{j=0}^n T_j(x) T_j^T(z) f(x) f(z) dx dz, \\
&= \sum_{j=0}^n \iint T_j(x) T_j^T(z) f(x) f(z) dx dz, \\
&= \sum_{j=0}^n \left[\int T_j(x) f(x) dx \int T_j^T(z) f(z) dz \right], \\
&= \sum_{j=0}^n \left[\left(\int T_j(x) f(x) dx \right) \left(\int T_j^T(z) f(z) dz \right) \right] \geq 0.
\end{aligned} \tag{3.43}$$

□

Therefore, the kernel $k_{(1)}(x, z)$ is a valid Mercer kernel. To prove that $k_{(2)}(x, z) = \frac{1}{\sqrt{m - \langle x, z \rangle}}$ (in the simplest form where $m = 1$) is a valid kernel function, we can say that $\langle x, y \rangle$ is the linear kernel and the Taylor series of $\frac{1}{\sqrt{1-x \cdot y}}$ is :

$$1 + \frac{1}{2} \langle x, y \rangle + \frac{3}{8} \langle x, y \rangle^2 + \frac{5}{16} \langle x, y \rangle^3 + \frac{35}{128} \langle x, y \rangle^4 + \dots,$$

which for each coefficient $\frac{\prod_{i=1}^n n(2i-1)}{n!2^n} \geq 0$, $k_{(2)}(x, z) = \frac{1}{\sqrt{m - \langle x, z \rangle}}$ is also a valid kernel.

3.3.2 Other Chebyshev kernel functions

3.3.2.1 Chebyhsev-Wavelet kernel

A valid kernel should satisfy the Mercer conditions [25, 28], and based on Mercer's theorem, a kernel made from multiplication or summation of two valid Mercer kernels is also a valid kernel. This idea motivated Jafarzadeh et al. [21] to construct *Chebyshev-Wavelet* kernel which is in fact the multiplication of generalized Chebyshev and Wavelet kernels as follow:

$$\begin{aligned}
k_{G_Cheb}(\mathbf{x}, \mathbf{z}) &= \frac{\sum_{i=0}^n \langle T_i(x), T_i(z) \rangle}{\sqrt{m - \langle x, z \rangle}}, \\
k_{wavelet}(\mathbf{x}, \mathbf{z}) &= \prod_{i=1}^m \left(\cos \left(1.75 \frac{x_j - z_j}{\alpha} \right) \exp \left(-\frac{\|x - z\|^2}{2\alpha^2} \right) \right).
\end{aligned} \tag{3.44}$$

Thus, the **Chebyshev-Wavelet kernel** is introduced as [21]:

$$k_{Cheb-wavelet}(\mathbf{x}, \mathbf{z}) = \frac{\sum_{i=0}^n \langle T_i(x), T_i(z) \rangle}{\sqrt{m - \langle x, z \rangle}} \times \prod_{i=1}^m \left(\cos(1.75 \frac{x_j - z_j}{\alpha}) \exp(-\frac{\|x - z\|^2}{2\alpha^2}) \right), \quad (3.45)$$

where n is the Chebyshev kernel parameter (the order) and a is the wavelet kernel parameter and m is the dimension of the input vector.

3.3.2.2 Unified Chebyshev Kernel (UCK)

As it is already mentioned there are four kinds of Chebyshev polynomials. Zhao et al. [22] proposed a new kernel by combining Chebyshev polynomials of the first (T_n) and second (U_n) kind of Chebyshev polynomials. It should be noted that the Chebyshev polynomials of the second kind are orthogonal on the interval (-1,1) with respect to the following weight function [23]:

$$w_{U_n}(x) = \sqrt{1 - x^2}, \quad (3.46)$$

while the trigonometric relation of the Chebyshev polynomials of the first kind is introduced at 3.3, polynomials of the second kind are described by [23]:

$$U_n(\cos\theta) = \frac{\sin((n+1)\theta)}{\sin(\theta)}, \quad n = 0, 1, 2, \dots \quad (3.47)$$

Similar to the generating function of Chebyshev polynomials of the first kind ($\frac{1-tx}{1-2tx+t^2} = \sum_{n=0}^{\infty} T_n(x)t^n$), the generating function for the second kind is defined as [23]:

$$\frac{1}{1-2xt+t^2} = \sum_{n=0}^{\infty} U_n(x)t^n, \quad |x| \leq 1, |t| \leq 1. \quad (3.48)$$

According to defined weight function Eq. 3.46, Zhao et al. [22] defined the orthogonality relation for the Chebyshev polynomials of the second kind as :

$$\int_{-1}^1 \sqrt{1-x^2} U_m(x) U_n(x) dx = \begin{cases} \frac{\pi}{2}, & (m = n \neq 0) \\ 0, & ((m \neq n) \text{ or } (m = n = 0)) \end{cases}. \quad (3.49)$$

Moreover, they used $P_n(\cdot)$ to denote unified Chebyshev polynomials (UPC), but in this book, notation $P_n(\cdot)$ is used for Legendre Orthogonal Polynomials. Therefore, in order to avoid any ambiguity, we consciously use $UCP_n(\cdot)$ instead. Considering the generating function of the first and the second kind, Zhao et al. [22] constructed the generating function of UCP:

$$\frac{1-xt}{1-axt} \times \frac{1}{1-2xt+t^2} = \sum_{n=0}^{\infty} UCP_n(x)t^n, \quad (3.50)$$

where $x \in [-1, 1]$, $t \in [-1, 1]$, $n = 0, 1, 2, 3, \dots$, and $UCP_n(x)$ is the UCP of the nth order. It is clear that the Chebyshev polynomials of the first and the second kinds are special cases of UCP where $a = 0$ and $a = 1$, respectively. Also, Zhao et al. [22] introduced the recurrence relation of the nth order of UCP as:

$$UCP_n(x) = (a+2)xUCP_{n-1}(x) - (2ax^2 + 1)UCP_{n-2}(x) + axUCP_{n-3}(x). \quad (3.51)$$

Therefore using 3.51 some instances of these polynomials are:

$$\begin{aligned} UCP_0(x) &= 1, \\ UCP_1(x) &= ax + x, \\ UCP_2(x) &= (a^2 + a + 2)x^2 - 1, \\ UCP_3(x) &= (a^3 + a^2 + 2a + 4)x^3 - (a + 3)x, \\ UCP_4(x) &= (a^4 + a^3 + 2a^2 + 4a + 8)x^4 - (a^2 + 3a + 8)x^2 + 1, \end{aligned}$$

For sake of simplicity, we do not go deeper into formulations provided in the corresponding paper. However, the details and proofs are investigated in the relevant paper [22].

On the othe hand, Zhao et al. [22] constructed the UCP kernel as:

$$\iint (m - \langle x, z \rangle + r)^{-\frac{1}{2}} \sum_{j=0}^n UCP_j(x), UCP_j^T(z) f(x) f(z) dx dz \geq 0. \quad (3.52)$$

Hence:

$$k_{UCP}(X, Z) = \frac{\sum_{j=0}^n \langle UCP_j(x) UCP_j(z) \rangle}{\sqrt{(m - \langle x, z \rangle + r)}}, \quad (3.53)$$

where m is the dimension of the input vector, and r is the minimum positive value to prevent the equation from being 0.

3.3.3 Fractional Chebyshev Kernel

Using the weight function Eq. 3.27, with the same approach performed in Eq. 3.35 and Eq. 3.37, we can introduce the corresponding fractional kernels as follows:

$$k_{FCheb}(\mathbf{x}, \mathbf{z}) = \prod_{j=1}^m \frac{\sum_{i=0}^n FT_i^\alpha(x_j) FT_i^\alpha(z_j)}{\sqrt{1 - (2(\frac{x_j z_j - a}{b-a})^\alpha - 1)^\alpha}}, \quad (3.54)$$

$$k_{Gen-FCheb}(\mathbf{x}, \mathbf{z}) = \frac{\sum_{i=0}^n \langle FT_i^\alpha(x), FT_i^\alpha(z) \rangle}{\sqrt{m - (2(\frac{\langle x, z \rangle - a}{b-a})^\alpha - 1)^\alpha}}. \quad (3.55)$$

where m is the dimension of vectors x and z .

3.3.3.1 Validation of Fractional Chebyshev Kernel

To prove validation of fractional Chebyshev kernels introduced in Eq. 3.55 a similar procedure like the proof of the generalized Chebyshev kernel Eq. 3.37 is followed.

Theorem 3.4 *The generalized Chebyshev kernel of fractional order introduced in Eq. 3.55 is a valid Mercer kernel.*

Proof Considering Mercer theorem states any SVM kernel to be a valid kernel must be non-negative (see 3.39) and by use of the fact that multiplication of two valid kernels is also a kernel (see 3.40), we have :

$$k_{(1)}(x, z) = \sum_{j=0}^n FT_j^\alpha(x)FT_j^\alpha(z),$$

$$k_{(2)}(x, z) = w(x) = \frac{1}{\sqrt{1 - (2(\frac{x-a}{b-a})^\alpha - 1)^\alpha}}.$$

Since f is a function where $f : \mathbb{R}^m \rightarrow \mathbb{R}$, we have:

$$\begin{aligned} \iint K_{(1)}(x, z) f(x) f(z) dx dz &= \iint \sum_{j=0}^n (FT_j^\alpha)(x) (FT_j^\alpha)^T(z) f(x) f(z) dx dz, \\ &= \sum_{j=0}^n \iint (FT_j^\alpha)(x) (FT_j^\alpha)^T(z) f(x) f(z) dx dz, \\ &= \sum_{j=0}^n \left[\int (FT_j^\alpha)(x) f(x) dx \int (FT_j^\alpha)^T(z) f(z) dz \right], \\ &= \sum_{j=0}^n \left[\left(\int (FT_j^\alpha)(x) f(x) dx \right) \left(\int (FT_j^\alpha)^T(z) f(z) dz \right) \right] \geq 0. \end{aligned} \tag{3.56}$$

Therefore, the kernel $k_{(1)}(x, z)$ is a valid Mercer kernel. In order to prove that $k_{(2)}(x, z)$ is a valid Mercer kernel too, one can show that $k_{(2)}(x, z)$ is *positive semi-definite*. According to definition Eq. 3.24, both x and b are positive because $0 \leq x \leq b$, $b \in \mathbb{R}^+$ and $\alpha \in \mathbb{R}^+$, in other words by mapping defined in Eq. 3.24, we are sure the output is always positive. Hence, the second kernel Eq. 3.56 or precisely the weight function is positive semi-definite, so $k_{(2)}(x, z) \geq 0$. \square

3.4 Application of Chebyshev kernel functions on real datasets

In this section, we will illustrate the use of *fractional Chebyshev kernel* in *SVM* applied to some real datasets which are widely used by machine learning experts to examine the accuracy of any given model. Then, we will compare the accuracy of the *fractional Chebyshev kernel* with the accuracy of the *Chebyshev kernel*, and two of other well-known kernels used in kernel-based learning algorithms, such as *RBF* and *polynomial kernel*. As we know, applying SVM to a dataset involves a number of pre-processing tasks, such as data cleansing and generation of the training/test. We do not dwell in these steps except for the normalization of the dataset that already has been mentioned that is mandatory using Chebyshev polynomials of any kind as the kernel function.

There are some online data stores available for public use, such a widely used datastore is the *UCI Machine Learning Repository*² of the *University of California, Irvine*, and also *Kaggle*³. In this section, four datasets are selected from UCI, which are well known and widely used for machine learning practitioners.

The polynomial kernel is widely used in kernel-based models and in general, is defined as:

$$K(X_1, X_2) = (a + X_1^T X_2)^b. \quad (3.57)$$

Also, *RBF kernel* a.k.a *guassian RBF kernel*, is another popular kernel for *SVM*, and is defined as:

$$K(X_1, X_2) = \exp\left(-\frac{\|x - x'\|^2}{2\sigma^2}\right). \quad (3.58)$$

3.4.1 Spiral dataset

The Spiral dataset is a famous classic dataset, but there are many datasets that have a spiraling distribution of data points such as the one that is considered here; however, there are simulated ones with similar characteristics and even better controlled distributional properties. The Spiral dataset is consisted of 1000 data points, equally clustered into 3 numerical labels with two other attributes "Chem 1" and "Chem2" of float type. It is known that *SVM* is a binary classification algorithm; therefore, to deal with multi-class datasets the common method is to split such datasets into multiple binary class datasets. Two examples of such methods are:

- One-vs-All (OVA),
- One-vs-One (OVO).

Using the **OVA**, Spiral classification task with numerical classes $\in \{1, 2, 3\}$ will be divided into 3 classifications of the following forms:

- Binary Classification task 1: class 1 vs class {2, 3},

² <https://archive.ics.uci.edu/ml/datasets.php>

³ <https://www.kaggle.com/datasets>

- Binary Classification task 2: class 2 vs class {1, 3},
- Binary Classification task 3: class 3 vs class {1, 2}.

Also, using the **OVO** method, the classification task is divided into any possible binary classification; then, for the Spiral dataset we have:

- Binary Classification task 1: class 1 vs class 2,
- Binary Classification task 2: class 1 vs class 3,
- Binary Classification task 3: class 2 vs class 3.

However, in our example, final number of split datasets are equally in both methods but generally, **OVO** method generates more datasets to classify. Assume a multi-class dataset with 4 numerical labels {1, 2, 3, 4}; afterward, using the first method binary classification datasets is:

- Binary Classification task 1: class 1 vs class {2, 3, 4},
- Binary Classification task 2: class 2 vs class {1, 3, 4},
- Binary Classification task 3: class 3 vs class {1, 2, 4},
- Binary Classification task 4: class 4 vs class {1, 2, 3}.

While using the second method, we have the following binary datasets to classify:

- Binary Classification task 1: class 1 vs class 2,
- Binary Classification task 2: class 1 vs class 3,
- Binary Classification task 3: class 1 vs class 4,
- Binary Classification task 4: class 2 vs class 3,
- Binary Classification task 5: class 2 vs class 4,
- Binary Classification task 6: class 3 vs class 4.

Generally, the **OVA** method is often preferred due to computational and time considerations.

The following figures show how the transformation to fractional order affects the original dataset's distribution, and how the Chebyshev kernel chooses decision boundaries in different orders. Fig. 3.5 depicts the transformed Spiral dataset in its original state and fractional form where $0.1 \leq \alpha \leq 0.9$. As fractional-order gets lower and closer to 0.1, transformation, moves more data points from the original dataset to the upright side of the space, into the positive quarter of the new space in precise, where $x_1 > 0$ and $x_2 > 0$. Although it seems that data points are conglomerated on one corner of the axis on a 2D plot, this is not necessarily what happens in 3D space when the kernel function is applied. In other words, when a kernel function such as the Chebyshev kernel function is applied, data points may spread over a new axis while concentrating on a 2D axis and making it possible to classify those points considering the higher dimension instead. This kind of transformation combined with a kernel function creates a new dimension for the original dataset and makes it possible for the kernel function's decision boundary to capture and classify the data points with a better chance of accuracy.

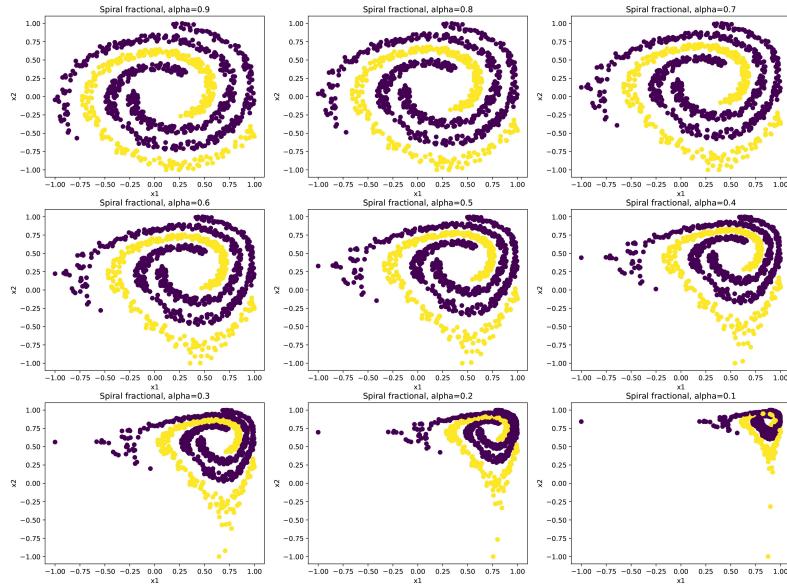


Fig. 3.5: Spiral dataset, different fractional order

In order to have a better intuition of how a kernel function takes the dataset to a higher dimension see Fig. 3.6, which shows the Chebyshev kernel function of the first kind (both normal and fractional form) applied to the Spiral dataset. The figure on the left depicts the dataset in a higher dimension when the normal Chebyshev kernel of order 3 is applied, and the one at right demonstrates how the data points are changed when the fractional Chebyshev kernel of order 3 and alpha 0.3 is applied to the original dataset. It is clear that the transformation has moved the data points to the positive side of the axis x, y, and also z.

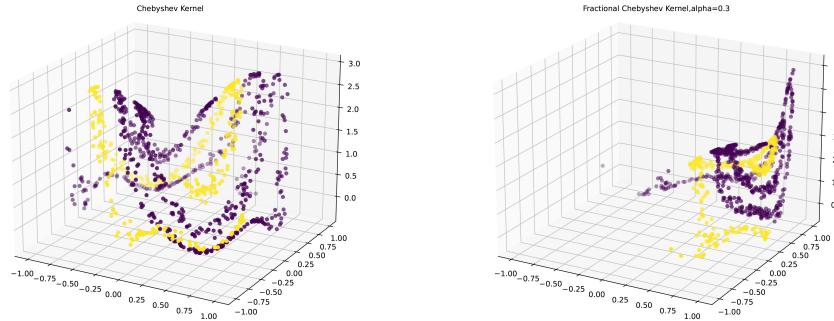


Fig. 3.6: Normal and fractional Chebyshev kernel of order 3 and alpha=0.3 applied on Spiral dataset

After the transformation of the dataset and taking the data points to a higher dimension, it is time to determine the decision boundaries of the classifier⁴. The decision boundary is highly dependent on the kernel function. In the case of orthogonal polynomials, the order of these functions have a critical role. For example, the Chebyshev kernel function with order 3 outcomes a different decision boundary in comparison to the same kernel function with the order of 6. There is no general rule to know which order outcomes the most suitable decision boundary. Hence, trying different decision boundaries gives a useful metric to compare and find the best order or even the kernel function itself.

In order to have an intuition of how the decision boundaries are different, take a look at Fig. 3.7 which depicts corresponding Chebyshev classifiers of different orders {3, 4, 5, 6} on the original Spiral dataset where the binary classification of 1-vs-{2, 3} is chosen. From these plots, one can see that decision boundaries are getting more twisted and twirled as we approach higher orders. In order 6 (the last subplot on the right) the decision boundary is slightly more complex in comparison with order 3 (the first subplot on the right). The complexity of the decision boundary even gets worse in fractional space. Fig. 3.8 demonstrates the decision boundary of the fractional Chebyshev classifier with different orders of 3,4,5 and 6 where $\alpha = 0.3$. Again there is a complicated decision boundary as one approaches from order 3 to order 6.

The purpose is not to specify which one has a better or worse accuracy or final result, it is all about which one is most suitable considering to dataset's distribution. Each decision boundary (or decision surface) classifies the data points in a specific form. As the decision boundary/surface of a kernel function of each order is fixed and determined, it is the degree of correlation between that specific shape and the data points that determine the final accuracy.

⁴ Please note that “decision boundary” refers to the 2D space and the “decision surface” refers to the 3D space.

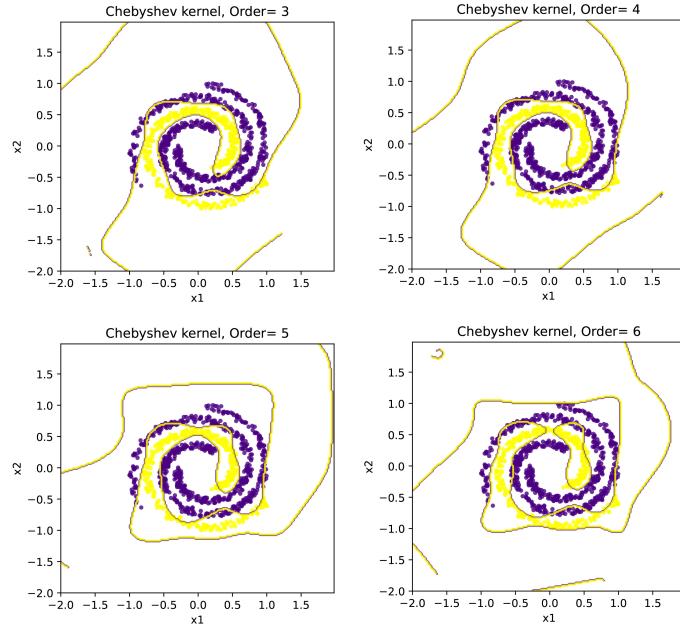


Fig. 3.7: Chebyshev kernel with orders of 3, 4, 5, and 6 on Spiral dataset

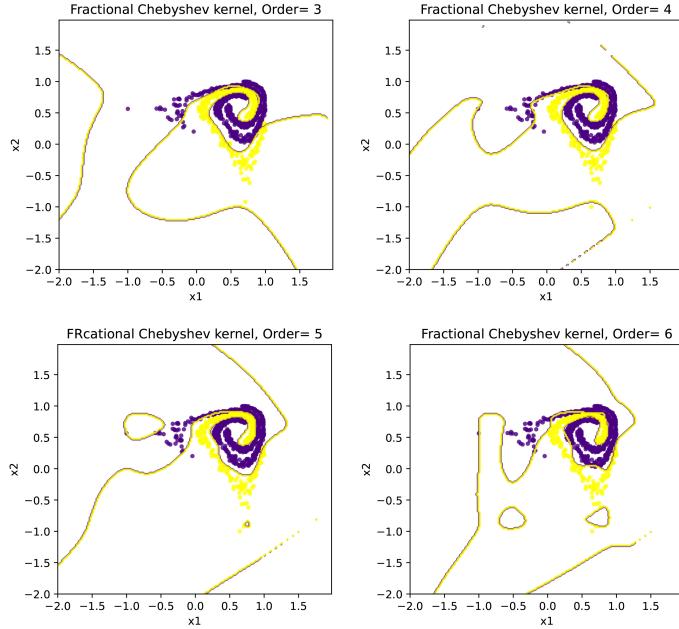


Fig. 3.8: Fractional Chebyshev kernel with orders of 3, 4, 5, and 6 on Spiral dataset ($\alpha = 0.3$)

Finally, the following tables provide a comparison of the experiments on the Spiral dataset. All decision boundaries depicted in previous figures are applied on the dataset and the accuracy score is adopted as the final metric, by the way, three possible binary classifications have been examined according to the One-vs-All method. As it is clear from Table 3.1, RBF kernel and the fractional Chebyshev kernel show better performance for the class 1-vs-{2, 3}. However, for other binary classification classifications, Tables 3.2 and 3.3, RBF and polynomials kernels outperform both normal and fractional kinds of Chebyshev kernels. It is worth mentioning that the accuracy scores are not definitely the best outcomes of those kernels, cause the accuracy of classification with SVM heavily depends on multiple parameters such as the regularization parameter C , *random state*. Here we are not intended to gain the best outcome rather we consider a comparison over similar conditions.

Table 3.1: Comparison of RBF, Polynomial, Chebyshev, and fractional Chebyshev kernels on Spiral dataset. It is clear that the RBF kernel outperforms other kernels

| | Sigma | Power | Order | Alpha(α) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-----------------|
| RBF | 0.73 | - | - | - | 0.97 |
| Polynomial | - | 8 | - | - | 0.9533 |
| Chebyshev | - | - | 5 | - | 0.9667 |
| Fractional Chebyshev | - | - | 3 | 0.3 | 0.9733 |

Table 3.2: Comparison of RBF, Polynomial, Chebyshev, and fractional Chebyshev kernels on Spiral dataset. It is clear that RBF kernel outperforms other kernels

| | Sigma | Power | Order | Alpha(α) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-----------------|
| RBF | 0.1 | - | - | - | 0.9867 |
| Polynomial | - | 5 | - | - | 0.9044 |
| Chebyshev | - | - | 8 | - | 0.9411 |
| Fractional Chebyshev | - | - | 8 | 0.5 | 0.9467 |

Table 3.3: Comparison of RBF, Polynomial, Chebyshev, and fractional Chebyshev kernels on Spiral dataset. It is clear that RBF and Polynomials kernels are better than others

| | Sigma | Power | Order | Alpha(α) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-----------------|
| RBF | 0.73 | - | - | - | 0.9856 |
| Polynomial | - | 5 | - | - | 0.9856 |
| Chebyshev | - | - | 6 | - | 0.9622 |
| Fractional Chebyshev | - | - | 6 | 0.6 | 0.9578 |

3.4.2 Three Monks's dataset

The Monk's problem is a classic one introduced in 1991, according to *Monk's Problems - A Performance Comparison of Different Learning algorithms*" by S.B. Thrun et al [56, 46]:

The Monk's problems rely on an artificial robot domain in which robots are described by six different attributes:

| | |
|---------|--|
| $x_1 :$ | <i>head shape</i> $\in \{\text{round}, \text{square}, \text{octagon}\}$ |
| $x_2 :$ | <i>body shape</i> $\in \{\text{round}, \text{square}, \text{octagon}\}$ |
| $x_3 :$ | <i>is smiling</i> $\in \{\text{yes}, \text{no}\}$ |
| $x_4 :$ | <i>holding</i> $\in \{\text{sword}, \text{balloon}, \text{flag}\}$ |
| $x_5 :$ | <i>jacket color</i> $\in \{\text{red}, \text{yellow}, \text{green}, \text{blue}\}$ |
| $x_6 :$ | <i>has tie</i> $\in \{\text{yes}, \text{no}\}$ |

The learning task is a binary classification one. Each problem is given by a logical description of a class. Robots belong either to this class or not, but instead of providing a complete class description to the learning problem, only a subset of all 432 possible robots with its classification is given. The learning task is to generalize over these examples and if the particular learning technique at hand allows this to derive a simple class description.

Then the three problems are:

- **Problem M_1 :** (**head shape = body shape**) or (**jacket color = red**) From 432 possible examples, 124 are randomly selected for the training set. There are no misclassifications.
- **Problem M_2 :** **Exactly two of the six attributes have their first value.** (E.g.: body shape = head shape = round implies that the robot is not smiling, holding no sword, jacket color is not red and has no tie, since then exactly two (body shape and head shape) attributes have their first value) From 432 possible examples, 169 are randomly selected. Again, there is no noise.
- **Problem M_3 :** (**jacket color is green and holding a sword**) or (**jacket color is not blue and body shape is not octagon**) From 432 examples, 122 are selected randomly, and among them, there are 5% misclassifications, i.e. noise in the training set.

Considering the above definitions, we applied *RBF*, *Polynomial*, *Chebyshev*, and *Fractional Chebyshev* kernels to all three Monk's Problems. Table 3.4 illustrates the output accuracy of each model where the *RBF kernel* has the best accuracy by $\sigma \approx 2.84$ at **0.8819** and the *Chebyshev kernel* has the worst among them at **0.8472**. Table 3.5 shows the results for Monk's second problem, where *fractional Chebyshev Kernel* has the best accuracy at **0.9653** while *Chebyshev Kernel* has the worst performance with an accuracy of **0.8426**. Finally, Table 3.6, represents the results for the third Monk's problem where both *RBF* and *fractional Chebyshev kernel* has the same best accuracy at **0.91**.

Table 3.4: Comparison of RBF, Polynomial, Chebyshev, and fractional Chebyshev kernels on the Monk's first problem. It is clear that the RBF kernel outperforms other kernels

| | Sigma | Power | Order | Alpha(α) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-----------------|
| RBF | 2.844 | - | - | - | 0.8819 |
| Polynomial | - | 3 | - | - | 0.8681 |
| Chebyshev | - | - | 3 | - | 0.8472 |
| Fractional Chebyshev | - | - | 3 | 1/16 | 0.8588 |

Table 3.5: Comparison of RBF, Polynomial, Chebyshev, and fractional Chebyshev kernels on the Monk's second problem. fractional Chebyshev Kernel outperforms other kernel with highest accuracy at 96 %

| | Sigma | Power | Order | Alpha(α) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-----------------|
| RBF | 5.5896 | - | - | - | 0.875 |
| Polynomial | - | 3 | - | - | 0.8657 |
| Chebyshev | - | - | 3 | - | 0.8426 |
| Fractional Chebyshev | - | - | 6 | 1/15 | 0.9653 |

Table 3.6: Comparison of RBF, Polynomial, Chebyshev, and fractional Chebyshev kernels on the Monk's third problem. Fractional Chebyshev Kenrel and RBF kernel had same accuracy result at 0.91

| | Sigma | Power | Order | Alpha(α) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-----------------|
| RBF | 2.1586 | - | - | - | 0.91 |
| Polynomial | - | 3 | - | - | 0.875 |
| Chebyshev | - | - | 6 | - | 0.8958 |
| Fractional Chebyshev | - | - | 5 | 1/5 | 0.91 |

3.5 Conclusion

This chapter is started with a brief history and basics of Chebyshev orthogonal polynomials. Chebyshev polynomials have been used in many use cases and recently as the kernel function in kernel-based learning algorithms and are proved to surpass traditional kernels in many situations. Construction of Chebyshev kernels explained and proved. It is demonstrated in this chapter that the fractional form of Chebyshev polynomials extends the applicability of the Chebyshev kernel. Thus, by using fractional Chebyshev kernel functions, a wider set of problems can be tackled. Experiments demonstrate fractional Chebyshev kernel leverages the accuracy of classification with SVM when being used as the kernel in the SVM kernel trick. In the last section, the results of such experiments are depicted.

References

1. Shuman, D. I., Vandergheynst, P., Kressner, D., Frossard, P.: Distributed signal processing via Chebyshev polynomial approximation. *IEEE Trans. Signal Inf. Process. Netw.* **4**, 736–751 (2018).
2. Pavlović, V. D., Dončov, N. S., Ćirić, D. G.: 1D and 2D economical FIR filters generated by Chebyshev polynomials of the first kind. *Int. J. Electron.* **100**, 1592–1619 (2013).
3. Sedaghat, S., Ordokhani, Y., Dehghan, M.: Numerical solution of the delay differential equations of pantograph type via Chebyshev polynomials. *Commun Nonlinear Sci Numer Simul* **17**, 4815–4830 (2012).
4. Zhao, F., Huang, Q., Xie, J., Li, Y., Ma, L., Wang, J.: Chebyshev polynomials approach for numerically solving system of two-dimensional fractional PDEs and convergence analysis. *Appl. Math. Comput* **313**, 321–330 (2017).
5. Shaban, M., Kazem, S., Rad, J. A.: A modification of the homotopy analysis method based on Chebyshev operational matrices. *Mathematical and Computer Modelling* **57**, 1227–1239 (2013).
6. Kazem, S., Shaban, M., Rad, J. A.: Solution of the coupled Burgers equation based on operational matrices of d-dimensional orthogonal functions. *Zeitschrift für Naturforschung A* **67** 267–274 (2012).
7. Parand, K., Moayeri, M. M., Latifi, S., Rad, J. A.: Numerical study of a multidimensional dynamic quantum model arising in cognitive psychology especially in decision making. *The European Physical Journal Plus* **134** 109 (2019)
8. Hadian-Rasanan, A. H., Rad, J. A.: Brain Activity Reconstruction by Finding a Source Parameter in an Inverse Problem. Chakraverty, S. (ed.) *Mathematical Methods in Interdisciplinary Sciences*, pp. 343–368. John Wiley & Sons, Amsterdam (2020)
9. Kazem, S., Shaban, M., Rad, J. A.: A new Tau homotopy analysis method for MHD squeezing flow of second-grade fluid between two parallel disks. *Applied and Computational Mathematics* **16** 114–132 (2017)
10. Ye, N., Sun, R., Liu, Y., Cao, L.: Support vector machine with orthogonal Chebyshev kernel. In 18th International Conference on Pattern Recognition (ICPR'06) **2** 752–755 (2006)
11. Ozer, S., Chen, C. H., Cirpan, H. A.: A set of new Chebyshev kernel functions for support vector machine pattern classification. *Pattern Recognit* **44**, 1435–1447 (2011)
12. Vert, J. P., Qiu, J., Noble, W. S.: A new pairwise kernel for biological network inference with support vector machines. In *BMC bioinformatics BioMed Central* **8**, 1–10 (2007)

13. Asghari, M., Hadian Rasanan, A.H., Gorgin, S., Rahmati, D., Parand, K.: FPGA-orthopoly: a hardware implementation of orthogonal polynomials. *Eng Comput* (2022) doi: 10.1007/s00366-022-01612-x
14. Achirul Nanda, M., Boro Seminar, K., Nandika, D., Maddu, A.: A comparison study of kernel functions in the support vector machine and its application for termite detection. *Information*, **9**, 5–29 (2018)
15. Hussain, M., Wajid, S. K., Elzaart, A., Berbar, M.: A comparison of SVM kernel functions for breast cancer detection. In 2011 eighth international conference computer graphics, imaging and visualization, 145–150 (2011)
16. An-na, W., Yue, Z., Yun-tao, H., Yun-lu, L. I.: A novel construction of SVM compound kernel function. In 2010 International conference on logistics systems and intelligent management (ICLSIM) **3**, 1462–1465 (2010)
17. Kazem, S., Abbasbandy, S., Kumar, S.: Fractional-order Legendre functions for solving fractional-order differential equations. *Appl. Math. Model* **37**, 5498–5510 (2013)
18. Musavi, M. T., Ahmed, W., Chan, K. H., Faris, K. B., Hummels, D. M.: On the training of radial basis function classifiers. *Neural Netw* **5**, 595–603 (1992)
19. Scholkopf, B., Sung, K. K., Burges, C. J., Girosi, F., Niyogi, P., Poggio, T., Vapnik, V.: Comparing support vector machines with Gaussian kernels to radial basis function classifiers. *IEEE Trans. Signal Process* **45**, 2758–2765 (1997)
20. Parand, K., Delkhosh, M.: Solving Volterra's population growth model of arbitrary order using the generalized fractional order of the Chebyshev functions. *Ric. Mat.* **65**, 307–328 (2016)
21. Jafarzadeh, S. Z., Aminian, M., Efati, S.: A set of new kernel function for support vector machines: An approach based on Chebyshev polynomials. In ICCKE 412–416 (2013)
22. Zhao, J., Yan, G., Feng, B., Mao, W., Bai, J.: An adaptive support vector regression based on a new sequence of unified orthogonal polynomials. *Pattern Recognit* **46**, 899–913 (2013)
23. Mason, J. C., Handscomb, D. C.: Chebyshev polynomials. CRC press, Florida (2002)
24. Jung, H. G., Kim, G.: Support vector number reduction: Survey and experimental evaluations. *IEEE trans Intell Transp Syst* **15**, 463–476 (2013)
25. Vapnik, V.: The nature of statistical learning theory. Springer, Berlin (2013)
26. Zhou, F., Fang, Z., Xu, J.: Constructing support vector machine kernels from orthogonal polynomials for face and speaker verification. In Fourth International Conference on Image and Graphics (ICIG), 627–632 (2007).
27. Ozer, S., Chen, C. H.: Generalized Chebyshev kernels for support vector classification. In 19th International Conference on Pattern Recognition, 1–4 (2008)
28. Schölkopf, B., Smola, A. J., Bach, F.: Learning with kernels: support vector machines, regularization, optimization, and beyond. MIT press, Cambridge (2002)
29. Omidi, M., Arab, B., Hadian Rasanan, A. H., Rad, J. A., Parand, K.: Learning nonlinear dynamics with behavior ordinary/partial/system of the differential equations: looking through the lens of orthogonal neural networks. *Eng Comput* , 1–20 (2021)
30. Hadian Rasanan, A. H., Rahmati, D., Gorgin, S., Rad, J. A.: MCILS: Monte-Carlo Interpolation Least-Square Algorithm for Approximation of Edge-Reliability Polynomial. In 9th International Conference on Computer and Knowledge Engineering (ICCKE) 295–299 (2019)
31. Boyd, J. P.: Chebyshev and Fourier spectral methods. Courier Corporation, Massachusetts (2001)
32. Parand, K., Delkhosh, M.: Accurate solution of the Thomas–Fermi equation using the fractional order of rational Chebyshev functions. *J. Comput. Appl. Math.* **317**, 624–642 (2017)
33. Parand, K., Moayeri, M. M., Latifi, S., Delkhosh, M.: A numerical investigation of the boundary layer flow of an Eyring–Powell fluid over a stretching sheet via rational Chebyshev functions. *Eur. Phys. J* **132**, 1–11 (2017)
34. Mall, S., Chakraverty, S.: A novel Chebyshev neural network approach for solving singular arbitrary order Lane–Emden equation arising in astrophysics. *NETWORK-COMP NEURAL* **31**, 142–165 (2020)
35. Chakraverty, S., Mall, S.: Single layer Chebyshev neural network model with regression-based weights for solving nonlinear ordinary differential equations. *Evol. Intell.* **13**, 687–694(2020)

36. Mall, S., Chakraverty, S.: Single layer Chebyshev neural network model for solving elliptic partial differential equations. *Neural Process. Lett.* **45**, 825–840 (2017)
37. Mall, S., Chakraverty, S.: Numerical solution of nonlinear singular initial value problems of Emden—Fowler type using Chebyshev Neural Network method. *Neurocomputing* **149**, 975–982 (2015)
38. Capozziello, S., D’Agostino, R., Luongo, O.: Cosmographic analysis with Chebyshev polynomials. *MNRAS* **476**, 3924–3938 (2018)
39. Hassani, H., Machado, J. T., Naraghkad, E.: Generalized shifted Chebyshev polynomials for fractional optimal control problems. *Commun Nonlinear Sci Numer Simul.* **75**, 50–61 (2019)
40. Glau, K., Mahlstedt, M., Pötz, C.: A new approach for American option pricing: The Dynamic Chebyshev method. *SIAM J Sci Comput* **41**, B153–B180 (2019)
41. Mesgarani, H., Beiranvand, A., Aghdam, Y. E.: The impact of the Chebyshev collocation method on solutions of the time-fractional Black–Scholes. *Math. Sci.* **15**, 137–143 (2021)
42. Dabiri, A., Butcher, E. A. Nazari, M.: Coefficient of restitution in fractional viscoelastic compliant impacts using fractional Chebyshev collocation. *J. Sound Vib.* **388**, 230–244 (2017)
43. Kheyrinataj, F., Nazemi, A.: Fractional Chebyshev functional link neural network-optimization method for solving delay fractional optimal control problems with Atangana-Baleanu derivative. *Optim Control Appl Methods* **41**, 808–832 (2020)
44. Habibli, M., Noori Skandari, M. H.: Fractional Chebyshev pseudospectral method for fractional optimal control problems. *Optim Control Appl Methods* **40**, 558–572 (2019)
45. Hadian Rasanan, A. H., Rahmati, D., Gorgin, S., Parand, K.: A single layer fractional orthogonal neural network for solving various types of Lane–Emden equation. *New Astron.* **75**, 101307 (2020)
46. Sun, L., Toh, K.A., Lin, Z.: A center sliding Bayesian binary classifier adopting orthogonal polynomials. *Pattern Recognit* **48**, 2013–2028 (2015)
47. Padierna, L. C., Carpio, M., Rojas-Domínguez, A., Puga, H., Fraire, H.: A novel formulation of orthogonal polynomial kernel functions for SVM classifiers: the Gegenbauer family. *Pattern Recognit* **84**, 211–225 (2018)
48. Moghaddam, V. H., Hamidzadeh, J.: New Hermite orthogonal polynomial kernel and combined kernels in support vector machine classifier. *Pattern Recognit* **60**, 921–935 (2016)
49. Tian, M., Wang, W.: Some sets of orthogonal polynomial kernel functions. *Appl. Soft Comput.* **61**, 742–756 (2017)
50. Mason, J. C., Handscomb, D. C.: Chebyshev polynomials. Chapman and Hall/CRC (2002)
51. Boyd, J. P.: Chebyshev and Fourier spectral methods. Courier Corporation (2001)
52. Shen, J., Tang, T., Wang, L. L.: Spectral methods: algorithms, analysis and applications (Vol. 41). Springer Science & Business Media, Berlin (2011)
53. Hajimohammadi, Z., Baharifard, F., Ghodsi, A., Parand, K.: Fractional Chebyshev deep neural network (FCDNN) for solving differential models. *Chaos Solitons Fractals* **153**, 111530 (2021)
54. Reddy, S. V. G., Reddy, K. T., Kumari, V. V., Varma, K. V.: An SVM based approach to breast cancer classification using RBF and polynomial kernel functions with varying arguments. *IJCSIT* **5**, 5901–5904 (2014)
55. Yaman, S., Pelecanos, J.: Using polynomial kernel support vector machines for speaker verification. *IEEE Signal Process. Lett.* **20**, 901–904 (2013)
56. Thrun, S.B., Bala, J.W., Bloedorn, E., Bratko, I., Cestnik, B., Cheng, J., De Jong, K.A., Dzeroski, S., Fisher, D.H., Fahlman, S.E. Hamann, R.: The monk’s problems: A performance comparison of different learning algorithms (1991)
57. Pan, Z. B., Chen, H., You, X. H.: Support vector machine with orthogonal Legendre kernel. In 2012 International Conference on Wavelet Analysis and Pattern Recognition, 125–130 (2012)

Chapter 4

Fractional Legendre Kernel Functions: Theory and Application

Amirreza Azmoon and Snehashish Chakraverty and Sunil Kumar

Abstract The support vector machine algorithm has been able to show great flexibility in solving many machine learning problems due to the use of different functions as a kernel. Linear, radial basis functions, and polynomial functions are the most common functions used in this algorithm. Legendre polynomials are among the most widely used orthogonal polynomials that have achieved excellent results in the support vector machine algorithm. In this chapter, some basic features of Legendre and fractional Legendre functions are introduced and reviewed, and then the kernels of these functions are introduced and validated. Finally, the performance of these functions in solving two problems (two sample datasets) is measured.

4.1 Introduction

Another family of classic complete and orthogonal polynomials is ***Legendre Polynomials*** which was discovered by French mathematician ***Adrien-Marie Legendre*** (1752-1833) in 1782.

Amirreza Azmoon

Department of Computer Science, The Institute for Advance Studies in Basic Sciences (IASBS),
Zanjan, Iran, e-mail: a.azmoon@iasbs.ac.ir

Snehashish Chakraverty

Department of Mathematics, National Institute of Technology Rourkela, Sundargarh, Odisha, India,
e-mail: sne_chak@yahoo.com

Sunil Kumar

Department of Mathematics, National Institute of Technology, Jamshedpur 831014, Jharkhand,
India e-mail: skumar.math@nitjsr.ac.in

Adrien-Marie Legendre (1752-1833) is a French mathematician who had numerous contributions to mathematics. Legendre got his education from Collège Mazarin of Paris and defended his thesis at the age of 18. In 1782 he won a prize from the Berlin Academy for an essay titled "Research on the trajectories of projectiles in a resistant medium", which brought significant attention to this young mathematician. During his career, many papers have been published on elliptic functions, number theory, and method of least squares. In a publication titled "Research on the shape of planets", he introduced Legendre polynomials. In honor of Legendre, a crater on the moon is named after him.^a

^a For more information about **A.M. Legendre** and his contribution, please see: <https://mathshistory.st-andrews.ac.uk/Biographies/Legendre/>.

Like other functions in the family of orthogonal functions, Legendre polynomials are used in various fields of science and industry, such as topography filtering [1], solving differential equations in the crystallization process in chemistry [2], expansions of hypergeometric functions [3, 4], calculating scattering of charged particles in physics [5, 6], financial market prediction models [7, 8], and modeling epileptic seizure [9]. Other important and common applications of Legendre functions in the industry include analysis of time-varying linear control systems [10], model reduction in control systems [11], estimation algorithms in PMU systems [12] in the field of power, wave motion in propagation [13, 16, 17] and reflection [18]. These examples mentioned are some of the fields in which orthogonal functions, especially Legendre functions, have been used extensively both to improve the scientific level of projects and to improve industrial efficiency. One of the most common and attractive applications of Legendre functions is to solve differential equations with different orders with the help of Legendre wavelet functions [19, 20] and shifted Legendre function [21, 22]. In particular, Mall and Chakraverty [23] designed neural networks using Legendre functions to solve ordinary differential equations. This method based on a single-layer Legendre neural network model has been developed to solve initial and boundary value problems. In the proposed approach, a Legendre polynomial-based functional link artificial neural network is developed. In fact, the hidden layer is eliminated by expanding the input pattern using Legendre polynomials. On the other hand, due to the diversity of different kernels, support vector machine algorithms [24] are one of the most frequent and varied machine learning methods. In a recent study on this subject [25], the usage of orthogonal functions as kernels has expanded substantially, so that these functions are now utilized as fractional, wavelet, or shifting order kernels in addition to regular orthogonal functions [25, 26, 27].

Table 4.1: Some applications for different kinds of Legendre polynomials

| | |
|-------------------------------|--|
| Legendre polynomials | Legendre polynomials are used frequently in different fields of science. For example, they are used as an orthogonal kernel in support vector machines [26], or as basis functions for solving various differential equations such as fractional equations [28], integral equations [29], optimal control problems [30] and so on [31]. Moreover, it has been used in long short term memory neural networks [48]. |
| Fractional Legendre functions | Based on fractional-order Legendre polynomials, Benouini et al. proposed a new set of moments that can be used to extract pattern features [24, 27]. These functions have many other applications, for instance, solving fractional differential equations [32, 34]. |
| Rational Legendre functions | They are used to solve differential equations in semi-infinite intervals [36, 37, 38]. |

The following is how this chapter is arranged. Section 4.2 presents the fundamental definitions and characteristics of orthogonal Legendre polynomials and fractional Legendre functions. In Section 4.3, the ordinary Legendre kernel is provided, and the innovative fractional Legendre kernel is introduced. In Section 4.4, The results of experiments on both the ordinary Legendre kernel and the fractional covered and give a comparison of the accuracy results of the mentioned kernels used as a kernel in the SVM algorithm with the normal polynomial and Gaussian kernels and ordinary Chebyshev and fractional Chebyshev on well-known datasets to demonstrate the validity and efficiency of kernels. Finally, in Section 4.5, the concluding remarks of this chapter are presented.

4.2 Preliminaries

Definition and basic properties of Legendre orthogonal polynomials are presented in this section. In addition to the basics of these polynomials, the fractional form of this family is discussed.

4.2.1 Properties of Legendre polynomials

Legendre polynomials of degree n , $P_n(x)$, are solutions to the following Sturm-Liouville differential equation [47, 32, 42, 35, 33]:

$$(1-x^2) \frac{d^2y}{dx^2} - 2x \frac{dy}{dx} + n(n+1)y = 0, \quad (4.1)$$

where n is a positive integer. This family is also orthogonal over the interval $[-1, 1]$ with respect to the weight function $w(x) = 1$ [39], such that

$$\int_{-1}^1 P_n(x)P_m(x) dx = \frac{2}{2n+1} \delta_{nm}, \quad (4.2)$$

in which δ_{nm} is the Kronecker delta and is defined as follows:

$$\delta_{mn} = \begin{cases} 1, & n = m, \\ 0, & \text{Otherwise.} \end{cases} \quad (4.3)$$

Besides the Sturm-Liouville differential equation, there is a generating function for these polynomials which can generate each order of the Legendre polynomials. By setting $|t| < 1$ and $|x| \leq 1$, the generating function of the Legendre polynomials as follows [47, 32, 42, 35, 33]:

$$\frac{1}{\sqrt{1-2tx+t^2}} = \sum_{n=0}^{\infty} t^n P_n(x). \quad (4.4)$$

Thus, the coefficient of t^n is a polynomial in x of degree n , which shows the n -th Legendre polynomial. Expanding this generating function up to t^1 yields:

$$P_0(x) = 1, P_1(x) = x. \quad (4.5)$$

Similar to the Chebyshev polynomials, the Legendre polynomials can be obtained by utilizing a recursive formula which is as follows [40, 47, 32, 42, 35, 33]:

$$\begin{aligned} P_0(x) &= 1, \\ P_1(x) &= x, \\ (n+1)P_{n+1}(x) - (2n+1)xP_n(x) + nP_{n-1}(x) &= 0, \quad n \geq 1. \end{aligned} \quad (4.6)$$

So, the first few Legendre polynomials in the explicit form are:

$$\begin{aligned}
P_0(x) &= 1, \\
P_1(x) &= x, \\
P_2(x) &= (3x^2 - 1)/2, \\
P_3(x) &= (5x^3 - 3x)/2, \\
P_4(x) &= (35x^4 - 30x^2 + 3)/8, \\
P_5(x) &= (63x^5 - 70x^3 + 15x)/8, \\
P_6(x) &= (231x^6 - 315x^4 + 105x^2 - 5)/16,
\end{aligned} \tag{4.7}$$

which are depicted in Fig. 4.1.

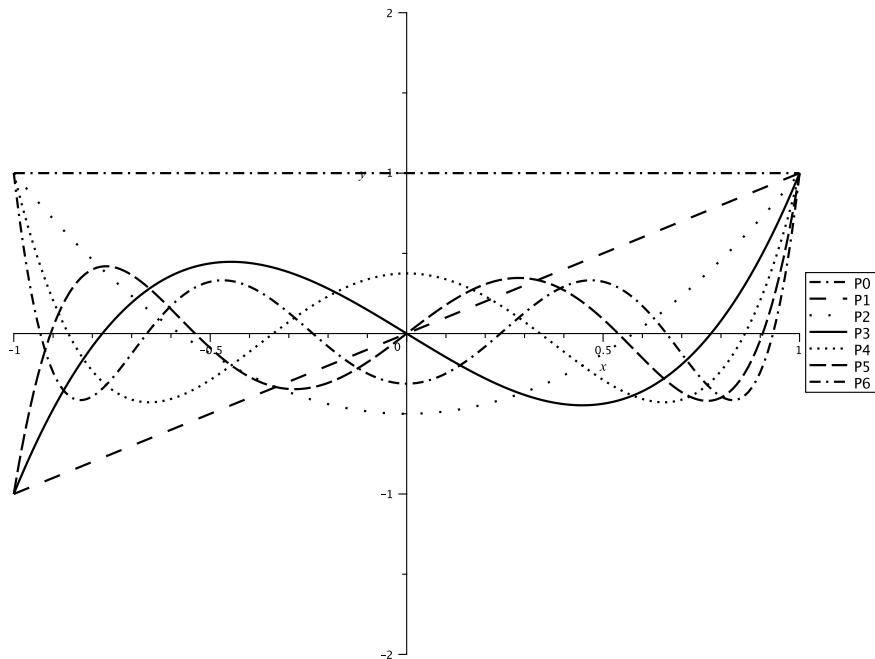


Fig. 4.1: The first six orders of Legendre Polynomials

In addition to the recursive relation in Eq. 4.6, using the generator function Eq. 4.4, the Legendre polynomials are defined in the following two ways as recursive relations.

$$\begin{aligned}
nP_n(x) &= xP'_n(x) - P'_{n-1}(x), \\
P'_{n+1}(x) &= xP'_n(x) + (n+1)P_n(x).
\end{aligned} \tag{4.8}$$

Also, the following recursive relations, by combining the above relations and their derivatives with each other, create other forms of recursive relations of Legendre

polynomials [41]:

$$\begin{aligned} (2n+1)P_n(x) &= P'_{n+1}(x) - P'_{n-1}(x), \\ (1-x^2)P'_n(x) &= n[P_{n-1}(x) - xP_n(x)]. \end{aligned} \quad (4.9)$$

The reader can use the following Python code to generate any order of Legendre Polynomials symbolically:

Program Code

```
import sympy
x = sympy.Symbol("x")

def Ln(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    elif n >= 2:
        return ((2 * n - 1) * (x * Ln(x, n-1)) - (n-1)*Ln(x, n-2))/n

sympy.expand(sympy.simplify(Ln(x, 3)))
```

$$> \frac{5x^3}{2} - \frac{3x}{2}$$

Similar to other classical orthogonal polynomials, Legendre polynomials follow symmetry. Therefore Legendre polynomials of even order have even symmetry and contain only even powers of x and similarly odd orders of Legendre polynomials have odd symmetry and contain only odd powers of x :

$$P_n(x) = (-1)^n P_n(-x) = \begin{cases} P_n(-x), & n \text{ is even}, \\ -P_n(-x), & n \text{ is odd}. \end{cases} \quad (4.10)$$

$$P'_n(x) = (-1)^n P'_n(-x) = \begin{cases} P'_n(-x), & n \text{ is even}, \\ -P'_n(-x), & n \text{ is odd}. \end{cases} \quad (4.11)$$

For $P_n(x)$, $n = 0, 1, \dots, n$, there exist exactly n zero in $[-1, 1]$ that are real, distinct from each other [49]. If they are regarded as dividing the interval $[-1, 1]$ into $n + 1$ sub-intervals, each sub-interval will contain exactly one zero of P_{n+1} [49]. Also, $P_n(x)$ has $n - 1$ local minimum and maximum in $(-1, 1)$ [49]. On the other hand, these polynomials have the following basic properties:

$$P_n(1) = 1, \quad (4.12)$$

$$P_n(-1) = \begin{cases} 1, & n = 2m \\ -1, & n = 2m + 1 \end{cases}, \quad (4.13)$$

$$P_n(0) = \begin{cases} \frac{(-1)^m}{4^m} \binom{2m}{m} = \frac{(-1)^m}{2^{2m}} \frac{(2m)!}{(m!)^2}, & n = 2m \\ 0, & n = 2m + 1 \end{cases}. \quad (4.14)$$

Above all, to shift the Legendre polynomials from $[-1, 1]$ to $[a, b]$ should use the following transformation:

$$x = \frac{2t - a - b}{b - a}, \quad (4.15)$$

where $x \in [-1, 1]$ and $t \in [a, b]$. It is worth mentioning that all the properties of the Legendre polynomials remain unchanged for shifted Legendre polynomials with this difference that the position of -1 transfers to a and the position of 1 transfers to b .

4.2.2 Properties of fractional Legendre functions

Legendre polynomials of fractional order of α (called $FP_n^\alpha(x)$) over finite interval $[a, b]$ by use of mapping $x' = 2(\frac{x-a}{b-a})^\alpha - 1$, ($\alpha > 0$) that $x' \in [-1, 1]$, is defined as [32]:

$$FP_n^\alpha(x) = P_n(x') = P_n\left(2\left(\frac{x-a}{b-a}\right)^\alpha - 1\right). \quad (4.16)$$

Also, generating function for fractional Legendre is defined in the same way as generating function for Legendre, with the difference that in this function x is defined as $(x = 2(\frac{x-a}{b-a})^\alpha - 1)$ in interval $[a, b]$ [47, 32, 42, 35, 33]:

$$\frac{1}{\sqrt{1 - 2t\left(\frac{x-a}{b-a}\right)^\alpha - 1 + t^2}} = \sum_{n=0}^{\infty} FP_n^\alpha(x)t^n = \sum_{n=0}^{\infty} P_n\left(2\left(\frac{x-a}{b-a}\right)^\alpha - 1\right)t^n. \quad (4.17)$$

On the other hand, fractional Legendre function with weight function $w(x) = x^{\alpha-1}$, like Legendre polynomial, has the property of orthogonality [42], this property for example over the interval $[0, 1]$ is easily defined as follows:

$$\int_0^1 FP_n^\alpha(x) FP_m^\alpha(x) dx = \frac{1}{(2n+1)\alpha} \delta_{nm}. \quad (4.18)$$

where δ_{nm} is the Kronecker delta that defined in Eq. 4.3.

It can also be easily shown that for fractional Legendre polynomials, the recursive relation will be as follow [32, 42]:

$$\begin{aligned}
FP_0^\alpha(x) &= 1, \\
FP_1^\alpha(x) &= 2\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\
FP_{n+1}^\alpha(x) &= \left(\frac{2n+1}{n+1}\right)\left(2\left(\frac{x-a}{b-a}\right)^\alpha - 1\right)FP_n^\alpha(x') - \left(\frac{n}{n+1}\right)FP_{n-1}^\alpha(x').
\end{aligned} \tag{4.19}$$

With this in mind, the first few fractional Legendre Polynomials order in the explicit form are :

$$\begin{aligned}
FP_0^\alpha(x) &= 1, \\
FP_1^\alpha(x) &= 2\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\
FP_2^\alpha(x) &= 6\left(\frac{x-a}{b-a}\right)^{2\alpha} - 6\left(\frac{x-a}{b-a}\right)^\alpha + 1, \\
FP_3^\alpha(x) &= 20\left(\frac{x-a}{b-a}\right)^{3\alpha} - 30\left(\frac{x-a}{b-a}\right)^{2\alpha} + 12\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\
FP_4^\alpha(x) &= 70\left(\frac{x-a}{b-a}\right)^{4\alpha} - 140\left(\frac{x-a}{b-a}\right)^{3\alpha} + 90\left(\frac{x-a}{b-a}\right)^{2\alpha} \\
&\quad - 20\left(\frac{x-a}{b-a}\right)^\alpha + 1, \\
FP_5^\alpha(x) &= 252\left(\frac{x-a}{b-a}\right)^{5\alpha} - 630\left(\frac{x-a}{b-a}\right)^{4\alpha} + 560\left(\frac{x-a}{b-a}\right)^{3\alpha} \\
&\quad - 210\left(\frac{x-a}{b-a}\right)^{2\alpha} + 30\left(\frac{x-a}{b-a}\right)^\alpha - 1, \\
FP_6^\alpha(x) &= 924\left(\frac{x-a}{b-a}\right)^{6\alpha} - 2772\left(\frac{x-a}{b-a}\right)^{5\alpha} + 3150\left(\frac{x-a}{b-a}\right)^{4\alpha} \\
&\quad - 1680\left(\frac{x-a}{b-a}\right)^{3\alpha} + 420\left(\frac{x-a}{b-a}\right)^{2\alpha} - 42\left(\frac{x-a}{b-a}\right)^\alpha + 1. \tag{4.20}
\end{aligned}$$

The interested readers can use the following Python code to generate any order of Legendre polynomials of fractional order symbolically

Program Code

```

import sympy
x = sympy.Symbol("x")
alpha = sympy.Symbol(r'\alpha')
a = sympy.Symbol("a")
b = sympy.Symbol("b")
x=sympy.sympify(2*((x-a)/(b-a))**alpha -1)

def FLn(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return x
    elif n >= 2:
        return ((2 * n - 1) * (x * Ln(x,n-1)) - (n-1)*Ln(x,n-2))/n

```

```
sympy.simplify(FLn(x, 3))
```

$$> 20(\frac{x-a}{b-a})^{3\alpha} - 30(\frac{x-a}{b-a})^{2\alpha} + 12(\frac{x-a}{b-a})^\alpha - 1$$

Fig. 4.2 shows the fractional Legendre functions of the first kind up to 6-th order where $a = 0$, $b = 5$, and α is 0.5.

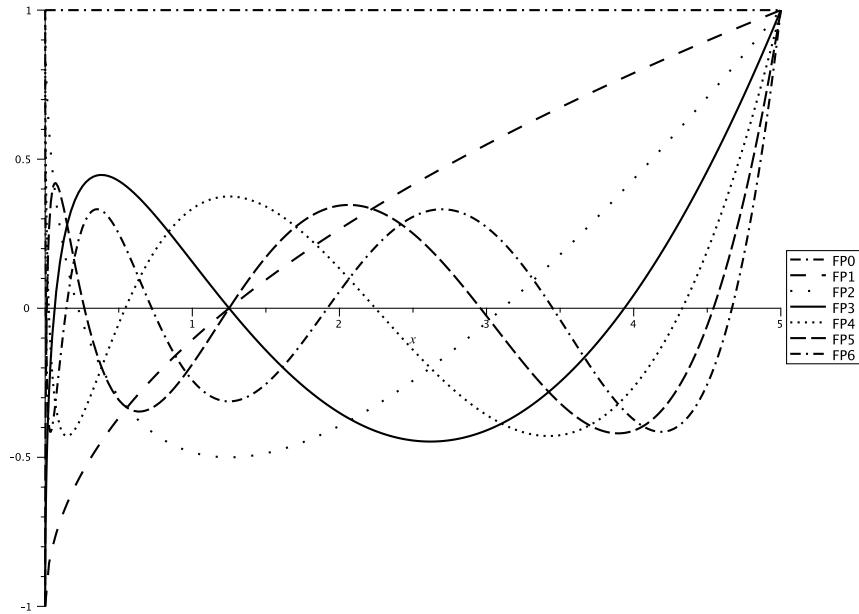


Fig. 4.2: The first six order of fractional Legendre function over the finite interval $[0, 5]$ where $\alpha = 0.5$

Also, Fig. 4.3 depicts the fractional Legendre functions of the first kind of order 5 for different values of α .

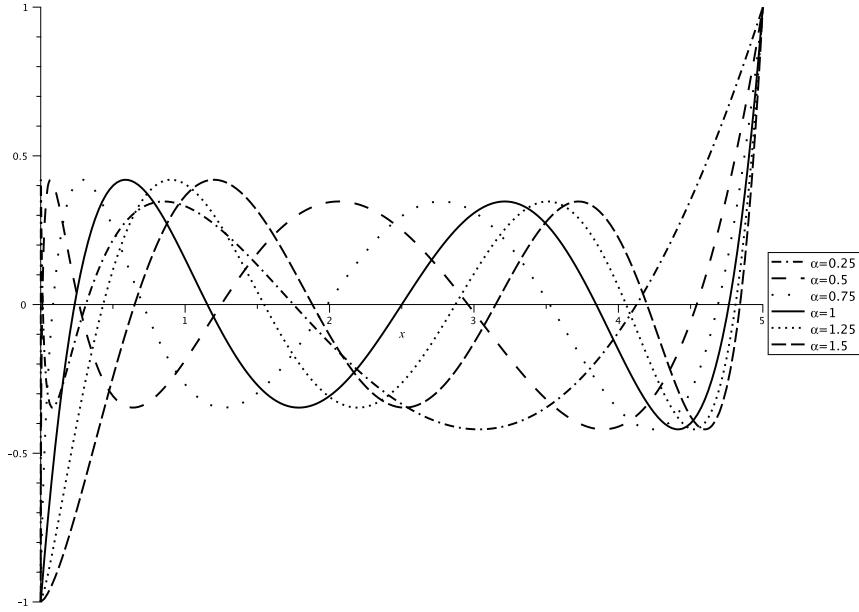


Fig. 4.3: Fifth order of fractional Legendre function over the finite interval $[0, 5]$ with different α

4.3 Legendre kernel functions

In this section, the ordinary Legendre polynomial kernel formula is presented first. Then, some other versions of this kernel function, and at the end, the fractional Legendre polynomial kernel will be introduced in this section.

4.3.1 Ordinary Legendre kernel function

Legendre Polynomials, like other polynomials, are orthogonal except that Legendre polynomials weight function is a constant function of value 1 ($w(x) = 1$). So according to Eq. 4.2 and Eq. 4.3 for different m and n , Legendre polynomials are orthogonal to each other with weight function 1. This enables us to construct the Legendre kernel without denominators. Therefore, for scalar inputs x and z , Legendre kernel is defined as follows

$$K(x, z) = \sum_{i=0}^n P_i(x)P_i(z) = \langle \phi(x), \phi(z) \rangle, \quad (4.21)$$

where $\langle \cdot, \cdot \rangle$ is an inner product and the unique parameter n is the highest order of the Legendre polynomials, and the nonlinear mapping determined by Legendre kernel is :

$$\phi(x) = (P_0(x), P_1(x), \dots, P_n(x)) \in \mathbb{R}^{n+1}. \quad (4.22)$$

Now, with this formula, it can be seen that the $\phi(x)$ values are orthogonal to each other, indicating that the Legendre kernel is also orthogonal [43].

Theorem 4.1 [24] *Legendre kernel is a valid Mercer kernel.*

Proof According to Mercer theorem introduced at (2.2.1), a valid kernel function that needs to be positive semi-definite or equivalently should satisfy the necessary and sufficient conditions of Mercer's Theorem. As Mercer theorem states any SVM kernel to be a valid kernel should be non-negative, in a precise way:

$$\iint K(x, z) w(x, z) f(x) f(z) dx dz \geq 0, \quad (4.23)$$

where

$$K(x, z) = \sum_{i=0}^n P_i(x) P_i^T(z), \quad (4.24)$$

and $w(x, z) = 1$, and also $f(x)$ is a function where : $f : \mathbb{R}^m \rightarrow \mathbb{R}$. Assuming each element is independent of the others, the mercer condition $K(x, z)$ can be evaluated as follows:

$$\begin{aligned} \iint K(x, z) w(x, z) f(x) f(z) dx dz &= \iint \sum_{j=0}^n P_j(x) P_j^T(z) f(x) f(z) dx dz, \\ &= \sum_{j=0}^n \iint P_j(x) P_j^T(z) f(x) f(z) dx dz, \\ &= \sum_{j=0}^n \left[\int P_j(x) f(x) dx \int P_j^T(z) f(z) dz \right], \\ &= \sum_{j=0}^n \left[\left(\int P_j(x) f(x) dx \right) \left(\int P_j^T(z) f(z) dz \right) \right], \\ &\geq 0. \end{aligned} \quad (4.25)$$

□

The Legendre kernels up to 3rd order can be expressed as:

$$\begin{aligned} K(x, z) &= 1 + xz, \\ K(x, z) &= 1 + xz + \frac{9(xz)^2 - 3x^2 - 3z^2 + 1}{4}, \\ K(x, z) &= 1 + xz + \frac{9(xz)^2 - 3x^2 - 3z^2 + 1}{4} + \frac{25(xz)^3 - 15x^3z - 15xz^3 + 9xz}{4}. \end{aligned} \quad (4.26)$$

This kernel can be expanded and specified as the following for vector inputs $x, z \in \mathbb{R}^d$:

$$K(\mathbf{x}, \mathbf{z}) = \prod_{j=1}^d K_j(x_j, z_j) = \prod_{j=1}^d \sum_{i=0}^n P_i(x_j)P_i(z_j), \quad (4.27)$$

where x_j is j-th element of the vector \mathbf{x} .

Given that the multiplication of two valid kernels is still a valid kernel, this kernel is also valid. The same as Chebyshev kernel function for vector input, each feature of the input vector for Legendre kernel function lies in [-1, 1]. So the input data has been normalized to [-1,1] via the formula:

$$x_i^{new} = \frac{2(x_i^{old} - Min_i)}{Max_i - Min_i} - 1, \quad (4.28)$$

where x_i is the i-th feature of the vector \mathbf{x} , Max_i and Min_i are the minimum and maximum values along the i-th dimensions of all the training and test data, respectively.

4.3.2 Other Legendre kernel functions

Apart from the ordinary form of the Legendre kernel function, other kernels with unique properties and special weight functions are defined on Legendre polynomials, some of which are introduced in this section.

4.3.2.1 Generalized Legendre kernel

Ozer et al. [44] applied kernel functions onto vector inputs directly instead of applying them to each input element. In fact, the generalized Legendre kernel by generalized Legendre polynomial was proposed as follows [45]:

$$K_{G-Legendre}(x, z) = \sum_{i=0}^n P_i(x)P_i^T(z), \quad (4.29)$$

where

$$\begin{aligned} P_0(x) &= 1, \\ P_1(x) &= x, \\ P_n(x) &= \frac{2n-1}{n}xP_{n-1}^T(x) - \frac{n-1}{n}P_{n-2}(x). \end{aligned} \quad (4.30)$$

Regarding Chebyshev generalized kernels, it should be noted that these kernels have more complex expressions than generalized Legendre kernels due to their

more complex weight function, so generalized Chebyshev kernels can depict more abundant nonlinear information.

4.3.2.2 Exponentially Legendre kernel

The exponentially modified orthogonal polynomial kernels are actually the product of the well-known Gaussian kernel and the corresponding generalized orthogonal polynomial kernels (without weighting function). Since an exponential function (the Gaussian kernel) can capture local information along the decision surface better than the square root function, Ozer et al. [44] replaced the weighting function with Gaussian kernel ($\frac{1}{\exp \gamma \|x-z\|^2}$), and defined the exponentially modified Legendre kernels as [45]:

$$K_{exp-Legendre}(x, z) = \frac{\sum_{i=0}^n P_i(x)P_i^T(z)}{\exp \gamma \|x - z\|^2}. \quad (4.31)$$

It should be noted that the modified generalized orthogonal polynomial kernels can be seen as semi-local kernels. Also, having two parameters, n and $\gamma > 0$, makes the optimization of these two kernels more difficult to exploit than that of the generalized orthogonal polynomial kernels [45].

4.3.2.3 Triangularly modified Legendre kernel

Triangular kernel, which is basically an affine function of the Euclidean distance ($d(i, j)$) between the points in the original space, is expressed as [46]:

$$K(x, z) = (1 - \frac{\|x - z\|}{\lambda})_+, \quad (4.32)$$

where the $(\cdot)_+$ forces this mapping to be positive and ensures this expression to be a kernel. Therefore, the triangularly modified Legendre kernels can be written as follows [45, 14, 15]:

$$K_{Tri-Legendre}(x, z) = (1 - \frac{\|x - z\|}{\lambda})_+ \sum_{i=0}^n P_i(x)P_i^T(z), \quad (4.33)$$

where $\lambda = \max\{d(x_i, \bar{x}) | \bar{x} = \frac{1}{N} \sum_{i=1}^N x_i, x_i \in X\}$, X is a finite sample set, and N is the number of sample. Thus all the data live in a ball of radius λ . Since the parameter λ only depends on the input data, the triangularly modified orthogonal polynomial kernels have a unique parameter chosen from a small set of integers.

4.3.3 Fractional Legendre kernel

Given that the fractional weight function is $(fw(x, z) = (xz^T)^{\alpha-1})$ and similar to the Legendre kernel, the corresponding fractional forms are introduced as:

$$K_{FLegendre}(X, Z) = \prod_{j=1}^m \sum_{i=0}^n FP_i^\alpha(x'_{x_j}) FP_i^\alpha(x'_{z_j}) fw(x'_{x_j}, x'_{z_j}), \quad (4.34)$$

where m is the dimension of vector X and Z . According to the procedure we have in this book, after introducing a kernel, we should guarantee its validity. So we can continue with the following theorem.

Theorem 4.2 *The fractional Legendre kernel is a valid Mercer kernel.*

Proof According to Mercer's theorem introduced at (2.2.1), a valid kernel should satisfy the sufficient conditions of Mercer's Theorem. As we know, Mercer theorem states any SVM kernel to be a valid kernel must be non-negative, in a precise way:

$$\iint K(x, z) w(x, z) f(x) f(z) dx dz \geq 0, \quad (4.35)$$

where:

$$K(x, z) = \sum_{i=0}^n FP_i^\alpha(x'_x) FP_i^\alpha(x'_z)^T fw(x'_{x_j}, x'_{z_j}), \quad (4.36)$$

and $fw(x, z) = (xz^T)^{\alpha-1}$, where $f(x)$ is function as $f : \mathbb{R}^m \rightarrow \mathbb{R}$. Thus, we have:

$$\begin{aligned} & \iint K(x, z) w(x, z) f(x) f(z) dx dz \\ &= \iint \sum_{i=0}^n FP_i^\alpha(x'_x) FP_i^\alpha(x'_z)^T fw(x'_{x_j}, x'_{z_j}) f(x) f(z) dx dz, \\ &= \sum_{i=0}^n \iint FP_i^\alpha(x'_x) FP_i^\alpha(x'_z)^T (x'_x x'^T_z)^{\alpha-1} f(x) f(z) dx dz, \\ &= \sum_{i=0}^n \left[\int FP_i^\alpha(x'_x) (x'_x)^{\alpha-1} f(x) dx \int FP_i^\alpha(x'_z)^T (x'^T_z)^{\alpha-1} f(z) dz \right], \\ &= \sum_{i=0}^n \left[\left(\int FP_i^\alpha(x'_x) (x'_x)^{\alpha-1} f(x) dx \right) \left(\int FP_i^\alpha(x'_z)^T (x'^T_z)^{\alpha-1} f(z) dz \right) \right], \\ &\geq 0. \end{aligned} \quad (4.37)$$

□

4.4 Application of Legendre kernel functions on real datasets

In this section, the application of the ordinary Legendre kernel and the fractional Legendre kernel are shown in SVM. Also, the obtained results on two real datasets are compared with the results of RBF kernels, ordinary polynomial kernel, ordinary Chebyshev kernel, and fractional Chebyshev kernel.

4.4.1 Spiral dataset

The Spiral dataset has been already introduced in the previous chapter. In this section, the Legendre kernel and the fractional Legendre kernel are used to classifying spiral datasets using SVM. As we mentioned before, this multi-class classification data set can be split into 3 binary classification data sets. It was also previously explained (Fig. 3.5), by transferring data to the fractional space using the α coefficient ($0.1 \leq \alpha \leq 0.9$), the data density gradually decreases with decreasing alpha value, and the central axis of data density from the point $(0, 0)$ is transferred to point $(1, 1)$. Although it seems that data points are conglomerated on one corner of the axis on 2D plot Fig. 4.4, this is not necessarily what happens in 3D space when the kernel function is applied Fig. 4.5.

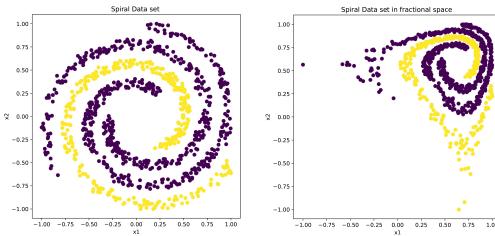


Fig. 4.4: Spiral dataset, in normal and fractional spaces

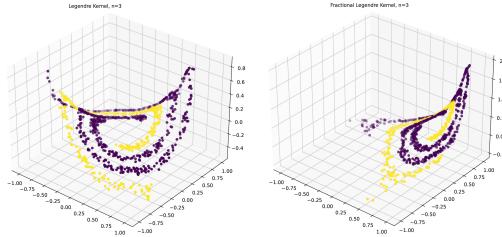


Fig. 4.5: Normal and fractional Legendre kernels of order 3, and $\alpha = 0.4$ applied on Spiral dataset in 3D space

Using transferring the data to the fractional space, the decision boundaries are found for the problem with the above-mentioned divisions with the help of the Legendre and fractional Legendre kernel functions.

In order to have an intuition of how the decision boundaries are different, we can see Fig. 4.6, which depicts corresponding Legendre classifiers of different orders [3, 4, 5, 6] on the original Spiral dataset, where the binary classification of 1-vs-[2,3] is chosen. From these figures, it can be seen that the boundaries of decision-making are becoming more twisted.

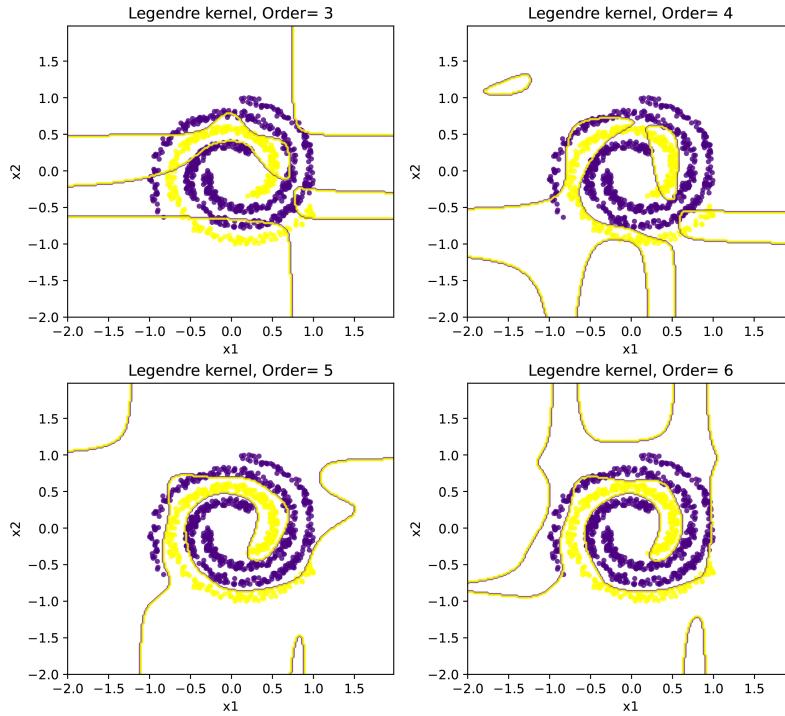


Fig. 4.6: Legendre kernel of orders 3, 4, 5, and 6 on Spiral dataset

The complexity of the decision boundary even gets worse in fractional space. Fig. 4.7 demonstrates the decision boundary of fractional Legendre classifier with different orders of 3, 4, 5, and 6, where $\alpha = 0.4$. Again there is a complicated decision boundary as one approaches from order 3 to order 6.¹

¹ Based on many studies, it was concluded that the fractional Legendre kernel with order 3 is not a suitable option for use on the spiral dataset.

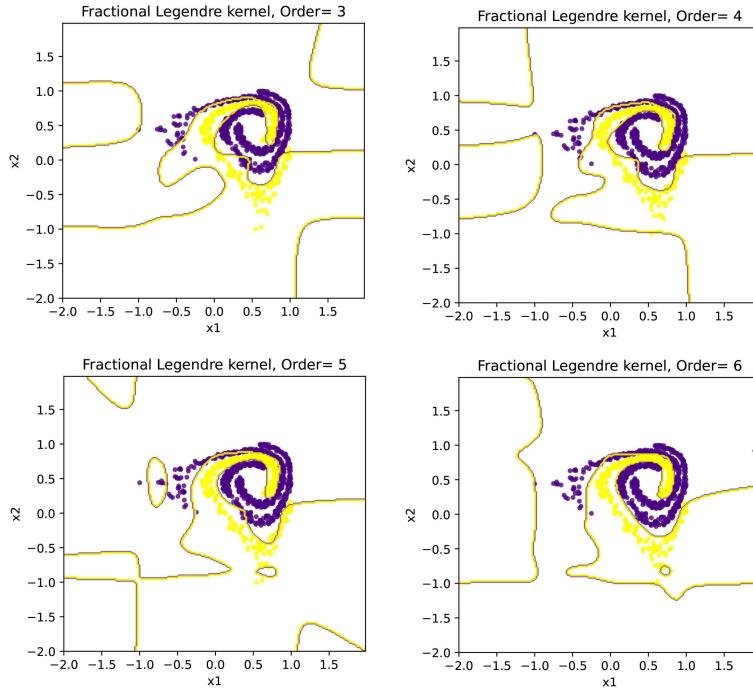


Fig. 4.7: Fractional Legendre kernel functions of orders 3, 4, 5, and 6 on Spiral dataset with $\alpha = 0.4$

Each decision boundary or decision surface in 3D space classifies the data points in a specific form. As the decision boundary/surface of a kernel function of each order is fixed and determined, it is the degree of correlation between that specific shape and the data points that determine the final accuracy.

The experiment results are summarized in the following tables. In particular, Table 4.2 summarizes the results of class 1-vs-[2,3]. As it can be seen, **fractional Legendre** kernel outperforms other kernels, and **fractional Chebyshev** kernel has the second-best accuracy. Also, the classification accuracy of mentioned kernels on class 2-vs-[1,3] is summarized in Table 4.3, where the **RBF** kernel has the best accuracy score. Finally, Table 4.4 is the classification accuracy scores on class 3-vs-[1,2], in which **fractional Legendre** kernel has the best performance.

Table 4.2: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, and fractional Legendre accuracy scores on Spiral dataset. It is clear that the RBF kernel outperforms other kernels

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 0.73 | - | - | - | - | 0.97 |
| Polynomial | - | 8 | - | - | - | 0.9533 |
| Chebyshev | - | - | 5 | - | - | 0.9667 |
| Fractional Chebyshev | - | - | 3 | 0.3 | - | 0.9733 |
| Legendre | - | - | 7 | - | - | 0.9706 |
| Fractional Legendre | - | - | 7 | 0.4 | - | 0.9986 |

Table 4.3: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, and fractional Legendre accuracy scores on Spiral dataset. It is clear that the RBF kernel outperforms other kernels

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 0.1 | - | - | - | - | 0.9867 |
| Polynomial | - | 5 | - | - | - | 0.9044 |
| Chebyshev | - | - | 6 | - | - | 0.9289 |
| Fractional Chebyshev | - | - | 6 | 0.8 | - | 0.9344 |
| Legendre | - | - | 8 | - | - | 0.9773 |
| Fractional Legendre | - | - | 8 | 0.4 | - | 0.9853 |

Table 4.4: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, and fractional Legendre accuracy scores on Spiral dataset. It is clear that the RBF and Polynomials kernels are better than others

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|-----------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 0.73 | - | - | - | - | 0.98556 |
| Polynomial | - | 5 | - | - | - | 0.98556 |
| Chebyshev | - | - | 6 | - | - | 0.9622 |
| Fractional Chebyshev | - | - | 6 | 0.6 | - | 0.9578 |
| Legendre | - | - | 7 | - | - | 0.9066 |
| Fractional Legendre | - | - | 5 | 0.4 | - | 0.9906 |

4.4.2 Three Monks's dataset

As another case example, the three Monks problem is considered here (see Chapter 3 for more information about the dataset). The *Legendre kernel* (i.e., Eq. 4.21) and the *fractional Legendre kernel* (i.e., Eq. 4.34) are applied to the data sets from *the three Monks's problem*. Table 4.5 illustrates the output accuracy of each model on the first problem of Monks each model where *RBF* kernel has the best accuracy by $\sigma \approx 2.844$ at 1 and *Chebyshev* kernel has worst among them at **0.8472**

Table 4.5: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, and fractional kernels on Monks's first problem. It can be seen that the Legendre and fractional Legendre have the most desirable accuracy which is 1

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|-----------------------------|-------|-------|-------|-------------------|---------------------|---------------|
| RBF | 2.844 | - | - | - | - | 0.8819 |
| Polynomial | - | 3 | - | - | - | 0.8681 |
| Chebyshev | - | - | 3 | - | - | 0.8472 |
| Fractional Chebyshev | - | - | 3 | 1/16 | - | 0.8588 |
| Legendre | - | - | 4 | - | - | 0.8333 |
| Fractional Legendre | - | - | 4 | 0.1 | - | 0.8518 |

Also, Table 4.6 illustrates the output accuracy of each model on the second problem of Monks each model, where *fractional Legendre* kernel has the best accuracy by $\alpha \approx 0.8$ at 1 and *Legendre* kernel has worst among them at **0.8032**

Table 4.6: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, and fractional kernels on Monks's second problem. The fractional Chebyshev Kernel has the second-best result, following fractional Legendre Kernel

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|-----------------------------|--------|-------|-------|-------------------|---------------------|----------|
| RBF | 5.5896 | - | - | - | - | 0.875 |
| Polynomial | - | 3 | - | - | - | 0.8657 |
| Chebyshev | - | - | 3 | - | - | 0.8426 |
| Fractional Chebyshev | - | - | 3 | 1/16 | - | 0.9653 |
| Legendre | - | - | 3 | - | - | 0.8032 |
| Fractional Legendre | - | - | 3 | 0.1 | - | 1 |

Finally, Table 4.7 illustrates the output accuracy of each model on the third problem of Monks each model, where *fractional Chebyshev* kernel by $\alpha \approx \frac{1}{16}$ and

RBF kernel have the best accuracies at **0.91** and *fractional Legendre* kernel has worst among them at **0.8379**

Table 4.7: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, and fractional kernels on Monks's third problem. It is shown that the RBF and fractional Chebyshev kernel have the best results

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|-----------------------------|--------|-------|-------|-------------------|---------------------|-------------|
| RBF | 2.1586 | - | - | - | - | 0.91 |
| Polynomial | - | 3 | - | - | - | 0.875 |
| Chebyshev | - | - | 6 | - | - | 0.895 |
| Fractional Chebyshev | - | - | 5 | 1/5 | - | 0.91 |
| Legendre | - | - | 4 | - | - | 0.8472 |
| Fractional Legendre | - | - | 3 | 0.8 | - | 0.8379 |

4.5 Conclusion

In this chapter, the Legendre polynomial kernel in fractional order for support vector machines is introduced and presented by using the ordinary Legendre polynomial kernel. The Legendre polynomial kernel can extract good properties from data due to the orthogonality of elements of the feature vector, thus reducing data redundancy. Also, based on the results of the experiment on two data sets in the previous section, it can be shown that SVM with Legendre and fractional Legendre kernels can separate nonlinear data well.

References

1. Haitjema, H.: Surface profile and topography filtering by Legendre polynomials. *Surf. Topogr.* **9**, 15–17 (2021)
2. Chang, R. Y., Wang, M. L.: Shifted Legendre function approximation of differential equations; application to crystallization processes. *Comput. Chem. Eng.* **8**, 117–125 (1984)
3. Holdeman, Jr., Jonas, T.: Legendre polynomial expansions of hypergeometric functions with applications. *J Math Phys* **11**, 114–117 (1970)
4. Sánchez-Ruiz, J. Dehesa, J.S.: Expansions in series of orthogonal hypergeometric polynomials. *Journal of computational and applied mathematics* **89**, 155–170 (1998)
5. Spencer, L. V.: Calculation of peaked angular distributions from Legendre polynomial expansions and an application to the multiple scattering of charged particles. *Phys. Rev.* **90**, 146–150 (1953)
6. Kaghoshvili, E Kh., Zank, GP., Lu, JY., Dröge, W. : Transport of energetic charged particles. Part 2. Small-angle scattering. *Journal of plasma physics* **70**, 505–532 (2004)

7. Dash, R.: Performance analysis of an evolutionary recurrent Legendre Polynomial Neural Network in application to FOREX prediction. *J. King Saud Univ. - Comput. Inf. Sci.* **32**, 1000–1011 (2020)
8. Dash, R., Dash, P. K.: MDHS–LPNN: A hybrid FOREX predictor model using a Legendre polynomial neural network with a modified differential harmony search technique. *Handbook of neural computation*. Academic Press, 459–486 (2017)
9. Moayeri, M. M., Rad, J. A., Parand, K.: Dynamical behavior of reaction-diffusion neural networks and their synchronization arising in modeling epileptic seizure: A numerical simulation study. *Comput. Math. with Appl.* **80**, 1887–1927 (2020)
10. HWANG, C., Mu-Yang, Chen.: Analysis and optimal control of time-varying linear systems via shifted Legendre polynomials. *Int. J. Control* **41**, 1317–1330 (1985)
11. Chang, R. Y., Wang, M. L.: Model reduction and control system design by shifted Legendre polynomial functions. *J Dyn Syst Meas Control* **105**, 52–55 (1983)
12. Qian, C. B., Tianshu, L., Jinsong, L. H. Liu, Z.: Synchrophasor estimation algorithm using Legendre polynomials. *IEEE PES General Meeting Conference and Exposition*, (2014)
13. Gao, Jie, Lyu, Yan, Zheng, Mingfang, Liu, Mingkun, Liu, Hongye, Wu, Bin, He, Cunfu.: Application of state vector formalism and Legendre polynomial hybrid method in the longitudinal guided wave propagation analysis of composite multi-layered pipes. *Wave Motion* **100**, 102670 (2021)
14. Belanche Muñoz, L. A.: Developments in kernel design. In *ESANN 2013 proceedings: European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 369–378 (2013)
15. Fleuret, F., Sahbi, H.: Scale-invariance of support vector machines based on the triangular kernel. In *3rd International Workshop on Statistical and Computational Theories of Vision*, 1–13 (2003)
16. Dahmen, S., Morched, B. A., Mohamed Hédi, B. G.: Investigation of the coupled Lamb waves propagation in viscoelastic and anisotropic multilayer composites by Legendre polynomial method. *Compos. Struct.* **153**, 557–568 (2016)
17. Zheng, Mingfang, He, Cunfu, Lyu, Yan, Wu, Bin.: Guided waves propagation in anisotropic hollow cylinders by Legendre polynomial solution based on state-vector formalism. *Compos. Struct.* **207**, 645–657 (2019)
18. Gao, Jie, Lyu, Yan, Zheng, Mingfang, Liu, Mingkun, Liu, Hongye, Wu, Bin, He, Cunfu.: Application of Legendre orthogonal polynomial method in calculating reflection and transmission coefficients of multilayer plates. *Wave Motion* **84**, 32–45 (2019)
19. Mohammadi, F., Hosseini, M. M. :A new Legendre wavelet operational matrix of derivative and its applications in solving the singular ordinary differential equations. *J Franklin Inst.* **348**, 1787–1796 (2011)
20. Chang, P., Isah, A.: Legendre Wavelet Operational Matrix of fractional Derivative through wavelet-polynomial transformation and its Applications in Solving Fractional Order Bruselator system. *Journal of Physics: Conference Series* **693**, (2016)
21. Chang, R. Y., Wang, M. L.: Optimal control of linear distributed parameter systems by shifted Legendre polynomial functions. *J Dyn Syst Meas Control* **105**, 222–226 (1983)
22. Zeghdane, R.: Numerical approach for solving nonlinear stochastic Itô–Volterra integral equations using shifted Legendre polynomials. *International Journal of Dynamical Systems and Differential Equations* **11**, 69–88 (2021)
23. Mall, S., Chakraverty, S.: Application of Legendre neural network for solving ordinary differential equations. *Applied Soft Computing* **43**, 347–356 (2016)
24. Pan, Z. B., Chen, H., You, X. H.: Support vector machine with orthogonal Legendre kernel. *International Conference on Wavelet Analysis and Pattern Recognition*. IEEE, 125–130 (2012)
25. Afifi, A., Zanaty, EA. : Generalized Legendre Polynomials for Support Vector Machines (SVMS) Classification. *International Journal of Network Security & Its Applications (IJNSA)* **11**, 87–104 (2019)
26. Marianela, P., Gómez, J. C.: Legendre polynomials based feature extraction for online signature verification. Consistency analysis of feature combinations. *Pattern Recognit* **47**, 128–140 (2014)

27. Benouini, R., Batioua, I., Zenkouar, Kh., Mrabti, F.: New set of generalized Legendre moment invariants for pattern recognition. *Pattern Recognit. Lett.* **123**, 39–46 (2019)
28. Saadatmandi, A., Dehghan, M.: A new operational matrix for solving fractional-order differential equations. *Comput. Math. with Appl.* **59**, 1326–1336 (2010)
29. Bhrawy, A. H., Abdelkawy, M. A., Machado, J. T., Amin, A. Z. M.: Legendre–Gauss–Lobatto collocation method for solving multi-dimensional Fredholm integral equations. *Comput. Math. Appl.* **4**, 1–13 (2016)
30. Ezz-Eldien, S. S., Doha, E. H., Baleanu, D., Bhrawy, A. H.: A numerical approach based on Legendre orthonormal polynomials for numerical solutions of fractional optimal control problems. *J VIB CONTROL* **23**, 16–30 (2017)
31. Kazem, S., Shaban, M., Rad, J. A.: Solution of the coupled Burgers equation based on operational matrices of d-dimensional orthogonal functions. *Zeitschrift für Naturforschung A* **67**, 267–274 (2012)
32. Kazem, S., Abbasbandy, S., Kumar, S.: Fractional-order Legendre functions for solving fractional-order differential equations. *Appl. Math. Model.* **37**, 5498–5510 (2013)
33. Asghari, M., Hadian Rasanan, A.H., Gorgin, S., Rahmati, D., Parand, K.: FPGA-orthopoly: a hardware implementation of orthogonal polynomials. *Eng Comput* (2022) doi: 10.1007/s00366-022-01612-x
34. Bhrawy, A. H., Doha, E. H., Ezz-Eldien, S. S., Abdelkawy, M. A. A numerical technique based on the shifted Legendre polynomials for solving the time-fractional coupled KdV equations. *Calcolo* **53**, 1–17 (2016)
35. Rad, J. A., Kazem, S., Shaban, M., Parand, K., Yildirim, A. H. M. E. T.: Numerical solution of fractional differential equations with a Tau method based on Legendre and Bernstein polynomials. *Math. Methods Appl. Sci.* **37**, 329–342 (2014)
36. Parand, K., Razzaghi, M.: Rational Legendre approximation for solving some physical problems on semi-infinite intervals. *Phys. Scr.* **69**, 353 (2004)
37. Parand, K., Shahini, M., Dehghan, M.: Rational Legendre pseudospectral approach for solving nonlinear differential equations of Lane–Emden type. *J. Comput. Phys.* **228**, 8830–8840 (2009)
38. N Parand, K., Delafkar, Z., Rad, J. A., Kazem S.: Numerical study on wall temperature and surface heat flux natural convection equations arising in porous media by rational Legendre pseudo-spectral approach. *Int. J. Nonlinear Sci* **9**, 1–12 (2010)
39. Olver, F. W.J., Lozier, D. W., Boisvert, R. F., Clark, C. W.: NIST handbook of mathematical functions hardback and CD-ROM. Cambridge university press, Singapore (2010)
40. Doman, B. G. S.: The classical orthogonal polynomials. World Scientific, Singapore (2015)
41. Lamb Jr, G. L.: Introductory applications of partial differential equations: with emphasis on wave propagation and diffusion. John Wiley & Sons, Amsterdam (2011)
42. Hadian Rasanan, A. H., Rahmati, D., Gorgin, S., Parand, K.: A single layer fractional orthogonal neural network for solving various types of Lane–Emden equation. *New Astron.* **75**, 101307, (2020)
43. Shawe-Taylor, J., Cristianini, N.: Kernel methods for pattern analysis. Cambridge university press, Cambridge (2004)
44. Ozer, S., Chi, H., Chen, Hakan, A., Cirpan.: A set of new Chebyshev kernel functions for support vector machine pattern classification. *Pattern Recognit.* **44**, 1435–1447 (2011)
45. Tian, M., Wang, W.: Some sets of orthogonal polynomial kernel functions. *Appl. Soft Comput.* **61**, 742–756 (2017)
46. Fleuret, F., Sahbi, H.: Scale-invariance of support vector machines based on the triangular kernel. 3rd International Workshop on Statistical and Computational Theories of Vision. (2003)
47. Shen, J.: Efficient spectral-Galerkin method I. Direct solvers of second-and fourth-order equations using Legendre polynomials. *SISC* **15**, 1489–1505 (1994)
48. Voelker, A., Kajić, I., Eliasmith, C.: Legendre memory units: Continuous-time representation in recurrent neural networks. *Advances in neural information processing systems* **32**, (2019)
49. Hadian Rasanan, A. H., Bajalan, N., Parand, K., Rad, J. A.: Simulation of nonlinear fractional dynamics arising in the modeling of cognitive decision making using a new fractional neural network. *Math. Methods Appl. Sci.* **43**, 1437–1466 (2020)

Chapter 5

Fractional Gegenbauer Kernel Functions: Theory and Application

Sherwin Nedaei Janbesaraei and Amirreza Azmoon and Dumitru Baleanu

Abstract Because of the usage of many functions as a kernel, the support vector machine method has demonstrated remarkable versatility in tackling numerous machine learning issues. Gegenbauer polynomials, like the Chebyshev and Legendre polynomials which are introduced in previous chapters, are among the most commonly utilized orthogonal polynomials that have produced outstanding results in the support vector machine method. In this chapter, some essential properties of Gegenbauer and fractional Gegenbauer functions are presented and reviewed, followed by the kernels of these functions, which are introduced and validated. Finally, the performance of these functions in addressing two issues (two example datasets) is evaluated.

5.1 Introduction

Gegenbauer polynomials or better known as *ultra-spherical polynomials*, are another family of classic orthogonal polynomials, named after Austrian mathematician **Leopold Bernhard Gegenbauer** (1849-1903). As the name itself suggests, *ultra-spherical polynomials* provide a natural extension to spherical harmonics in higher dimensions[1]. *Spherical Harmonics* are special functions defined on the surface of a sphere. They are often being used in applied mathematics for solving differ-

Sherwin Nedaei Janbesaraei

School of Computer Science, Institute for Research in Fundamental Sciences (IPM), Tehran, Iran,
e-mail: sherwin.nedaei@gmail.com

Amirreza Azmoon

Department of Computer Science, The Institute for Advance Studies in Basic Sciences (IASBS),
Zanjan, Iran e-mail: a.azmoon@iasbs.ac.ir

Dumitru Baleanu

Department of Mathematics, Cankaya University, Ankara, 06530, Turkey e-mail: dumitru@cankaya.edu.tr

ential equations[2, 3]. In recent years the classical orthogonal polynomials, especially the Gegenbauer orthogonal polynomials have been used to address pattern recognition[9, 10, 11], classification[12, 13], and kernel-based learning in many fields such as physic[22], medical image processing[4, 5, 20], electronic[14], and other basic fields[6, 7, 8]. In 2006, Pawlak [15] showed the benefits of the image reconstruction process based on Gegenbauer polynomials and corresponding moments. Then in the years that followed, Hosny [16] used the high computational requirements of Gegenbauer moments to propose novel methods for image analysis and recognition, which in these studies, fractional-order shifted Gegenbauer moments [17] and Gegenbauer moment Invariants [18] are seen. Also in 2014, Abd Elaziz and Hosny [19] used artificial bee colony based on orthogonal Gegenbauer moments to be able to classify galaxies images with the help of support vector machines. In 2009, with the help of the orthogonality property of the Gegenbauer polynomial, Langley [21] introduced a new 3D phase unwrapping algorithm for the analysis of magnetic resonance imaging (MRI). Also, in 1995, Ludlow studied the application of Gegenbauer polynomials to the emission of light from spheres [22]. Clifford polynomials are particularly well-suited as kernel functions for a higher-dimensional continuous wavelet transform. In 2004, Brackx [23] constructed Clifford–Gegenbauer and generalized Clifford–Gegenbauer polynomials as new specific wavelet kernel functions for a higher dimensional continuous wavelet transform. Then, Ilić [24] in 2011, with the use Christoffel-Darboux formula for Gegenbauer orthogonal polynomials, present the filter function solution that exhibits optimum amplitude as well as optimum group delay characteristics.

Flexibility in using different kernels in the SVM algorithm is one of the reasons why orthogonal classical polynomials have recently been used as kernels. The Gegenbauer polynomial is one of those that have been able to show acceptable results in this area. In 2018, Padierna [25] introduced a formulation of orthogonal polynomial kernel functions for SVM classification. Also in 2020, he [27] used this kernel to classify peripheral arterial disease in patients with type 2 diabetes. In addition to using orthogonal polynomials as kernels in SVM to solve classification problems, these polynomials can be used as kernels in support vector regression (SVR) to solve regression problems as well as help examine time series problems. In 1992, Azeri [28] proposed a corresponding ultra-spherical kernel. Also, in 2019, Feng suggests using extended support vector regression (X-SVR), a new machine learning-based metamodel, for the reliability study of dynamic systems using first-passage theory. Furthermore, the power of X-SVR is strengthened by a novel kernel function built from the vectorized Gegenbauer polynomial, specifically for handling complicated engineering problems [29]. On the other hand, in 2001, Ferrara [30] dealt with the k-factor extension of the long memory Gegenbauer process, which this model investigated the predictive ability of the k-factor Gegenbauer model on real data of urban transport traffic in the Paris area. Other applications of orthogonal Gegenbauer polynomials include their effectiveness in building and developing neural networks. In 2018, Zhen [32] investigated and constructed a two-input Gegenbauer orthogonal neural network (TIGONN) using probability theory, polynomial interpolation, and approximation theory to avoid the inherent problems of the back-propagation (BP)

training algorithm. Then, In 2019, a novel type of neural network based on Gegenbauer orthogonal polynomials, termed as GNN, was constructed and investigated [33]. This model could overcome the computational robustness problems of extreme learning machines (ELM), while still having comparable structural simplicity and approximation capability [33]. In addition to the applications mentioned here, Table 5.1 can be seen for some applications for different kinds of these polynomials.

Leopold Bernhard Gegenbauer (1849-1903) was an Austrian mathematician. However, he entered the University of Vienna to study history and linguistics, but he graduated as a mathematics and physics teacher. His professional career in mathematics specifically began with a grant awarded to him and made it possible for him to undertake research at the University of Berlin for two years, where he could attend lectures of great mathematicians of that time such as Karl Weierstrass, Eduard Kummer, Hermann Helmholtz, and Leopold Kronecker. Leopold Gegenbauer had many interests in mathematics such as number theory and function theory. He introduced a class of orthogonal polynomials, which has known as Gegenbauer Orthogonal Polynomials, in his doctoral thesis of 1875. However the name of Gegenbauer appears in many mathematical concepts such as Gegenbauer transforms, Gegenbauer's integral inequalities, Gegenbauer approximation, Fourier-Gegenbauer sums, the Gegenbauer oscillator, and many more, but another important contribution of Gegenbauer was designing a course on "Insurance Theory" at the University of Vienna, where he was appointed as a full professor of mathematics from 1893 till his death at 1903.^a

^a For more information about **L.B. Gegenbauer** and his contribution, please see: <https://mathshistory.st-andrews.ac.uk/Biographies/Gegenbauer/>.

Table 5.1: Some applications for different kinds of Gegenbauer polynomials

| | |
|---|--|
| Gegenbauer polynomials | Srivastava [34] suggested a potentially helpful novel approach for solving the Bagley-Torvik problem based on the Gegenbauer wavelet expansion and operational matrices of fractional integral and block-pulse functions. |
| Rational Gegenbauer functions | Based on rational Gegenbauer functions, Parand has solved numerically the third-order nonlinear differential equation [35] and a sequence of linear ordinary differential equations (ODE) that was converted with the quasi-linearization method (QLM) [36]. |
| Generalized Gegenbauer functions | Belmehdi [38] created a differential-difference relation, and this sequence solves the second-order and fourth-order differential equation fulfilled by the association (of arbitrary order) of the generalized Gegenbauer. Also, Cohl and Liu [39, 40], have demonstrated applications of these functions in their studies. Then, in 2020, Yang [41] proposed a novel looseness state recognition approach for bolted structures based on multi-domain sensitive features derived from quasi-analytic wavelet packet transform (QAWPT) and generalized Gegenbauer support vector machine (GGSVM). |
| Fractional order of rational Gegenbauer functions | To solve the Thomas-Fermi problem, Hadian et al. [42] proposed two numerical methods based on the Newton iteration method and spectral algorithms. The spectral technique was used in both approaches and was based on the fractional order of rational Gegenbauer functions [42]. |

In this chapter, a new type of Gegenbauer orthogonal polynomials are introduced, called fractional Gegenbauer functions, and the corresponding kernels are constructed, which is excellently applicable in kernel-based learning methods like

support vector machines. In Section 5.2, the basic definition and properties of Gegenbauer polynomials and their fractional form are introduced. Then, in Section 5.3 in addition to explaining the previously proposed Gegenbauer kernel functions, the fractional Gegenbauer function is proposed. Also, in Section 5.4, the results of SVM classification with Gegenbauer and fractional Gegenbauer kernel on well-known data set are compared with similar results from the previous chapters. Finally, in Section 5.5 the conclusion remarks will summarize the whole chapter.

5.2 Preliminaries

In this section, the basics of Gegenbauer polynomials have been covered. These polynomials have been defined using the related differential equation and, in the following, the properties of Gegenbauer polynomials have been introduced, and also the fractional form of them has been defined besides their properties too.

5.2.1 Properties of Gegenbauer polynomials

Gegenbauer polynomials of degree n , $G_n^\lambda(x)$ and order $\lambda > -\frac{1}{2}$ are solutions to the following Sturm-Liouville differential equation [43, 25, 31]:

$$(1-x^2) \frac{d^2y}{dx^2} - (2\lambda+1)x \frac{dy}{dx} + n(n+2\lambda)y = 0, \quad (5.1)$$

where n is a positive integer, and λ is a real number and greater than -0.5 . Gegenbauer polynomials are orthogonal on the interval $[-1, 1]$ with respect to the weight function[25, 36, 31]:

$$w(x) = (1-x^2)^{\lambda-\frac{1}{2}}. \quad (5.2)$$

Therefore the orthogonality relation is defined as[22, 36, 42, 31]:

$$\int_{-1}^1 G_n^\lambda(x) G_m^\lambda(x) w(x) dx = \frac{\pi 2^{1-2\lambda} \Gamma(n+2\lambda)}{n!(n+\lambda)(\Gamma(\lambda))^2} \delta_{nm}, \quad (5.3)$$

where δ_{nm} is the Kronecker delta function [45]. The standard Gegenbauer polynomial $G_n^{(\lambda)}(x)$, can be defined also as follows[36, 42]:

$$G_n^\lambda(x) = \sum_{n=1}^{\lfloor \frac{n}{2} \rfloor} (-1)^j \frac{\Gamma(n+\lambda-j)}{j!(n-2j)!\Gamma(\lambda)} (2x)^{n-2j}, \quad (5.4)$$

where $\Gamma(.)$ is the Gamma function.

Assuming $\lambda \neq 0$ and $\lambda \in \mathbb{R}$, the generating function for Gegenbauer polynomial is given as[39, 43, 31]:

$$G_n^\lambda(x, z) = \frac{1}{(1 - 2xz + z^2)^\lambda}. \quad (5.5)$$

It can be shown that it is true for $|z| < 1$, $|x| \leq 1$, $\lambda > -\frac{1}{2}$ [39, 43]. Considering for fixed x the function is *holomorphic* in $|z| < 1$, it can be expanded in *Taylor* series[44]:

$$G_n^\lambda(x, z) = \sum_{n=0}^{\infty} G_n^\lambda(x) z^n. \quad (5.6)$$

Gegenbauer polynomials can be obtained by the following recursive formula [46, 41, 42]:

$$\begin{aligned} G_0^\lambda(x) &= 1, & G_1^\lambda(x) &= 2\lambda x, \\ G_n^\lambda(x) &= \frac{1}{n}[2x(n+\lambda-1)G_{n-1}^\lambda(x) - (n+2\lambda-2)G_{n-2}^\lambda(x)]. \end{aligned} \quad (5.7)$$

Also, other recursive relations for this orthogonal polynomial are as follow[43]:

$$(n+2)G_{n+2}^\lambda(x) = 2(\lambda+n+1)xG_{n+1}^\lambda(x) - (2\lambda+n)G_n^\lambda(x), \quad (5.8)$$

$$nG_n^\lambda(x) = 2\lambda\{xG_{n-1}^{\lambda+1}(x) - G_{n-2}^{\lambda+1}(x)\}, \quad (5.9)$$

$$(n+2\lambda)G_n^\lambda(x) = 2\lambda\{G_n^{\lambda+1}(x) - xG_{n-1}^{\lambda+1}(x)\}, \quad (5.10)$$

$$nG_n^\lambda(x) = (n-1+2\lambda)xG_{n-1}^\lambda(x) - 2\lambda(1-x^2)G_{n-2}^{\lambda-1}(x), \quad (5.11)$$

$$\frac{d}{dx}G_n^\lambda = 2\lambda G_{n+1}^{\lambda+1}. \quad (5.12)$$

Everyone who has tried to generate Gegenbauer polynomials has experienced some difficulties as the number of terms rises at each higher order, for example, for the orders of zero to four, in the following :

$$\begin{aligned} G_0^\lambda(x) &= 1, \\ G_1^\lambda(x) &= 2\lambda x, \\ G_2^\lambda(x) &= (2\lambda^2 + 2\lambda)x^2 - \lambda, \\ G_3^\lambda(x) &= (\frac{4}{3}\lambda^3 + 4\lambda^2 + \frac{8}{3}\lambda)x^3 + (-2\lambda^2 - 2\lambda)x, \\ G_4^\lambda(x) &= (\frac{2}{3}\lambda^4 + 4\lambda^3 + \frac{22}{3}\lambda^2 + 4\lambda)x^4 + (-2\lambda^3 - 6\lambda^2 - 4\lambda)x^2. \end{aligned} \quad (5.13)$$

which are depicted in Figures 5.1, 5.2, and there will be more terms in higher orders subsequently. To overcome this difficulty *Pochhammer Polynomials* have already been used to simplify[43]. The *Pochhammer Polynomials* or the *rising factorial* is defined as:

$$x^{(n)} = x(x+1)(x+2)\dots(x+n-1) = \prod_{k=1}^n (x+k-1). \quad (5.14)$$

Consequently, some of Pochhammer polynomials are:

$$\begin{aligned} x^{(0)} &= 1, \\ x^{(1)} &= x, \\ x^{(2)} &= x(x+1) = x^2 + x, \\ x^{(3)} &= x(x+1)(x+2) = x^3 + 3x^2 + 2x, \\ x^{(4)} &= x(x+1)(x+2)(x+3) = x^4 + 6x^3 + 11x^2 + 6x, \\ x^{(5)} &= x^5 + 10x^4 + 35x^3 + 50x^2 + 24x, \\ x^{(6)} &= x^6 + 15x^5 + 85x^4 + 225x^3 + 274x^2 + 120x. \end{aligned} \quad (5.15)$$

By means of Pochhammer polynomials, first few orders of Gegenbauer polynomials of order λ are:

$$\begin{aligned} G_0^\lambda(x) &= 1, \\ G_1^\lambda(x) &= 2\lambda x, \\ G_2^\lambda(x) &= a^{(2)}2x^2 - \lambda, \\ G_3^\lambda(x) &= \frac{a^{(3)}4x^3}{3} - 2a^{(2)}x, \\ G_4^\lambda(x) &= \frac{a^{(4)}2x^4}{3} - a^{(3)}2x^2 + a^{(2)}, \\ G_5^\lambda(x) &= \frac{a^{(5)}4x^5}{15} - \frac{a^{(4)}4x^3}{3} + a^{(3)}x, \\ G_6^\lambda(x) &= \frac{a^{(6)}4x^6}{45} - \frac{a^{(5)}2x^4}{3} + \frac{a^{(4)}x^2}{2} - a^{(3)}, \end{aligned} \quad (5.16)$$

where $a^{(n)}$ are *Pochhammer* polynomials.

Here for the sake of convenience, a python code has been introduced that can generate any order of Gegegnbauer polynomials of order λ symbolically, using *sympy* library:

Program Code

```
import sympy
x = sympy.Symbol("x")
lambda = sympy.Symbol(r'\lambda')

def Gn(x, n):
    if n == 0:
        return 1
```

```

elif n == 1:
    return 2*lambd*x
elif n >= 2:
    return (sympy.Rational(1,n)*(2*x*(n+lambd-1)*Gn(x,n-1) \
    -(n+2*lambd-2)*Gn(x,n-2)))

```

Program Code

```

sympy.expand(sympy.simplify(Gn(x,2)))
> 2x2λ2 + 2x2λ - λ

```

For $G_n^\lambda(x)$, $n = 0, 1, \dots, n$, $\lambda > -\frac{1}{2}$, which is orthogonal with respect to weight function $(1 - x^2)^{\lambda - \frac{1}{2}}$, there exist exactly n zeros in $[-1, 1]$ which are denoted by $x_{nk}(\lambda)$, $k = 1, \dots, n$ and are enumerated in decreasing order $1 > x_{n1}(\lambda) > x_{n2} > \dots > x_{nn}(\lambda) > -1$ [44]. The zeros of $G_n^\lambda(x)$ and $G_m^\lambda(x)$, $m > n$ separate each other and between any two zeros of $G_n^\lambda(x)$, there is at least one zero [46].

Gegenbauer polynomials follow the symmetry same as other classical orthogonal polynomials. Therefore Gegenbauer polynomials of even order have even symmetry and contain only even powers of x and, similarly odd orders of Gegenbauer polynomials have odd symmetry and contain only odd powers of x [46].

$$G_n^\lambda(x) = (-1)^n G_n^\lambda(-x) = \begin{cases} G_n^\lambda(-x), & n \text{ even} \\ -G_n^\lambda(-x), & n \text{ odd} \end{cases}, \quad (5.17)$$

Also, some special values of $G_n^\lambda(x)$ can be written as follows:

$$G_n^\lambda(1) = \frac{(2\lambda)n}{n!}, \quad (5.18)$$

$$G_{2n}^\lambda(0) = \frac{(-1)^n (\lambda)n}{n!}, \quad (5.19)$$

$$G_{2n+1}^\lambda(0) = \frac{2(-1)^n (\lambda)n + 1}{n!}. \quad (5.20)$$

5.2.2 Properties of fractional Gegenbauer polynomials

Gegenbauer polynomial of the fractional order of λ over a finite interval $[a, b]$ by use of mapping $x' = 2(\frac{x-a}{b-a})^\alpha - 1$, where $\alpha > 0$ and $x' \in [-1, 1]$, is defined as [37]:

$$FG_n^{\alpha, \lambda}(x) = G_n^\lambda(x') = G_n^\lambda(2(\frac{x-a}{b-a})^\alpha - 1), \quad (5.21)$$

where $\alpha \in \mathbb{R}^+$ is the "fractional order of function" which is determined with respect to the context and $\lambda > -\frac{1}{2}$ is Gegenbauer polynomial order.

Regarding recursive relation which is already introduced for Gegenbauer polynomials Eq. 5.7, one can define the recursive relation for fractional Gegenbauer functions simply by inserting mapping $x' = 2(\frac{x-a}{b-a})^\alpha - 1$ into Eq. 5.7:

$$\begin{aligned} FG_n^{\alpha, \lambda}(x) &= \frac{1}{n}[2x(n+\lambda-1)FG_{n-1}^{\alpha, \lambda}(x) - (n+2\lambda-2)FG_{n-2}^{\alpha, \lambda}(x)], \\ FG_0^{\alpha, \lambda}(x) &= 1, \quad FG_1^{\alpha, \lambda}(x) = 2\lambda(2(\frac{x-a}{b-a})^\alpha - 1). \end{aligned} \quad (5.22)$$

Consequently, first few orders of *Gegenbauer polynomials of fractional order* are:

$$\begin{aligned} FG_0^{\alpha, \lambda}(x) &= 1, \\ FG_1^{\alpha, \lambda}(x) &= 4\lambda(\frac{x-a}{b-a})^\alpha - 2\lambda, \\ FG_2^{\alpha, \lambda}(x) &= 8\lambda^2(\frac{x-a}{b-a})^{2\alpha} - 8\lambda^2(\frac{x-a}{b-a})^\alpha + 2\lambda^2 + 8\lambda(\frac{x-a}{b-a})^{2\alpha} - 8\lambda(\frac{x-a}{b-a})^\alpha + \lambda. \end{aligned}$$

For higher orders there exist many terms which makes it impossible to write so for sake of convenience one can use the below Python code to generate any order of *Gegenbauer polynomials of fractional order*:

Program Code

```
import sympy
x = sympy.Symbol("x")
a = sympy.Symbol("a")
b = sympy.Symbol("b")
lambd = sympy.Symbol(r'\lambda')
alpha = sympy.Symbol(r'\alpha')
x=sympy.sympify(1-2*(x-a/b-a)**alpha)

def FGn(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return 2*lambd*x
    elif n >= 2:
        return (sympy.Rational(1,n)*(2*x*(n+lambd-1)*FGn(x,n-1))\
            -(n+2*lambd-2)*FGn(x,n-2)))
```

For example, the 3rd order can be generated as follow :

Program Code

```
sympy.expand(sympy.simplify(FGn(x,2)))
```

$$> 8\lambda^2 \left(\frac{x-a}{b-a}\right)^{2\alpha} - 8\lambda^2 \left(\frac{x-a}{b-a}\right)^\alpha + 2\lambda^2 + 8\lambda \left(\frac{x-a}{b-a}\right)^{2\alpha} - 8\lambda \left(\frac{x-a}{b-a}\right)^\alpha + \lambda$$

To define a generating function for *fractional Gegenbauer polynomials*, one can follow the same process as already done for the recursive relation of *Gegenbauer polynomials* in Eq. 5.22, by replacing the transition equation ($x' = 2\left(\frac{x-a}{b-a}\right)^\alpha - 1$) with the x parameter in Eq. 5.5, then rewriting the equation for *fractional Gegenbauer polynomial* $FG_n^{\alpha,\lambda}(x)$:

$$FG_n^{\alpha,\lambda}(x) = G_n^\lambda(x') = \frac{1}{(1 - 2(2\left(\frac{x-a}{b-a}\right)^\alpha - 1)z + z^2)^\lambda}. \quad (5.23)$$

Similarly, the weight function to which the *fractional Gegenbauer functions* are orthogonal is simple to define. Considering the weight function as Eq. 5.2 and the transition $x' = 2\left(\frac{x-a}{b-a}\right)^\alpha - 1$, we have:

$$w^{\alpha,\lambda}(x') = 2\left(\frac{x-a}{b-a}\right)^{\alpha-1} \left(4\left(\frac{x-a}{b-a}\right)^\alpha - 4\left(\frac{x-a}{b-a}\right)^{2\alpha}\right)^{\lambda-\frac{1}{2}}. \quad (5.24)$$

The *fractional Gegenbauer polynomials* are orthogonal over a finite interval respecting weight function in Eq. 5.24, therefore one can define the orthogonality relation as [47]:

$$\begin{aligned} \int_{-1}^1 G_n^\lambda(x') G_m^\lambda(x') w(x') dx &= \int_0^1 FG_n^{\alpha,\lambda}(x) FG_m^{\alpha,\lambda}(x) w^{\alpha,\lambda}(x) dx \\ &= \frac{2^{1-4\lambda} \pi \Gamma(2\lambda + m)}{(\lambda + m)m! \Gamma^2(\lambda)} \delta_{mn}. \end{aligned} \quad (5.25)$$

where δ_{nm} is the Kronecker delta function.

5.3 Gegenbauer kernel functions

In this section, the ordinary Gegenbauer kernel function has been covered considering addressing annihilation and explosion problems and proving the validity of such a kernel. Furthermore, the generalized Gegenbauer kernel function and also the fractional form of the Gegenbauer kernel function have been covered and its validation has been proved according to the Mercer theorem.

5.3.1 Ordinary Gegenbauer kernel function

Gegenbauer polynomials have been used as a kernel function due to less need to support vectors while acquiring high accuracy in comparison with well-known kernel functions such as RBF or many other classical and orthogonal kernels [25]. That is

because Gegenbauer polynomials, same as any other orthogonal polynomial kernels produce kernel matrices with lower significant eigenvalues which in turn means fewer support vectors are needed [25].

As we know, a multidimensional SVM kernel function can be defined as:

$$K(X, Z) = \prod_{j=0}^d K_j(x_j, z_j). \quad (5.26)$$

It can be seen that produce two undesired results, ***annihilation effect***, when any or both of x_j and z_j of $k(x_j, z_j)$ is/are close to zero, then kernel outputs very small values, and the second one is considered as ***explosion effect*** which refers to very big output of the kernel $|\prod_{j=1}^d K_j(x_j, z_j)| \rightarrow \infty$ which leads to numerical difficulties [25]. To overcome ***annihilation*** and ***explosion*** effects, Padierna et al. [25] proposed a new formulation for SVM kernel:

$$k(X, Z) = \langle \phi(X), \phi(Z) \rangle = \prod_{j=1}^d \sum_{i=0}^n p_i(x_j) p_i(z_j) w(x_j, z_j) u(p_i)^2, \quad (5.27)$$

where $w(x_j, z_j)$ is the weight function, the scaling function is $u(p_i) \equiv \beta \|p_i(\cdot)\|_{max}^{-1} \in \mathbb{R}^+$, which is the reciprocal of the maximum absolute value of an i th degree polynomial within its operational range, and β is a convenient positive scalar factor [25].

The Gegenbauer polynomials G_n^λ can be classified into two groups considering the value of λ :

1. $-0.5 < \lambda \leq 0.5$

As it is clear in Figures 5.1,5.2, the amplitude is $-1 \leq G_n^\lambda(x) \leq 1$, so a scaling function is not required and also the weight function for this group is equal to 1 [25].

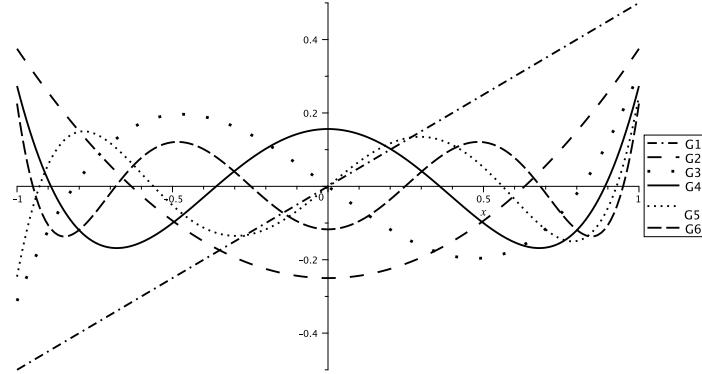


Fig. 5.1: Gegenbauer polynomials with $\lambda = 0.25$ in the positive range of $\lambda \in (-0.5, 0.5]$

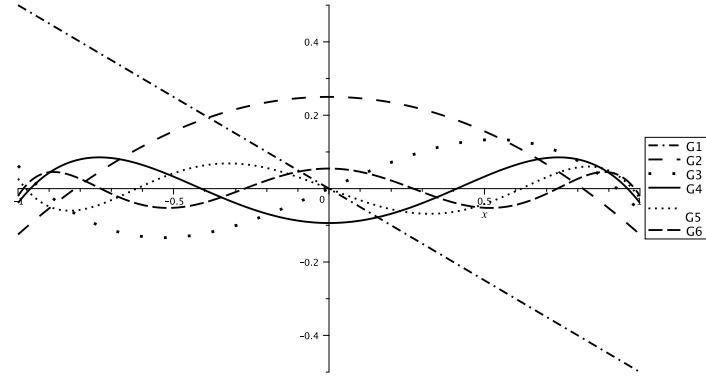


Fig. 5.2: Gegenbauer polynomials with $\lambda = -0.25$ in the negative range of $\lambda \in (-0.5, 0.5]$

2. $\lambda > 0.5$

In this group of Gegenbauer polynomials, the amplitude may grow way bigger than 1, thereby it is probable to face an *explosion effect* [25], as mentioned before a weight function and scaling function come to play (see Figures 5.3, 5.4, 5.5). In this case, the weight function for Gegenbauer polynomials is:

$$w_\lambda(x) = (1 - x^2)^{\lambda - \frac{1}{2}}. \quad (5.28)$$

According to formulation in Eq. 5.27, the scaling function is $u(p_i) \equiv \beta |p_i(.)|_{max}^{-1} \in \mathbb{R}^+$ and considering for Gegenbauer polynomials, the maximum amplitudes are reached at $x = \pm 1$. Thereupon, by means of **Pochhammer operator** $(a)_n \equiv a(a+1)(a+2)\dots(a+n-1)$, where $\lambda > 0.5$ and $i > 0$, Padierna et al. [25] proposed the scaling function. By setting $\beta = \frac{1}{\sqrt{n+1}}$, we have:

$$u(p_i) = u(G_i^\lambda) = (\sqrt{n+1} |G_i^\lambda(1)|)^{-1}. \quad (5.29)$$

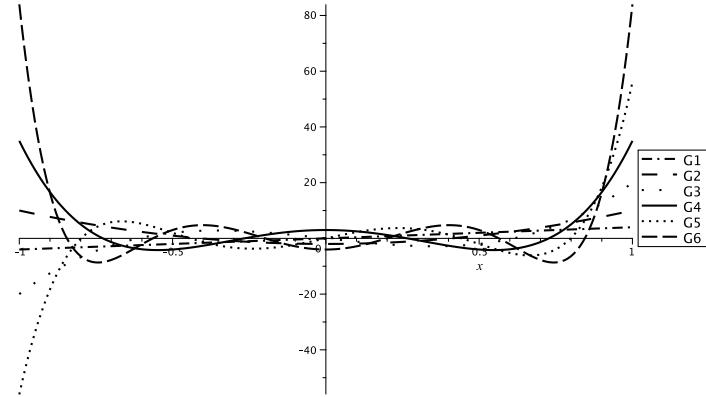


Fig. 5.3: The original polynomials in the second group, where $\lambda = 2$

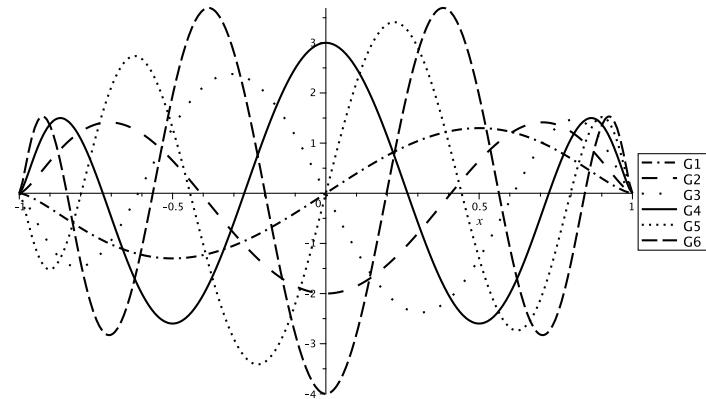


Fig. 5.4: The weighted Gegenbauer polynomials in the second group, where $\lambda = 2$

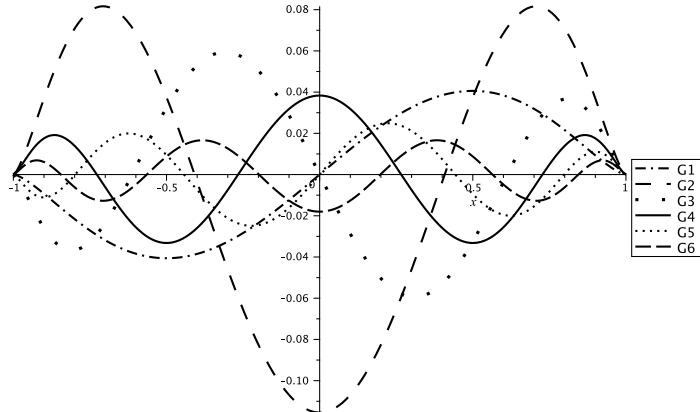


Fig. 5.5: the weighted-and-scaled Gegenbauer polynomials in the second group, where $\lambda = 2$

The weight function introduced in Eq. 5.28 is the univariate weight function, meaning that is the weight function for one variable. To use the Gegenbauer polynomials kernel function a bivariate weight function need, in which one can define a product of the corresponding univariate weight functions [48]:

$$w_\lambda(x, z) = ((1 - x^2)(1 - z^2))^{\lambda - \frac{1}{2}} + \varepsilon. \quad (5.30)$$

The ε term in Eq. 5.30 is to prevent weight function from getting zero at values on the border in the second group of Gegenbauer polynomials (Figures 5.4 and 5.5) by adding a small value to the output (e.g. $\varepsilon = 0.01$) [25].

Using relations in Eq. 5.27, Eq. 5.29, and Eq. 5.30, one can define the Gegenbauer kernel function as:

$$K_{Geg}(X, Z) = \prod_{j=1}^d \sum_{i=0}^n G_i^\lambda(x_j) G_j^\lambda(z_j) w_\lambda(x_j, z_j) u(G_j^\lambda)^2. \quad (5.31)$$

5.3.2 Validation of Gegenbauer kernel function

Theorem 5.1 [25] *Gegenbauer kernel function introduced at Eq. 5.31 is a valid Mercer kernel.*

Proof A valid kernel function needs to be positive semi-definite or equivalently must satisfy the necessary and sufficient conditions of *Mercer's theorem* introduced in 2.2.1. Furthermore, positive semi-definiteness property ensures that the optimization problem of the SVM algorithm can be solved with **Convex Optimization Programming**[26]. According to *Mercer theorem*, a symmetric and continuous function

$K(X, Z)$ is a kernel function if satisfies:

$$\iint K(x, z)f(x)f(z)dx dz \geq 0. \quad (5.32)$$

Bearing in mind that $K_{Geg}(x, z) = \prod_{j=1}^d K_{Geg}(x_j, z_j)$, denoting the multidimensionality of the Gegenbauer kernel function, consequently the Gegenbauer for scalar x and z is:

$$K_{Geg}(x, z) = \sum_{i=0}^n G_i^\lambda(x)G_i^\lambda(z)w_\lambda(x, z)u(G_i^\lambda)^2. \quad (5.33)$$

By substitution of Eq. 5.33 into Eq. 5.32, we conclude:

$$\begin{aligned} & \iint K_{Geg}(x, z)g(x)g(z)dx dz = \\ & \iint \sum_{i=0}^n G_i^\lambda(x)G_i^\lambda(z)w_\lambda(x, z)u(G_i^\lambda)^2 g(x)g(z)dx dz. \end{aligned} \quad (5.34)$$

Also, by inserting weight function from Eq. 5.30 into Eq. 5.34, we have:

$$= \iint \sum_{i=0}^n G_i^\lambda(x)G_i^\lambda(z) \left[(1-x^2)^{\lambda-\frac{1}{2}}(1-z^2)^{\lambda-\frac{1}{2}} + \varepsilon \right] u(G_i^\lambda)^2 g(x)g(z)dx dz. \quad (5.35)$$

It should be noted that $u(G_i^\lambda)$ is always positive and independent of the data, so:

$$\begin{aligned} & = \sum_{i=0}^n u(G_i^\lambda)^2 \iint G_i^\lambda(x)G_i^\lambda(z)(1-x^2)^{\lambda-\frac{1}{2}}(1-z^2)^{\lambda-\frac{1}{2}} g(x)g(z)dx dz, \\ & \quad + \sum_{i=0}^n \varepsilon u(G_i^\lambda)^2 \iint G_i^\lambda(x)G_i^\lambda(z)g(x)g(z)dx dz, \quad (5.36) \\ & = \sum_{i=0}^n u(G_i^\lambda)^2 \int G_i^\lambda(x)(1-x^2)^{\lambda-\frac{1}{2}}g(x)dx \int G_i^\lambda(z)(1-z^2)^{\lambda-\frac{1}{2}}g(z)dz, \\ & \quad + \sum_{i=0}^n \varepsilon u(G_i^\lambda)^2 \int G_i^\lambda(x)g(x)dx \int G_i^\lambda(z)g(z)dz, \\ & = \sum_{i=0}^n u(G_i^\lambda)^2 \left(\int G_i^\lambda(x)(1-x^2)^{\lambda-\frac{1}{2}}g(x)dx \right)^2 + \sum_{i=0}^n \varepsilon u(G_i^\lambda)^2 \left(\int G_i^\lambda(x)g(x)dx \right)^2 \geq 0. \end{aligned}$$

Therefore, $K(x, z)$ is a valid kernel. Considering the product of two kernels is a kernel too, then $K_{Geg}(X, Z) = \prod_{j=1}^d K_{Geg}(x_j, z_j)$ for the vectors x and z is also a valid kernel. consequently, one can deduce that the Gegenbauer kernel function in Eq. 5.31 is a valid kernel too. \square

5.3.3 Other Gegenbauer kernel functions

In this section, the generalized Gegenbauer kernel has been introduced which was recently proposed by Yang et al. [49].

5.3.3.1 Generalized Gegenbauer kernel

The Generalized Gegenbauer kernel (GGK) is constructed by using the partial sum of the inner products of generalized Gegenbauer polynomials. The generalized Gegenbauer polynomials have the recursive relation as follows [49]:

$$\begin{aligned} GG_0^\lambda(x) &= 1, \\ GG_1^\lambda(x) &= 2\lambda x, \\ GG_n^\lambda(x) &= \frac{1}{d}[2x(d + \lambda - 1)GG_{n-1}^\lambda(x) - (n + 2\lambda - 2)GG_{n-2}^\lambda(x)]. \end{aligned} \quad (5.37)$$

where $x \in \mathbb{R}^n$ denotes the vector of input variables. The output of generalized Gegenbauer polynomial $GG_n^\lambda(x)$ is scalar for even orders of n and is a vector for odd orders of n .

Then Yang et al. [49], proposed generalized Gegenbauer kernel function $K_{GG}(x_i, x_j)$ for order n of two input vectors x_i and x_j as:

$$K_{GG}(x_i, x_j) = \frac{\sum_{l=0}^n GG_l^\lambda(x_i)^T GG_l^\lambda(x_j)}{\exp(\sigma \|x_i - x_j\|_2^2)}, \quad (5.38)$$

where each element of x_i and x_j is normalized in the range [-1, 1]. In this context, both α and σ are considered as the kernel scales or the so-called decaying parameters of the proposed kernel function.

Theorem 5.2 [25] *The proposed GGK is a valid Mercer kernel.*

Proof The proposed GGK can be alternatively formulated as the product of two kernel functions such that :

$$\begin{aligned} K_1(x_i, x_j) &= \exp(-\sigma \|x_i - x_j\|_2^2), \\ K_2(x_i, x_j) &= \sum_{l=0}^d GG_l^\alpha(x_i)^T GG_l^\alpha(x_j), \\ K_{GG}(x_i, x_j) &= K_1(x_i, x_j)K_2(x_i, x_j). \end{aligned} \quad (5.39)$$

As already been discussed in, 2.2.1, the multiplication of two valid Mercer Kernels is also a valid kernel function. Since that $K_1(x_i, x_j)$ is the Gaussian kernel ($\sigma > 0$) which satisfies the Mercer theorem, $K_{GG}(x_i, x_j)$ can be proved as a valid kernel by verifying that $K_2(x_i, x_j)$ satisfies the Mercer theorem. Given an arbitrary squared

integrable function $g(x)$ defined as $g : \mathbb{R}^n \rightarrow \mathbb{R}$ and assuming each element in x_i and x_j are independent with each other, we can conclude that

$$\begin{aligned}
& \iint K_2(x_i, x_j)g(x_i)g(x_j)dx_i dx_j = \\
&= \iint \sum_{l=0}^n GG_l^\lambda(x_i)^T GG_l^\lambda(x_j)g(x_i)g(x_j)dx_i dx_j, \\
&= \sum_{l=0}^n \iint GG_l^\lambda(x_i)^T GG_l^\lambda(x_j)g(x_i)g(x_j)dx_i dx_j, \\
&= \sum_{l=0}^n \left[\int GG_l^\lambda(x_i)^T g(x_i)dx_i \int GG_l^\lambda(x_j)g(x_j)dx_j \right], \\
&= \sum_{l=0}^n \left[\int GG_l^\lambda(x_i)^T g(x_i)dx_i \right] \left[\int GG_l^\lambda(x_j)g(x_j)dx_j \right] \geq 0.
\end{aligned} \tag{5.40}$$

Thus, $K_2(x_i, x_j)$ is a valid Mercer kernel, and it can be concluded that, the proposed GGK $K_{GG}(x_i, x_j)$ is an admissible Mercer kernel function. \square

5.3.4 Fractional Gegenbauer kernel function

Similar to the latest kernel function introduced at Eq. 5.31 as the Gegenbauer kernel, and the weight function introduced at Eq. 5.28 as the fractional weight function, the fractional Gegegnbauer kernel has been introduced. First, the bivariate form of the fractional weight function has to be defined that its approach is again similar to the previous corresponding definition at Eq. 5.30, i.e.:

$$fw^{\alpha, \lambda}(x, z) = ((1 - (\frac{x-a}{b-a})^{2\alpha})(1 - (\frac{z-a}{b-a})^{2\alpha}))^{\lambda-\frac{1}{2}} + \epsilon. \tag{5.41}$$

Therefore, the *fractional Gegenbauer kernel* function is:

$$K_{FGeG}(X, Z) = \prod_{j=1}^d \sum_{i=0}^n G_i^\lambda(x'_{x_j}) G_j^\lambda(x'_{z_j}) fw^{\alpha, \lambda}(x'_{x_j}, x'_{z_j}) u(G_j^\lambda)^2. \tag{5.42}$$

Theorem 5.3 *Gegenbauer kernel function introduced at Eq. 5.42 is a valid Mercer kernel.*

Proof According to the Mercer theorem, a valid kernel must satisfy sufficient conditions of Mercer's theorem. As Mercer theorem states, any SVM kernel to be a valid kernel must be non-negative, in a precise way:

$$\iint K(x, z)w(x, z)f(x)f(z)dxdz \geq 0. \quad (5.43)$$

By the fact that here the kernel is as Eq. 5.42, it can be seen with a simple replacement that:

$$\begin{aligned} & \iint K_{FGeG}(x, z)g(x)g(z)dxdz = \\ & \iint \sum_{i=0}^n G_i^\lambda(x'_{x_j})G_j^\lambda(x'_{z_j})f_{W\lambda}(x'_{x_j}, x'_{z_j})u(G_j^\lambda)^2 g(x)g(z)dxdz. \end{aligned} \quad (5.44)$$

In the last equation, the fractional bivariate weight function (i.e., Eq. 5.41) is considered, so we have:

$$\iint \sum_{i=0}^n G_i^\lambda(x'_{x_j})G_j^\lambda(x'_{z_j}) \left[\left(\left(1 - \left(\frac{x-a}{b-a} \right)^{2\alpha} \right) \left(1 - \left(\frac{z-a}{b-a} \right)^{2\alpha} \right) \right)^{\lambda-\frac{1}{2}} + \epsilon \right] u(G_j^\lambda)^2 g(x)g(z)dxdz. \quad (5.45)$$

Note that $u(G_i^\lambda)$ is always positive and independent of the data, therefore:

$$\begin{aligned} & = \sum_{i=0}^n u(G_i^\lambda)^2 \iint \sum_{i=0}^n G_i^\lambda(x'_{x_j})G_j^\lambda(x'_{z_j}) \left(\left(1 - \left(\frac{x-a}{b-a} \right)^{2\alpha} \right) \left(1 - \left(\frac{z-a}{b-a} \right)^{2\alpha} \right) \right)^{\lambda-\frac{1}{2}} g(x)g(z)dxdz \\ & + \sum_{i=0}^n \epsilon u(G_i^\lambda)^2 \iint G_i^\lambda(x'_{x_j})G_j^\lambda(x'_{z_j})g(x)g(z)dxdz. \end{aligned} \quad (5.46)$$

□

5.4 Application of Gegenbauer kernel functions on real datasets

In this section, the result of *Gegenbauer* and *fractional Gegenbauer* kernels have been compared on some data sets, with other well-known kernels such as *RBF*, *polynomial*, and also *Chebyshev* and *fractional Chebyshev* kernels, introduced in the previous chapters. In order to have a neat classification, there may need to be some preprocessing steps according to the data set. Here, these steps have not been focused on, but they are mandatory when using *Gegenbauer polynomials* as the kernel. For this section, two data sets have been selected, which are well-known and helpful for machine learning practitioners.

5.4.1 Spiral dataset

The spiral dataset as already has been introduced in chapter 3, is one of the well-known multi-class classification tasks. Using the **OVA** method, this multi-class classification data set has been split into three binary classification datasets and applies the SVM Gegenbauer kernel. Fig. 5.6 depicts the data points in normal and fractional space of the spiral dataset.

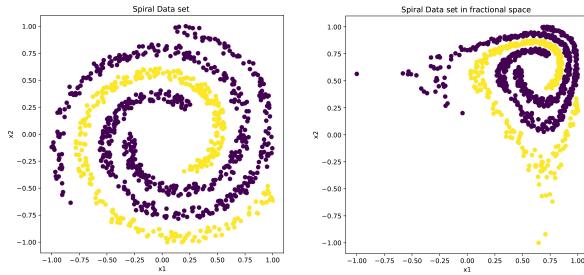


Fig. 5.6: Spiral dataset, 1000 data points

Despite the data density in the spiral data set fraction mode, using more features (three dimensions or more) it can be seen that the Gegenbauer kernel can display data classification separately more clearly and simply (see Fig. 5.7).

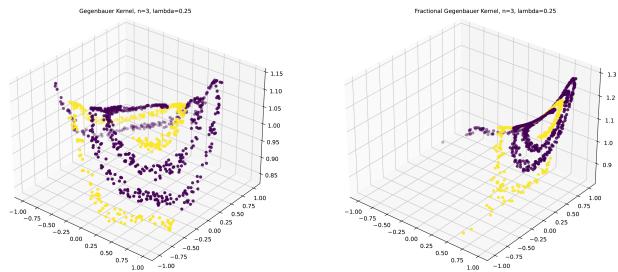


Fig. 5.7: 3D Gegenbauer spiral dataset

Also, Fig. 5.8 shows how the classifiers of the Gegenbauer kernel for different orders [3,4,5,6] and $\lambda = 0.6$ have chosen the boundaries. Similarly, Fig. 5.6 depicts the data points of the spiral dataset after transforming into fractional space of order $\alpha = 0.5$. Thereby, Fig. 5.9 depicts the decision boundaries of relevant fractional Gegenbauer kernel, where $\alpha = 0.5$ and $\lambda = 0.6$.

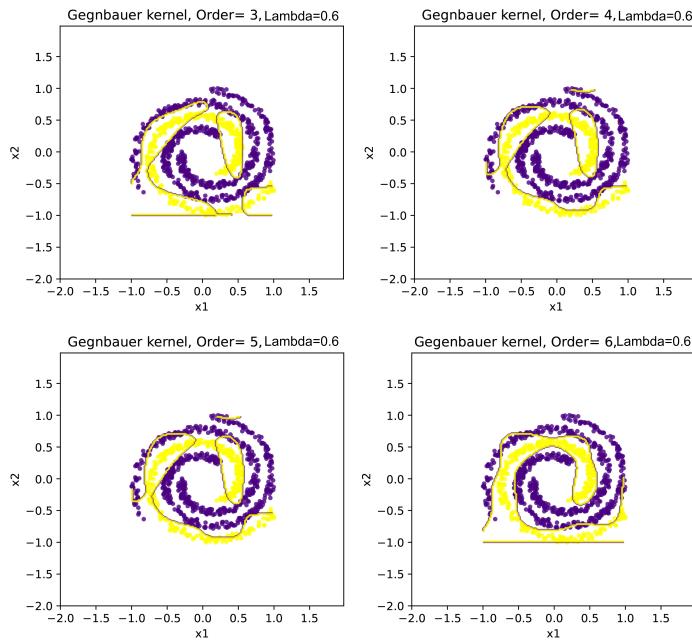


Fig. 5.8: Gegenbauer kernel with orders of 3, 4, 5, and 6 on Spiral dataset with $\lambda = 0.6$

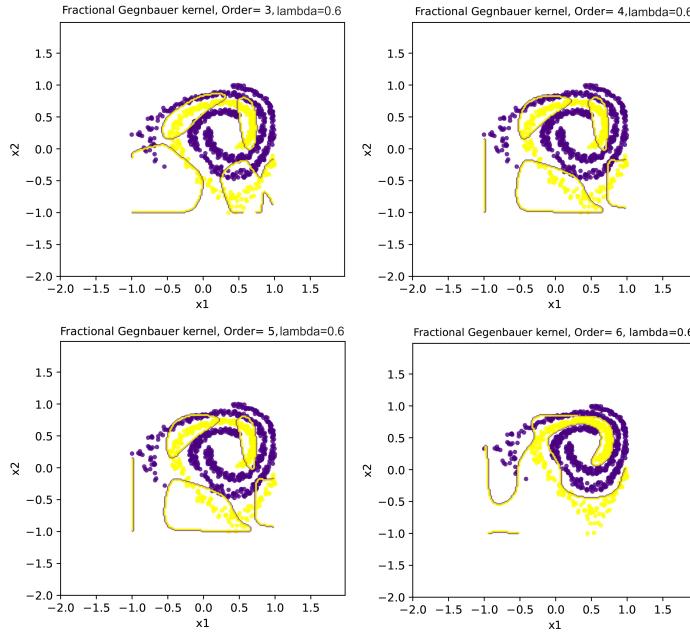


Fig. 5.9: Fractional Gegenbauer kernel with orders of 3, 4, 5, and 6 on Spiral dataset with $\alpha = 0.5$ and $\lambda = 0.6$

On the other hand, the following tables provide a comparison of the experiments on the spiral dataset. Three possible binary classifications have been examined according to the One-vs-All method. As it is clear from Table 5.2, **fractional Legendre** kernel shows better performance for the class 1-vs-[2,3]. However, for other binary classifications on this spiral dataset, according to results in Tables 5.3 and 5.4, RBF kernels outperform fractional kinds of Legendre kernels.

Table 5.2: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, and fractional Gegenbauer kernels functions on Spiral dataset

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 0.73 | - | - | - | - | 0.97 |
| Polynomial | - | 8 | - | - | - | 0.9533 |
| Chebyshev | - | - | 5 | - | - | 0.9667 |
| Fractional Chebyshev | - | - | 3 | 0.3 | - | 0.9733 |
| Legendre | - | - | 7 | - | - | 0.9706 |
| Fractional Legendre | - | - | 7 | 0.4 | - | 0.9986 |
| Gegenbauer | - | - | 6 | - | 0.3 | 0.9456 |
| Fractional Gegenbauer | - | - | 6 | 0.3 | 0.7 | 0.9533 |

Table 5.3: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, and fractional Gegenbauer kernels on Spiral dataset

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 0.1 | - | - | - | - | 0.9867 |
| Polynomial | - | 5 | - | - | - | 0.9044 |
| Chebyshev | - | - | 6 | - | - | 0.9289 |
| Fractional Chebyshev | - | - | 6 | 0.8 | - | 0.9344 |
| Legendre | - | - | 8 | - | - | 0.9773 |
| Fractional Legendre | - | - | 8 | 0.4 | - | 0.9853 |
| Gegenbauer | - | - | 5 | - | 0.3 | 0.9278 |
| Fractional Gegenbauer | - | - | 4 | 0.6 | 0.6 | 0.9356 |

Table 5.4: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, and fractional Gegenbauer kernels on Spiral dataset

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 0.73 | - | - | - | - | 0.9856 |
| Polynomial | - | 5 | - | - | - | 0.9856 |
| Chebyshev | - | - | 6 | - | - | 0.9622 |
| Fractional Chebyshev | - | - | 6 | 0.6 | - | 0.9578 |
| Legendre | - | - | 7 | - | - | 0.9066 |
| Fractional Legendre | - | - | 5 | 0.4 | - | 0.9906 |
| Gegenbauer | - | - | 6 | - | 0.3 | 0.9611 |
| Fractional Gegenbauer | - | - | 6 | 0.9 | 0.3 | 0.9644 |

5.4.2 Three Monks's dataset

Another problem in point is the three Monks problem, which is addressed here (see Chapter 3 for more information about the dataset). We applied the *Gegenbauer kernel* introduced at Eq. 5.31 on the datasets from *the three monks's problem* and append the result to Tables 5.5, 5.6, and 5.7. It can be seen that *fractional Gegenbauer kernel* shows strong performance on these datasets, specifically on the first problem which had **100 %** accuracy, and also on the third data set, both kinds of *Gegenbauer kernels* have the best accuracy among all kernels under comparison. Tables 5.5, 5.6, and 5.7 illustrate the details of these comparisons.

Table 5.5: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, fractional Legendre, Gegenbauer, and fractional Gegenbauer kernels on Monk's first problem. It is shown that the Gengenbauer kernel outperforms others, and also the fractional Gegenbauer and fractional Legendre have the most desirable accuracy which is 1

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 2.844 | - | - | - | - | 0.8819 |
| Polynomial | - | 3 | - | - | - | 0.8681 |
| Chebyshev | - | - | 3 | - | - | 0.8472 |
| Fractional Chebyshev | - | - | 3 | 1/16 | - | 0.8588 |
| Legendre | - | - | 4 | - | - | 0.8333 |
| Fractional Legendre | - | - | 4 | 0.1 | - | 0.8518 |
| Gegenbauer | - | - | 3 | - | -0.2 | 0.9931 |
| Fractional Gegenbauer | - | - | 3 | 0.7 | 0.2 | 1 |

Table 5.6: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, fractional Legendre, Gegenbauer, and fractional Gegenbauer kernels on The Monk's second problem. It is shown that the fractional Chebyshev kernel has the second-best result, following the fractional Legendre kernel

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 5.5896 | - | - | - | - | 0.875 |
| Polynomial | - | 3 | - | - | - | 0.8657 |
| Chebyshev | - | - | 3 | - | - | 0.8426 |
| Fractional Chebyshev | - | - | 3 | 1/16 | - | 0.9653 |
| Legendre | - | - | 3 | - | - | 0.8032 |
| Fractional Legendre | - | - | 3 | 0.1 | - | 1 |
| Gegenbauer | - | - | 3 | - | 0.5 | 0.7824 |
| Fractional Gegenbauer | - | - | 3 | 0.1 | 0.5 | 0.9514 |

Table 5.7: Comparison of RBF, Polynomial, Chebyshev, fractional Chebyshev, Legendre, fractional Legendre, Gegenbauer, and fractional Gegenbauer kernels on The Monk's third problem. Note that the Gegenbauer and also fractional Gegenbauer kernels have the best result

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-----------------|
| RBF | 2.1586 | - | - | - | - | 0.91 |
| Polynomial | - | 3 | - | - | - | 0.875 |
| Chebyshev | - | - | 6 | - | - | 0.895 |
| Fractional Chebyshev | - | - | 5 | 1/5 | - | 0.91 |
| Legendre | - | - | 4 | - | - | 0.8472 |
| Fractional Legendre | - | - | 3 | 0.8 | - | 0.8379 |
| Gegenbauer | - | - | 4 | - | -0.2 | 0.9259 |
| Fractional Gegenbauer | - | - | 3 | 0.7 | -0.2 | 0.9213 |

5.5 Conclusion

Gegenbauer (ultra-spherical) orthogonal polynomials are very important among orthogonal polynomials since have been used to address many differential equational problems, specifically in spherical spaces as its name suggests. In this chapter, a brief background on the research history and explained the basics and properties of these polynomials, and constructed the related kernel function have been reviewed. Moreover, the novel method of the fractional Gegenbauer orthogonal polynomials

and the corresponding kernels have been introduced, and in the last section, how to successfully use this new kernel through an experiment on well-known datasets in kernel-based learning algorithms such as SVM has been depicted.

References

1. Avery, J.S.: Hyperspherical harmonics applications in quantum theory (Vol. 5) Springer Science & Business Media, Berlin (2012)
2. Doha, E.H.: The ultraspherical coefficients of the moments of a general-order derivative of an infinitely differentiable function. *J. Comput. Appl. Math.* **89**, 53–72 (1998)
3. Elliott, D.: The expansion of functions in ultraspherical polynomials. *J. Aust. Math. Soc.* **1**, 428–438 (1960)
4. Arfaoui, S., Ben Mabrouk, A., Cattani, C.: New type of Gegenbauer–Hermite monogenic polynomials and associated Clifford wavelets. *J Math Imaging Vis* **62**, 73–97 (2020)
5. Stier, A.C., Goth, W., Hurley, A., Feng, X., Zhang, Y., Lopes, F.C., Sebastian, K.R., Fox, M.C., Reichenberg, J.S., Markey, M.K., Tunnell, J.W.: Machine learning and the Gegenbauer kernel improve mapping of sub-diffuse optical properties in the spatial frequency domain. In *Molecular-Guided Surgery: Molecules, Devices, and Applications VII* **11625**, 1162509 (2021)
6. Soufivand, F., Soltanian, F., Mamehrashi, K.: An Operational Matrix Method Based on the Gegenbauer Polynomials for Solving a Class of Fractional Optimal Control Problems. *Int. j. ind. electron.* **4**, 475–484 (2021)
7. Park, R. W.: Optimal compression and numerical stability for Gegenbauer reconstructions with applications. Arizona State University (2009)
8. Feng, B. Y., Varshney, A.: SIGNET: Efficient Neural Representation for Light Fields. In *Proceedings of the IEEE/CVF* (2021)
9. Liao, S., Chiang, A., Lu, Q., Pawlak, M.: Chinese character recognition via Gegenbauer moments. In *Object recognition supported by user interaction for service robots* **3**, 485–488 (2002)
10. Liao, S., Chen, J.: Object recognition with lower order gegenbauer moments. *Lecture Notes on Software Engineering*, **1**, 387 (2013)
11. Herrera-Acosta, A., Rojas-Domínguez, A., Carpio, J. M., Ornelas-Rodríguez, M., Puga, H.: Gegenbauer-Based Image Descriptors for Visual Scene Recognition. In *Intuitionistic and Type-2 Fuzzy Logic Enhancements in Neural and Optimization Algorithms: Theory and Applications*, 629–643 (2020)
12. Hjouji, A., Bouikhalene, B., EL-Mekkaoui, J., Qjidaa, H.: New set of adapted Gegenbauer–Chebyshev invariant moments for image recognition and classification. *J Supercomput* **77**, 5637–5667 (2021)
13. Eassa, M., Selim, I. M., Dabour, W., Elkafrawy, P.: Automated detection and classification of galaxies based on their brightness patterns. *Alex. Eng. J.* **61**, 1145–1158 (2022)
14. Tamandani, A., Alijani, M. G.: Development of an analytical method for pattern synthesizing of linear and planar arrays with optimal parameters. *Int J Electron Commun* **146**, 154135 (2022)
15. law Pawlak, M.: Image analysis by moments: reconstruction and computational aspects. Oficyna Wydawnicza Politechniki Wrocławskiej (2006)
16. Hosny, K. M.: Image representation using accurate orthogonal Gegenbauer moments. *Pattern Recognit. Lett.* **32**, 795–804 (2011)
17. Hosny, K. M., Darwish, M. M., Eltoukhy, M. M. : New fractional-order shifted Gegenbauer moments for image analysis and recognition. *J. Adv. Res.* **25**, 57–66 (2020)
18. Hosny, K. M. : New set of Gegenbauer moment invariants for pattern recognition applications. *Arab J Sci Eng* **39**, 7097–7107 (2014)

19. Abd Elaziz, M., Hosny, K. M., Selim, I. M. : Galaxies image classification using artificial bee colony based on orthogonal Gegenbauer moments. *Soft Comput.* **23**, 9573–9583 (2019)
20. Öztürk, S., Ahmad, R., Akhtar, N.: Variants of Artificial Bee Colony algorithm and its applications in medical image processing. *Appl. Soft Comput.* **97**, 106799 (2020)
21. Langley, J., Zhao, Q. : A model-based 3D phase unwrapping algorithm using Gegenbauer polynomials. *Phys. Med. Biol.* **54**, 5237–5252 (2009)
22. Ludlow, I. K., Everitt, J. : Application of Gegenbauer analysis to light scattering from spheres: theory. *Phys. Rev. E* **51**, 2516–2526 (1995)
23. Brackx, F., De Schepper, N., Sommen, F.: The Clifford–Gegenbauer polynomials and the associated continuous wavelet transform. *Integral Transform Spec Funct* **15**, 387–404 (2004)
24. Ilić, A. D., Pavlović, V. D. : New class of filter functions generated most directly by Christoffel-Darboux formula for Gegenbauer orthogonal polynomials. *Int. J. Electron.* **98**, 61–79 (2011)
25. Padierna, L. C., Carpio, M., Rojas-Dominguez, A., Puga, H., Fraire, H. : A novel formulation of orthogonal polynomial kernel functions for SVM classifiers: the Gegenbauer family. *Pattern Recognit.* **84**, 211–225 (2018)
26. Wu, Q., Zhou, D. X.: SVM soft margin classifiers: linear programming versus quadratic programming. *Neural Comput.* **17**, 1160–1187 (2005)
27. Padierna, L. C., Amador-Medina, L. F., Murillo-Ortiz, B. O., Villaseñor-Mora, C. : Classification method of peripheral arterial disease in patients with type 2 diabetes mellitus by infrared thermography and machine learning. *Infrared Phys Technol* **111**, 103531 (2020)
28. Azari, A. S., Mack, Y. P., Müller, H. G.: Ultraspherical polynomial, kernel and hybrid estimators for non parametric regression. *Sankhya: Indian J. Stat.* 80–96 (1992)
29. Feng, J., Liu, L., Wu, D., Li, G., Beer, M., Gao, W. : Dynamic reliability analysis using the extended support vector regression (X-SVR). *Mech Syst Signal Process* **126**, 368–391 (2019)
30. Ferrara, L., Guégan, D.: Forecasting with k-factor Gegenbauer processes: Theory and applications. *J. Forecast.* **20**, 581–601 (2001)
31. Asghari, M., Hadian Rasanan, A.H., Gorgin, S., Rahmati, D., Parand, K.: FPGA-orthopoly: a hardware implementation of orthogonal polynomials. *Eng Comput* (2022) doi: 10.1007/s00366-022-01612-x
32. Zhang, Z., He, J., Tang, L. : Two-Input Gegenbauer Orthogonal Neural Network with Growing-and-Pruning Weights and Structure Determination. In International Conference on Cognitive Systems and Signal Processing, 288–300 (2018)
33. He, J., Chen, T., Zhang, Z.: A Gegenbauer Neural Network with Regularized Weights Direct Determination for Classification. *arXiv preprint arXiv:1910.11552* (2019)
34. Srivastava, H. M., Shah, F. A., Abass, R. : An application of the Gegenbauer wavelet method for the numerical solution of the fractional Bagley-Torvik equation. *Russ. J. Math. Phys.* **26**, 77–93 (2019)
35. Parand, K., Dehghan, M., Baharifard, F.: Solving a laminar boundary layer equation with the rational Gegenbauer functions. *Appl. Math. Model.* **37**, 851–863 (2013)
36. Parand, K., Bahramnezhad, A., Farahani, H.: A numerical method based on rational Gegenbauer functions for solving boundary layer flow of a Powell–Eyring non-Newtonian fluid. *Comput. Appl. Math.* **37**, 6053–6075 (2018)
37. Parand, K., Delkhosh, M.: Solving Volterra’s population growth model of arbitrary order using the generalized fractional order of the Chebyshev functions. *Ricerche di Matematica*, **65**, 307–328 (2016)
38. Belmehdi, S.: Generalized Gegenbauer orthogonal polynomials. *J. Comput. Appl. Math.* **133**, 195–205 (2001)
39. Cohl, H. S. : On a generalization of the generating function for Gegenbauer polynomials. *Integral Transform Spec Funct* **24**, 807–816 (2013)
40. Liu, W., Wang, L. L.: Asymptotics of the generalized Gegenbauer functions of fractional degree. *J. Approx. Theory* **253**, 105378 (2020)
41. Yang, W., Zhousuo, Z., Hong, Y.: State recognition of bolted structures based on quasi-analytic wavelet packet transform and generalized Gegenbauer support vector machine. In 2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), 1–6 (2020)

42. Hadian-Rasanan, A. H., Nikarya, M., Bahramnezhad, A., Moayeri, M. M., Parand, K.: A comparison between pre-Newton and post-Newton approaches for solving a physical singular second-order boundary problem in the semi-infinite interval. arXiv preprint arXiv:1909.04066.
43. Doman, B. G. S.: The classical orthogonal polynomials. World Scientific, Singapore (2015)
44. Reimer, M.: Multivariate polynomial approximation. Springer Science & Business Media, Berlin (2003)
45. El-Kalaawy, A. A., Doha, E. H., Ezz-Eldien, S. S., Abdelkawy, M. A., Hafez, R. M., Amin, A. Z. M., Zaky, M. A. : A computationally efficient method for a class of fractional variational and optimal control problems using fractional Gegenbauer functions. Rom Rep Phys **70**, 90109 (2018)
46. Olver, F. WJ., Lozier, D. W., Boisvert, R. F., Clark, C. W.: NIST handbook of mathematical functions hardback and CD-ROM. Cambridge university press, Cambridge (2010)
47. Dehestani, H., Ordokhani, Y., Razzaghi, M. : Application of fractional Gegenbauer functions in variable-order fractional delay-type equations with non-singular kernel derivatives. Chaos Solitons Fractals **140**, 110111 (2020)
48. Dunkl, C. F., Yuan X.: Orthogonal polynomials of several variables. Cambridge University Press, Cambridge (2014)
49. Yang, W., Zhang, Z., Hong, Y.: State recognition of bolted structures based on quasi-analytic wavelet packet transform and generalized Gegenbauer support vector machine. In 2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC), 1–6 (2020)

Chapter 6

Fractional Jacobi Kernel Functions: Theory and Application

Amir Hosein Hadian Rasanan and Jamal Amani Rad and Malihe Shaban and Abdon Atangana

Abstract Orthogonality property in some kinds of polynomials has led to significant attention on them. Classical orthogonal polynomials have been under investigation for many years, with Jacobi polynomials being the most common. In particular, these polynomials are used to tackle multiple mathematical, physics, and engineering problems as well as their usage as the kernels has been investigated in the SVM algorithm classification problem. Here, by introducing the novel fractional form, a corresponding kernel is going to be proposed to extend the SVM's application. Through transforming the input data-set to a fractional space, the fractional Jacobi kernel-based SVM shows the excellent capability of solving non-linear problems in the classification task. In this chapter, the literature, basics, and properties of this family of polynomials will be reviewed and explain the corresponding kernel, introduce the fractional form of Jacobi polynomials and prove the validation according to Mercer conditions. Finally, a comparison of the obtained results over a well-known dataset will be provided, using the mentioned kernels with some other orthogonal kernels as well as RBF and polynomial kernels.

Amir Hosein Hadian Rasanan

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran, e-mail: amir.h.hadian@gmail.com

Jamal Amani Rad

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran, e-mail: j.amanirad@gmail.com

Maliheh Shaban Tameh

Department of Chemistry, University of Minnesota, Minneapolis, MN, 55455, USA e-mail: malihe.shaban@gmail.com

Abdon Atangana

Institute for Groundwater Studies, Faculty of Natural and Agricultural Sciences, University of the Free State, Bloemfontein 9300, South Africa e-mail: AtanganaA@ufs.ac.za

6.1 Introduction

Classical orthogonal functions have many applications in approximation theory [29, 28, 22] as well as in various numerical algorithms such as spectral methods [8, 10, 13]. The most general type of these orthogonal functions is Jacobi polynomials [22, 24, 30]. There are some special cases of this orthogonal polynomial family such as Legendre, the four kinds of Chebyshev, and Gegenbauer polynomials [1, 2]. The Jacobi polynomials have been used to solve various problems in different fields of science [3, 4, 5]. Ping et al. used Jacobi–Fourier moments for a deterministic image, and then they reconstructed the original image with the moments [7]. In 2013, Kazem [8] used Jacobi polynomials to solve linear and nonlinear fractional differential equations, which we know that the fractional differential equations have many use cases to show model physical and engineering processes. Shojaeizadeh et al. [9] used the shifted Jacobi polynomials to address the optimal control problem in advection-diffusion-reaction. Also, Bahravy et.al [10, 13, 39, 12] has many contributions to Jacobi applications. In [10], they proposed a shifted Jacobi-Gauss collocation method to solve the nonlinear Lane-Emden equations. Also, in [13], the authors used Jacobi polynomials for solving multi-dimensional Volterra integral equations. In [39], Bhrawy used Jacobi spectral collocation method for solving multi-dimensional nonlinear fractional sub-diffusion equations. Then, Bhrawy and Zaky [12] used fractional order of Jacobi functions to solve time fractional problems. On the other hand, Jacobi polynomials are also used as a kernel in high-performance heterogeneous computers, named "the Jacobi iterative method" as an example, to gain higher speed up, in [15]. In addition, the orthogonal rotation invariant moments have many use cases in image processing and pattern recognition, including Jacobi moments. In particular, Rahul Upneja et al. [26] used Jacobi-Fourier moments for invariant image processing using a novel fast method leveraging the time complexity.

It can be said explicitly that the applications of these polynomials are so many that they have penetrated into various other branches of science such as neuroscience, etc., and have found important applications. In 2020, Moayeri et al. [36] used Jacobi polynomials for the collocation method (the generalized Lagrange Jacobi Gauss–Lobatto, precisely) to simulate nonlinear spatio-temporal neural dynamic models and also the relevant synchronizations that resulted in a lower computational complexity. Hadian Rasanan et al. [25] has used recently the fractional order of Jacobi polynomials as the activation functions of neural networks to simulate the nonlinear fractional dynamics arising in the modeling of cognitive decision making. Also, Nkengfack et al. [14] used Jacobi polynomials for the classification of EEG signals for epileptic seizures detection and eye states identification. As another application of these polynomials, Abdallah et al. [21] have introduced contiguous relations for Wilson polynomials using Jacobi polynomials. Also, Khodabandehlo et al. [37] used shifted Jacobi operational matrix to present generalized nonlinear delay differential equations of fractional variable-order. For more applications of different types of Jacobi polynomials, the interested readers can see [38, 40, 39, 41].

If one can choose a general family among classical orthogonal polynomials, without no doubt it is the *Jacobi polynomials* $J_n^{\psi, \omega}$. The **Jacobi polynomials** are

also known as **hypergeometric polynomials** named after **Carl Gustav Jacob Jacobi** (1804-1851). In fact, Jacobi polynomials are the most general class in the domain $[-1, 1]$. All of the other families are special cases of this family. For example, $\psi = \omega = 0$ is the *Legendre polynomials* and $\psi = \omega$ is the *Gegenbauer family*.

Carl Gustav Jacob Jacobi (1804-1851) was a Prussian (German) genius mathematician who could reach the standards of entering university at the age of 12, however, he had to wait until the age of 16 according to the University of Berlin's laws. He received his doctorate at the age of 21 from the University of Berlin. Jacobi's contribution is mainly recognized by his theory of elliptic functions and its relations with theta functions in 1829. Moreover, he also had many discoveries in number theory. His results on cubic residues impressed Carl Friedrich Gauss, and brought significant attention to him among other mathematicians such as Friedrich Bessel, and Adrien-Marie Legendre. Jacobi had important research on partial differential equations and their applications and also determinants and as a result, functional determinants also known as Jacobian determinants. Due to the numerous contributions of Jacobi, his name has appeared in many mathematical concepts such as Jacobi's elliptic functions, Jacobi symbol, Jacobi polynomials, Jacobi transform, and many others. His political point of view, during the revolution days of Prussia, made his last years of life full of disturbances. He died under infection to smallpox, at the age of 46^a.

^a For more information about Carl Gustav Jacob Jacobi and his contribution, please see: <https://mathshistory.st-andrews.ac.uk/Biographies/Jacobi/>.

6.2 Preliminaries

This section covers the basic definitions and properties of Jacobi orthogonal polynomials. Moreover, the fractional form of Jacobi orthogonal polynomials is introduced and relevant properties are clarified.

6.2.1 Properties of Jacobi polynomials

Let's start with a simple recap on the definition of orthogonal polynomials. In [17], Szego defined the orthogonal polynomials using a function, such as $F(x)$, to be non-decreasing which includes many points of increase in the interval $[a, b]$. Suppose the following moments exist for this function:

$$c_n = \int_a^b x^n dF(x), \quad n = 0, 1, 2, \dots \quad (6.1)$$

By orthogonalizing the set of non-negative powers of x , i.e. $1, x, x^2, x^3, \dots, x^n, \dots$, he obtained a set of polynomials as follows [17, 30]:

$$J_0(x), J_1(x), J_2(x), \dots, J_n(x), \dots \quad (6.2)$$

which can be determined, uniquely by the following conditions [17]:

- $J_n(x)$ is a polynomial of exactly degree n , where the coefficient of x^n is positive.
- $\{J_n(x)\}$ is orthonormal, i.e.:

$$\int_a^b J_n(x) J_m(x) dF(x) = \delta_{nm}, \quad n, m = 0, 1, 2, \dots, \quad (6.3)$$

where δ_{nm} is the Kronecker's delta function.

Using Eq. 6.3 one can define Jacobi orthogonal polynomials. Therefore, Jacobi polynomials which are denoted by $J_n^{\psi, \omega}(x)$ are orthogonal over $[-1, 1]$ interval with $w^{\psi, \omega}(x)$ weight function [25, 22, 8, 30]:

$$w^{(\psi, \omega)}(x) = (1 - x)^\psi (1 + x)^\omega, \quad (6.4)$$

where $\psi, \omega > -1$. However, the Eq. 6.4 formulation of the weight function suffers from computational difficulties at two distinct input points where $\psi, \omega < 0$ [42]. The input data has to be normalized into interval $[-1, 1]$. In general, it can be written that:

The orthogonal polynomials with the weight function $(b - x)^\psi (x - a)^\omega$ on the interval $[a, b]$, for any orthogonal polynomial denoted by $J_n^{\psi, \omega}(x)$ can be expressed in the following form [42, 11]:

$$J_n^{(\psi, \omega)} \left\{ 2 \left(\frac{x - a}{b - a} \right) - 1 \right\}, \quad (6.5)$$

where $a = -1$ and $b = 1$. It is clear that the minimum and maximum of possible input can be -1 and +1, respectively. Practically while $\psi < 0$ and $x = +1$ causes the term $(1 - x)^\psi$ equal to zero to the power of a negative number and that yields infinity. Similarly, it happens when $\omega < 0$ for the second part $(1 + x)^\omega$, while $x = -1$. To tackle this issue adding a trivial noise to x is proposed through summing with a slack variable $\epsilon = 10^{-4}$ [43].

Consequently, one can rewrite the relation 6.4 as follows:

$$w^{(\psi, \omega)}(x) = (1 - x + \epsilon)^\psi (1 + x + \epsilon)^\omega. \quad (6.6)$$

Note that in programming languages like Python, it is necessary that floating point part precision of $1 - x$ and $1 + x$ should be handled and set equal to precision of the slack variable, for example, following figures compare Eq. 6.4 and Eq. 6.6 for different ψ, ω and $\epsilon = 0.0001$. Here, Maple software has been used to plot which calculates using approximation on boundaries of ordinary weight functions there exist computation difficulties. Fig. 6.1 demonstrates the weight function of Jacobi polynomials without a noise term for different ψ and ω , whereas Fig. 6.2 depicts

the plots of weight function of Jacobi polynomials for different ψ and ω while a noise ($\epsilon = 0.0001$) is added to terms. As these plots depict, there is no considerable difference in the output of related functions.

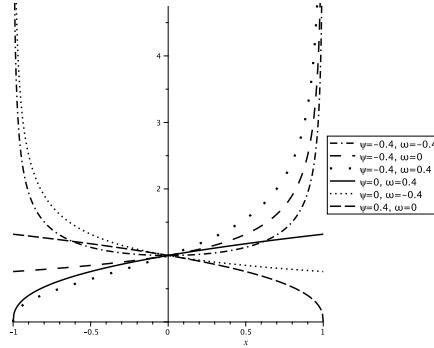


Fig. 6.1: Ordinary Jacobi weight function, using different values of ψ and ω

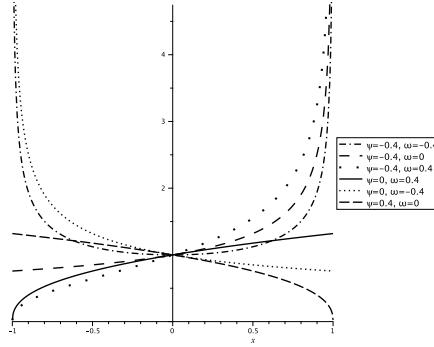


Fig. 6.2: Jacobi weight function using $\epsilon = 0.0001$ and different values of ψ and ω

Thus, the orthogonality relation is [40, 42, 23]:

$$\int_{-1}^1 J_m^{\psi, \omega}(x) J_n^{\psi, \omega}(x) w^{(\psi, \omega)}(x) dx = 0, \quad m \neq n. \quad (6.7)$$

The Jacobi polynomials can be defined as eigenfunctions of a singular Sturm-Liouville differential equation as below [42, 23, 11, 25]:

$$\frac{d}{dx} \left((1-x+\epsilon)^{\psi+1} (1+x+\epsilon)^{\omega+1} \frac{d}{dx} J_n^{\psi, \omega}(x) \right) + (1-x+\epsilon)^{\psi} (1+x+\epsilon)^{\omega} \rho_n J_n^{\psi, \omega}(x) = 0, \quad (6.8)$$

where $\rho_n = n(n + \psi + \omega + 1)$.

Considering the general form of recurrence relation which the orthogonal polynomials follow, for *Jacobi Polynomials* there exist the following relation [42, 23, 11, 25]

$$J_{n+1}^{\psi, \omega}(x) = (A_n x + B_n) J_n^{\psi, \omega}(x) - C_n J_{n-1}^{\psi, \omega}(x), \quad n \geq 1, \quad (6.9)$$

while

$$\begin{aligned} J_0^{\psi, \omega}(x) &= 1, \\ J_1^{\psi, \omega}(x) &= \frac{1}{2}(\psi + \omega + 2)x + \frac{1}{2}(\psi - \omega), \end{aligned} \quad (6.10)$$

where

$$\begin{aligned} A_n &= \frac{(2n + \psi + \omega + 1)(2n + \psi + \omega + 2)}{2(n+1)(n + \psi + \omega + 1)}, \\ B_n &= \frac{(\omega^2 - \psi^2)(2n + \psi + \omega + 1)}{2(n+1)(n + \psi + \omega + 1)(2n + \psi + \omega)}, \\ C_n &= \frac{(n + \psi)(n + \omega)(2n + \psi + \omega + 2)}{(n+1)(n + \psi + \omega + 1)(2n + \psi + \omega)}. \end{aligned} \quad (6.11)$$

The first few *Jacobi Polynomials* are same as Eq. 6.10. On the other hand, because Jacobi polynomials form lengthy terms the higher orders are ignored to present here. For higher orders of *Jacobi Polynomials* the following code can be used which uses python's *sympy* module to calculate symbolically rather than numerically:

Program Code

```
import sympy
x = sympy.Symbol("x")
a = sympy.Symbol("a")
b = sympy.Symbol("b")
beta = sympy.Symbol(r'\omega')
alpha = sympy.Symbol(r'\psi')
x=sympy.Symbol("x")

def Jacobi(x, n):
    An=(2*n+alpha+beta+1)*(2*n+alpha+beta+2)/2*(n+1) \
        *(n+alpha+beta+1)
    Bn=(beta**2-alpha**2)*(2*n+alpha+beta+1)/2*(n+1) \
        *(n+alpha+beta+1)*(2*n+alpha+beta)
    Cn=(n+alpha)*(n+beta)*(2*n+alpha+beta+2)/(n+1) \
        *(n+alpha+beta+1)*(2*n+alpha+beta)
```

```

if n == 0:
    return 1
elif n == 1:
    return ((alpha+beta+2)*x)/2 +(alpha-beta)/2
elif n >= 2:
    return ((An*x+Bn)*Jacobi(x,n-1)-(Cn)*Jacobi(x,n-2))

```

Moreover, these polynomials have some special properties which are introduced in Eqs. 6.12-6.14 [42, 23]:

$$J_n^{\psi, \omega}(-x) = (-1)^n J_n^{\omega, \psi}(x), \quad (6.12)$$

$$J_n^{\psi, \omega}(1) = \frac{\Gamma(n + \psi + 1)}{n! \times \Gamma(\psi + 1)}, \quad (6.13)$$

$$\frac{d^m}{dx^m}(J_n^{\psi, \omega}(x)) = \frac{\Gamma(m + n + \psi + \omega + 1)}{2^m \times \Gamma(n + \psi + \omega + 1)} J_{n-m}^{\psi+m, \omega+m}(x). \quad (6.14)$$

Theorem 6.1 [25] *The Jacobi polynomial $J_n^{\psi, \omega}(x)$, has exactly n real zeros on the interval $(-1, 1)$.*

Proof Referring to [25] shows that n zeros of Jacobi polynomial $J_n^{\psi, \omega}(x)$ can be achieved by calculating the eigenvalues of the following three-diagonal matrix

$$K_n = \begin{bmatrix} \rho_1 & \gamma_2 & & & \\ \gamma_1 & \rho_2 & \gamma_3 & & \\ & \gamma_3 & \rho_3 & \gamma_4 & \\ & & \ddots & \ddots & \ddots \\ & & & \gamma_{n-1} & \rho_{n-1} & \gamma_n \\ & & & & \gamma_n & \rho_n \end{bmatrix}$$

where

$$\rho_{i+1} = \frac{\int_{-1}^1 x J_i^{\psi, \omega}(x) J_i^{\psi, \omega}(x) w^{\psi, \omega}(x) dx}{\int_{-1}^1 J_i^{\psi, \omega}(x) J_i^{\psi, \omega}(x) w^{\psi, \omega}(x) dx}, \quad (6.15)$$

$$\gamma_{i+1} = \begin{cases} 0 & i = 0, \\ \frac{\int_{-1}^1 J_i^{\psi, \omega}(x) J_{i-1}^{\psi, \omega}(x) w^{\psi, \omega}(x) dx}{\int_{-1}^1 J_{i-1}^{\psi, \omega}(x) J_{i-1}^{\psi, \omega}(x) w^{\psi, \omega}(x) dx} & i = 1, 2, \dots \end{cases}. \quad (6.16)$$

Jacobi polynomials satisfy the following ordinary differential equation [42, 23, 25]:

$$(1 - x^2) \frac{d^2y}{dx^2} + [\omega - \psi(\psi + \omega + 2)x] \frac{dy}{dx} + \lambda y = 0. \quad (6.17)$$

The solution can be expressed by means of power series $y = \sum_{n=0}^{\infty} a_n x^n$:

$$(1 - x^2) \sum_{n=0}^{\infty} n(n-1)a_n x^{n-2} + [\omega - \psi - (\psi + \omega + 2)x] \sum_{n=0}^{\infty} a_n^{n-1} + \lambda \sum_{n=0}^{\infty} a_n^n = 0. \quad (6.18)$$

Also, generating function for *Jacobi Polynomials* can be defined as [42, 23]:

$$\frac{2^{\psi+\omega}}{R(1+R-x)^\psi(1+R+X)^\omega} = \sum_{n=0}^{\infty} J_n^{\psi,\omega}(x)x^n, \quad (6.19)$$

where $R = \sqrt{1 - 2xz + z^2}$ and $|z| < 1$.

This family of orthogonal polynomials follows the same symmetry as previous orthogonal families already introduced, i.e.

$$J_n^{\psi,\omega}(x) = (-1)^n J_n^{\psi,\omega}(x) = \begin{cases} J_n(-x), & n \text{ even} \\ -J_n(-x), & n \text{ odd} \end{cases} \quad (6.20)$$

Figures 6.3 and 6.4 illustrate *Jacobi Polynomials* of orders 0 to 6 and compare negative ψ , positive ω in Fig. 6.3 versus positive ψ while negative ω in Fig. 6.4. Figures 6.5 and 6.6 depict Jacobi Polynomials of order 5 with different ψ , fixed ω , fixed ψ , and fixed ω , respectively.

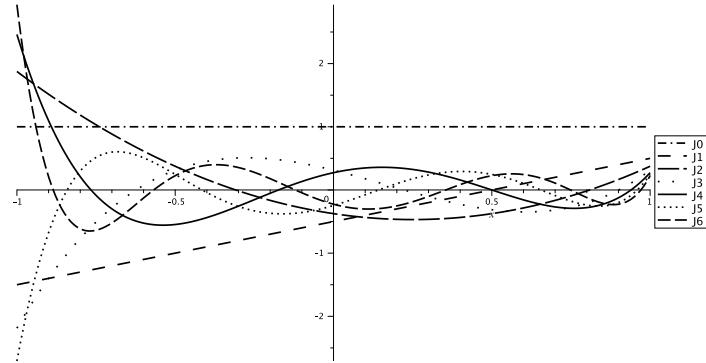
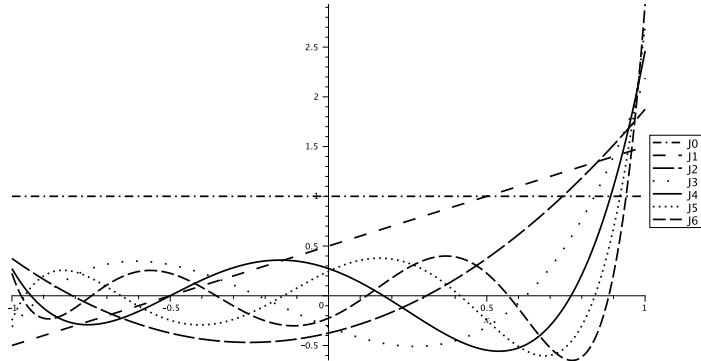
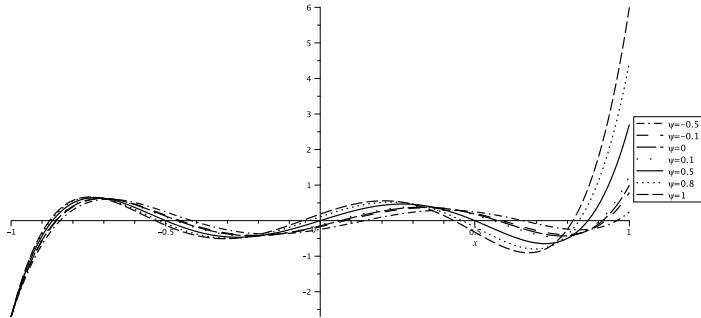
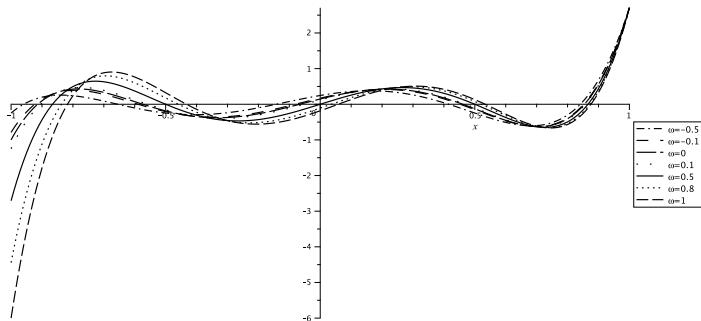


Fig. 6.3: Jacobi polynomials of order 0 to 6, negative ψ and positive ω

Fig. 6.4: Jacobi polynomials of order 1 to 6, positive ψ and negative ω Fig. 6.5: Jacobi polynomials of order 5, different ψ and fixed ω Fig. 6.6: Jacobi polynomials of order 5, fixed ψ and different ω

6.2.2 Properties of Fractional Jacobi functions

In order to obtain fractional order of Jacobi functions, a transformation is used as $x' = 2(\frac{x-a}{b-a})^\alpha - 1$, where a and b are the minimum and maximum of the transition and $\alpha > 0$. By applying this transformation, the fractional order of Jacobi functions are obtained which are denoted by $FJ_n^{\psi, \omega, \alpha}(x)$ as follow:

$$FJ_n^{\psi, \omega, \alpha}(x) = J_n^{\psi, \omega}(2(\frac{x-a}{b-a})^\alpha - 1). \quad (6.21)$$

The fractional order of Jacobi function is orthogonal over the interval $[-1, 1]$ with the following weight function [42, 23, 25]:

$$w^{\psi, \omega, \alpha}(x) = \left(1 - \left(2(\frac{x-a}{b-a})^\alpha - 1\right) - \epsilon\right)^\psi \left(1 + \left(2(\frac{x-a}{b-a})^\alpha - 1\right) + \epsilon\right)^\omega. \quad (6.22)$$

Similarly, there exists a Sturm-Liouville differential equation for the fractional order of Jacobi functions. This Sturm-Liouville equation is as follows [42, 23, 25]:

$$\begin{aligned} \frac{d}{dx} \left((1 - (2(\frac{x-a}{b-a})^\alpha - 1) - \epsilon)^{\psi+1} (1 + (2(\frac{x-a}{b-a})^\alpha - 1) + \epsilon)^{\omega+1} \frac{d}{dx} J_n^{\psi, \omega}(x) \right) \\ + (1 - (2(\frac{x-a}{b-a})^\alpha - 1) - \epsilon)^\psi (1 + (2(\frac{x-a}{b-a})^\alpha - 1) + \epsilon)^\omega \rho_n J_n^{\psi, \omega}(x) = 0, \end{aligned} \quad (6.23)$$

where $\rho_n = n(n + \psi + \omega + 1)$. By applying the mapping $z = 2(\frac{x-a}{b-a})^\alpha - 1$ to the generating function defined for *Jacobi polynomials*, one can get the equation for the fractional form:

$$\frac{2^{\psi, \omega}}{R(1 + R - (2(\frac{x-a}{b-a})^\alpha - 1)^\psi (1_R + (2(\frac{x-a}{b-a})^\alpha - 1)^\psi)^\omega)} = \sum_{n=0}^{\infty} FJ_n^{\psi, \omega, \alpha}(x) ((2(\frac{x-a}{b-a})^\alpha - 1)^\psi)^n, \quad (6.24)$$

where

$$R = \sqrt{1 - 2x((2(\frac{x-a}{b-a})^\alpha - 1)^\psi) + ((2(\frac{x-a}{b-a})^\alpha - 1)^\psi)^2},$$

and $(2(\frac{x-a}{b-a})^\alpha - 1)^\psi \in [-1, 1]$.

Fractional Jacobi polynomials are also orthogonal. These polynomials are orthogonal in interval $[-1, 1]$ with respect to the weight function similar to Eq. 6.6 where the input x is mapped by means of $2(\frac{x-a}{b-a})^\alpha - 1$. So the proper *weight function* for orthogonality relation of *fractional Jacobi polynomials* is [11, 25, 8]:

$$Fw^{\psi, \omega, \alpha}(x) = (1 - (2(\frac{x-a}{b-a})^\alpha - 1) + \epsilon)^\psi (1 - (2(\frac{x-a}{b-a})^\alpha - 1) + \epsilon)^\omega. \quad (6.25)$$

Now one can define the orthogonality relation for the *fractional Jacobi polynomials* as:

$$\int_a^b FP_m^{\psi, \omega, \alpha}(x) FP_n^{\psi, \omega, \alpha}(x) Fw^{\psi, \omega, \alpha}(x) dx. \quad (6.26)$$

The recursive relation for *fractional Jacobi polynomials* can be defined the same way as already has been done so far by using the mapped x into the normal equation [42, 23, 11, 25]:

$$\begin{aligned} FJ_0^{\psi, \omega, \alpha}(x) &= 1, \\ FJ_1^{\psi, \omega, \alpha}(x) &= \frac{1}{2}(\psi + \omega + 2)(2(\frac{x-a}{b-a})^\alpha - 1) + \frac{1}{2}(\psi - \omega), \\ FJ_{n+1}^{\psi, \omega, \alpha}(x) &= (A_n(2(\frac{x-a}{b-a})^\alpha - 1) + B_n)FJ_n^{\psi, \omega, \alpha}(x) - C_nFJ_{n+1}^{\psi, \omega, \alpha}(x), \quad n \geq 1. \end{aligned} \quad (6.27)$$

where

$$\begin{aligned} A_n &= \frac{(2n + \psi + \omega + 1)(2n + \psi + \omega + 2)}{2(n+1)(n + \psi + \omega + 1)}, \\ B_n &= \frac{(\omega^2 - \psi^2)(2n + \psi + \omega + 1)}{2(n+1)(n + \psi + \omega + 1)(2n + \psi + \omega)}, \\ C_n &= \frac{(n + \psi)(n + \omega)(2n + \psi + \omega + 2)}{(n+1)(n + \psi + \omega + 1)(2n + \psi + \omega)}. \end{aligned} \quad (6.28)$$

Any order of *fractional Jacobi polynomials* can be obtained by means of following python code, as well:

Program Code

```
import sympy
x = sympy.Symbol("x")
a = sympy.Symbol("a")
b = sympy.Symbol("b")
beta = sympy.Symbol(r'\omega')
alpha = sympy.Symbol(r'\psi')
delta = sympy.Symbol(r'\alpha')
x=sympy.sympify(1-2*(x-a/b-a)**delta)

def A(n):
    return (2*n+alpha+beta+1)*(2*n+alpha+beta+2)/2*(n+1)*(n+alpha+beta+1)
def B(n):
    return (beta**2-alpha**2)*(2*n+alpha+beta+1)/2*(n+1)*(n+alpha+beta+1)
    * (2*n+alpha+beta)
def C(n):
    return (n+alpha)*(n+beta)*(2*n+alpha+beta+2)/(n+1)*(n+alpha+beta+1)
    * (2*n+alpha+beta)
def FJacobi(x, n):
    if n == 0:
        return 1
    elif n == 1:
        return ((alpha+beta+2)*x)/2+(alpha-beta)/2
```

```
elif n >= 2:  
    return ((A(n-1)*x+B(n-1))*Jacobi(x,n-1)-(C(n-1))*Jacobi(x,n-2))
```

Here Fig. 6.7 illustrates *fractional Jacobi polynomials* of order 0 to 6 with $\psi = -0.5$, $\omega = 0.5$, $\alpha = 0.5$, and Fig. 6.8 compares *fractional Jacobi polynomials of order 5* with different α .

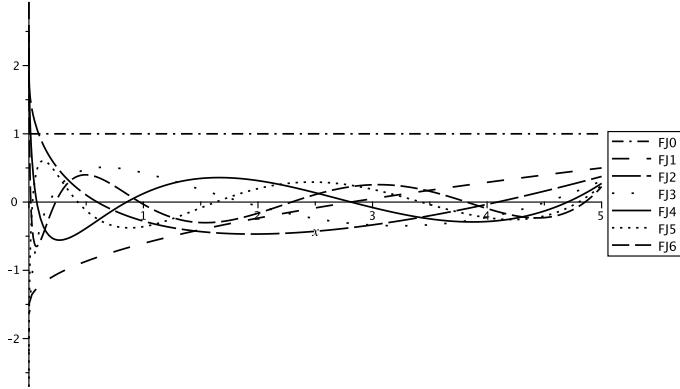


Fig. 6.7: Fractional Jacobi polynomials of order 0 to 6, positive ψ and ω

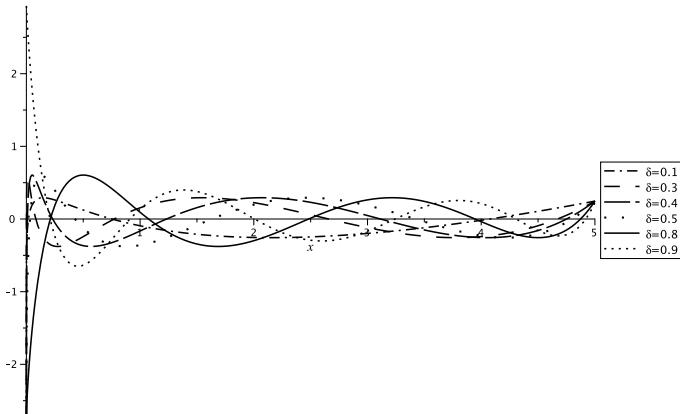


Fig. 6.8: Fractional Jacobi polynomials of order 5, positive ψ and ω and different α

6.3 Jacobi kernel functions

In this section, the kernel functions of ordinary type will be introduced and also its validation will be proved according to Mercer condition. Moreover, Wavelet-Jacobi kernel will be introduced which recently attracted the attention of researchers. The last subsection is devoted to fractional Jacobi kernel.

6.3.1 Ordinary Jacobi kernel function

As we already know, the unweighted orthogonal polynomial kernel function for SVM can be written as:

$$K(x, z) = \sum_{i=0}^n J_i(x)J_i(z), \quad (6.29)$$

where $J(\cdot)$ denotes the evaluation of the polynomial. x and z are the kernel's input arguments and n is the highest polynomial order. Using this definition and the fact that need the multidimensional form of the kernel function, one can introduce the *Jacobi kernel function* by the evaluation of inner product of input vectors ($\langle x, z \rangle = xz^T$) [19, 32]:

$$k_{Jacobi}(x, z) = \sum_{i=0}^n J_i^{\psi, \omega}(x)J_i^{\psi, \omega T}(z)w^{\psi, \omega}(x, z). \quad (6.30)$$

Theorem 6.2 [19] *The Jacobi Kernel introduced in Eq. 6.30 is valid mercer kernel.*

Proof Mercer theorem states that a SVM kernel should be positive semi definite or non-negative, in other words, the kernel should satisfy below relation:

$$\iint K(x, z)f(x)f(z)dxdz \geq 0. \quad (6.31)$$

By using the fact that multiplication of two valid kernels is also a valid kernel, one can divide the Jacobi kernel introduced at Eq. 6.30 into two kernels, one the inner product and the other is weight function, therefore:

$$K_{(1)}(x, z) = \sum_{i=0}^n J_i^{\psi, \omega}(x)J_i^{\psi, \omega T}(z), \quad (6.32)$$

$$\begin{aligned} K_{(2)}(x, z) &= w^{\psi, \omega}(x, z) \\ &= (d - \langle x, z \rangle + \epsilon)^{\psi}(d + \langle x, z \rangle + \epsilon)^{\omega}, \end{aligned} \quad (6.33)$$

where d is the dimension of input x and z . Considering $f(x)$ is a function where $f : \mathbb{R}^m \rightarrow \mathbb{R}$, one can evaluate the mercer condition for $K_{(1)}(\mathbf{x}, \mathbf{z})$ as follows, assuming each element is independent from others:

$$\begin{aligned}
\iint K_{(1)}(x, z) f(x) f(z) dx dz &= \iint \sum_{j=0}^n J_i^{\psi, \omega}(x) J_i^{\psi, \omega^T}(z) f(x) f(z) dx dz \\
&= \sum_{j=0}^n \iint J_i^{\psi, \omega}(x) J_i^{\psi, \omega^T}(z) f(x) f(z) dx dz \\
&= \sum_{j=0}^n \left[\int J_i^{\psi, \omega}(x) f(x) dx \int J_i^{\psi, \omega^T}(z) f(z) dz \right] \\
&= \sum_{j=0}^n \left[\left(\int J_i^{\psi, \omega}(x) f(x) dx \right) \left(\int J_i^{\psi, \omega^T}(z) f(z) dz \right) \right] \geq 0.
\end{aligned} \tag{6.34}$$

Therefore, the kernel $K_{(1)}(x, z)$ is a valid Mercer kernel. To prove $K_{(2)}(x, z) \geq 0$, we can prove the weight function in Eq. 6.6 is positive semi definite, because it is easier to intuitively consider, then the general weight function for kernel which reformulated for two vectors input, becomes clear. Due to normalization of the input data, this weight function is positive semi-definite, so the weight function $w^{\psi, \omega}(x, z) = (d - \langle x, z \rangle)^\psi (d + \langle x, z \rangle)^\omega$ which is a generalized form for two input vector of the kernel is positive semi-definite too. \square

6.3.2 Other Jacobi kernel functions

SVM with the kernel-trick heavily depends on the proper kernel to choose according to data, that is still an attractive topic to introduce and examine new kernels for SVM. Orthogonality properties of some polynomials such as Jacobi has made them a hot alternative for such use cases. By the way, some previously introduced kernels such as wavelet has been used successfully [19]. Combining the wavelet and orthogonal Kernel functions such as Chebyshev, Hermit and Legendre has already been proposed and examined in signal processing [6], solving differential equations[44, 45], optimal control [46, 47], and calculus variations problems [48].

6.3.2.1 Regularized Jacobi Wavelets kernel function

Abassa et al. [19] recently introduced SVM kernels based on Jacobi wavelets. However, the proposed kernel will be glanced here, interested reader may find the proof and details in original paper [19]. It should be noted that some notations has to be changed to preserve the integrity of the book, but the integration of formulation are intact and same as the original work.

The Jacobi polynomials $J_m^{(\psi, \omega)}$ are defined as follows:

$$\begin{aligned} J_m^{(\psi, \omega)}(x) &= \frac{(\psi + \omega + 2m - 1)[\psi^2 - \omega^2 + x(\psi + \omega + 2m)(\psi + \omega + 2m - 2)]}{2m(\psi + \omega + 2m - 2)(\psi + \omega + m)} J_{m-1}^{(\psi, \omega)}(x) \\ &\quad - \frac{(\psi + m - 1)(\omega + m - 1)(\psi + \omega + 2m)}{m(\psi + \omega + 2m - 2)(\psi + \omega + m)} J_{m-2}^{(\psi, \omega)}(x), \end{aligned}$$

where $\psi > -1$, $\omega > -1$, and

$$J_0^{(\psi, \omega)}(x) = 1, J_1^{(\psi, \omega)}(x) = \frac{\psi + \omega + 2}{2}x + \frac{\psi - \omega}{2}.$$

These polynomials belong to the weight space $L_w^2([-1, 1])$, i.e.

$$\left\langle J_m^{(\psi, \omega)}, J_{m'}^{(\psi, \omega)} \right\rangle_{L_w^2} = h_m^{(\psi, \omega)} \delta_{m, m'}, \quad \forall m, m' \in \mathbb{N},$$

where

$$h_m^{(\psi, \omega)} = \| J_m^{(\psi, \omega)} \| = \frac{2^{\psi+\omega+1} \Gamma(\psi + m + 1) \Gamma(\omega + m + 1)}{(2m + 1 + \psi + \omega)m! \Gamma(\psi + \omega + m + 1)},$$

and $w(x) = (1 - x)^\psi (1 + x)^\omega$. In addition, $\delta_{n,m}$ is the Kronecker function, Γ is the Euler gamma, and $\langle \cdot, \cdot \rangle_{L_w^2}$ denotes the inner product of $L_w^2([-1, 1])$. The family $\{J_m^{(\psi, \omega)}\}_{m \in \mathbb{N}}$ forms an orthogonal basis for $L_w^2([-1, 1])$.

6.3.3 Fractional Jacobi kernel

Since the weight function of the fractional Jacobi functions is equal to Eq. 6.25, defining the fractional Jacobi kernel is easy. One can construct the *fractional Jacobi kernel* as:

$$K_{FJacobi}(x, z) = \sum_{i=0}^n \langle FJ_i^{\psi, \omega, \alpha}(x), FJ_i^{\psi, \omega, \alpha}(z) \rangle Fw^{\psi, \omega, \alpha}(x, z), \quad (6.35)$$

Theorem 6.3 *The Jacobi kernel introduced at Eq. 6.35 is valid Mercer kernel.*

Proof Similar to the proof for theorem 6.2, fractional Jacobi kernel function Eq. 6.35 can be considered as multiplication of two kernels:

$$K_{(1)}(x, z) = \sum_{i=0}^n FJ_i^{\psi, \omega, \alpha}(x) FJ_i^{\psi, \omega, \alpha T}(z), \quad (6.36)$$

$$\begin{aligned} K_{(2)}(x, z) &= Fw^{\psi, \omega, \alpha}(x, z) \\ &= (d - \langle 2(\frac{x-a}{b-a})^\alpha - 1, 2(\frac{z-a}{b-a})^\alpha - 1 \rangle - \epsilon)^\psi (d + \langle 2(\frac{x-a}{b-a})^\alpha - 1, 2(\frac{z-a}{b-a})^\alpha - 1 \rangle + \epsilon)^\omega, \end{aligned} \quad (6.37)$$

where d is the dimension of inputs x and z . Consider $f : \mathbb{R}^m \rightarrow \mathbb{R}$, and assume each element is independent from others. One can evaluate the Mercer condition for $K_{(1)}(x, z)$ as follows:

$$\begin{aligned}
& \iint K_{(1)}(x, z) f(x) f(z) dx dz \\
&= \iint \sum_{j=0}^n F J_i^{\psi, \omega, \alpha}(x) F J_i^{\psi, \omega, \alpha^T}(z) f(x) f(z) dx dz \\
&= \sum_{j=0}^n \iint F J_i^{\psi, \omega, \alpha}(x) F J_i^{\psi, \omega, \alpha^T}(z) f(x) f(z) dx dz \\
&= \sum_{j=0}^n \left[\int F J_i^{\psi, \omega, \alpha}(x) f(x) dx \int F J_i^{\psi, \omega, \alpha^T}(z) f(z) dz \right] \\
&= \sum_{j=0}^n \left[\left(\int F J_i^{\psi, \omega, \alpha}(x) f(x) dx \right) \left(\int F J_i^{\psi, \omega, \alpha^T}(z) f(z) dz \right) \right] \geq 0.
\end{aligned} \tag{6.38}$$

Therefore, Eq. 6.36 is a valid Mercer kernel. Validity of $K_{(2)}(x, z)$ can be considered similar to the weight function of ordinary Jacobi kernel function. It can be deduced that the output of $K_{(2)}(x, z)$ is never less than zero, as the inner product of two vectors which are normalized over $[-1, 1]$ is in the range $[-d, d]$, where d is the dimension of the input vectors x or z , and the effect of negative inner product will be neutralized by the parameter d . Therefore, Eq. 6.37 is also a valid Mercer kernel. \square

6.4 Application of Jacobi kernel functions on real datasets

In this section, the results of *Jacobi* and the *fractional Jacobi kernel* on some well-known datasets, are compared with other kernels such as *RBF*, *polynomial*, *Chebyshev*, *fractional Chebyshev*, *Gegenbauer*, *fractional Gegenbauer*, *Legendre*, and *fractional Legendre kernels* introduced in the previous chapters. To have a neat classification, there may need to apply some prepossessing steps to dataset, here these steps are not in focus, but the normalization which is mandatory when using *Jacobi polynomials* as a kernel. There are some online data stores available for public use, such a widely used datastore is the *UCI Machine Learning Repository*¹ of *University of California, Irvine*, and also *Kaggle*². For this section, 4 datasets from UCI are used that are well known for machine learning practitioners.

¹ <https://archive.ics.uci.edu/ml/datasets.php>

² <https://www.kaggle.com/datasets>

6.4.1 Spiral dataset

The Spiral dataset is already introduced in details in Chapter 3. As a quick recap, Fig. 6.9 depicts the original Spiral dataset (on the left) and also the same dataset in fractional space of order $\alpha = 0.3$ (on the right), consist of 1000 data points. In classification task on this dataset, 90% of data points are chosen as test set. It is clear that in fractional space data points are concentrated in positive quarter as it seems in 2D plot. As already discussed, this is only a step in preparing the dataset to apply the kernel function and find the most suitable classifier.

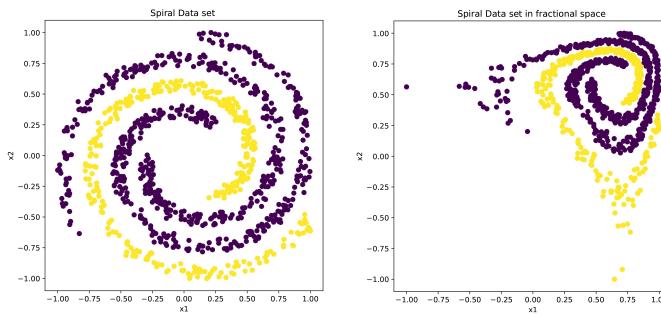


Fig. 6.9: Spiral dataset

Fig. 6.10 demonstrates the same dataset while the Jacobi kernel function is applied. The plot on the left depicts the original dataset after the Jacobi kernel of order 3 and $\psi = -0.2$, and $\omega = 0.3$ is applied to. The plot on the right demonstrates the Spiral dataset in fractional space of order 0.3 where the fractional Jacobi kernel of order 3, $\psi = -0.2$, $\omega = 0.3$, and the fractional order of 0.3 is applied.

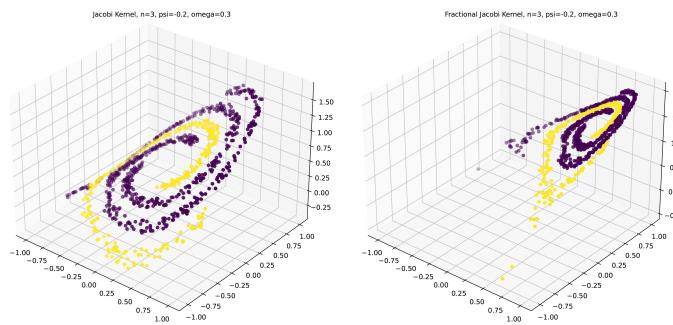


Fig. 6.10: Jacobi kernel applied to Spiral dataset, both normal and fractional

Applying Jacobi kernel has given a higher dimension to each data point of the Spiral dataset. Thus, a new order has emerged in this way, and consequently a chance to find a decision surface (in 3D) to do a binary classification. However, it may does not seem much improvement, but one have to wait for the Jacobi classifiers. Afterwards, one can achieve a better judgment. Fig. 6.11 depicts the classifiers of different settings of Jacobi kernel on Spiral dataset. Due computational difficulties of Jacobi kernel, orders of 7 and 8 have been ignored. Following figures only present the classifiers of Jacobi kernel and fractional Jacobi kernels of orders 3 to 6. Since it is clear in following plots, with the fixed parameters $\psi = -0.5$ and $\omega = 0.2$, corresponding classifier get more curved and twisted as order raises from 3 to 6. Nevertheless, this characteristics does not necessarily mean better classifying opportunity due to its highly depends on multiple parameters beside kernel's specific parameters. According to these plots, one can deduce that Jacobi kernels of order 3 and 5 with $\psi = -0.5$ and $\omega = 0.2$ are slightly better classifiers for binary classification of class 1-v-[2,3] on Spiral dataset in comparison to same kernel of orders 4 and 6.

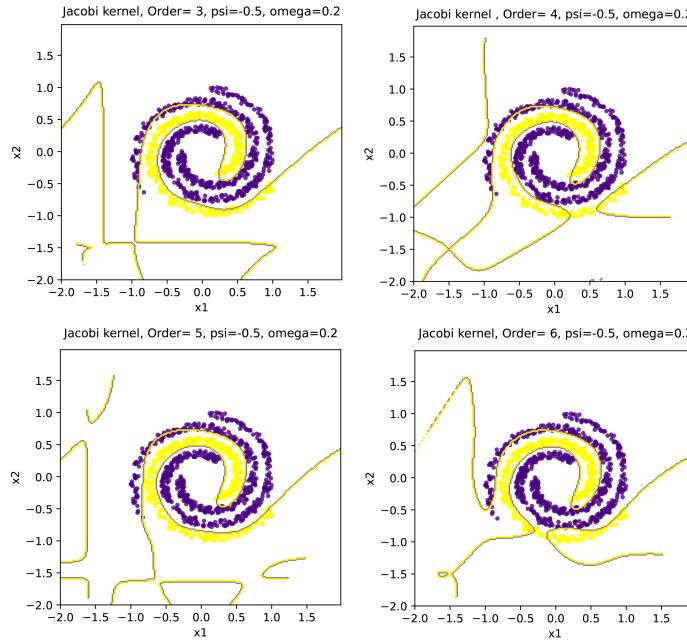


Fig. 6.11: Jacobi kernel applied to Spiral dataset, corresponding classifiers of order 3 to 6, at fixed parameters $\psi = -0.5, \omega = 0.2$

Fig. 6.12 demonstrates corresponding plots of fractional Jacobi kernel of orders 3 to 6, with fixed parameters of $\psi = -0.5$ and $\omega = 0.2$ and fractional order of 0.3. Clearly in fractional space, the classifier gets more twisted and intricate in

comparison to the normal one. The fractional Jacobi kernel is fairly successful in this classification task, both in normal and fractional space. The following figure depicts how the fractional Jacobi kernel determines the decision boundary.

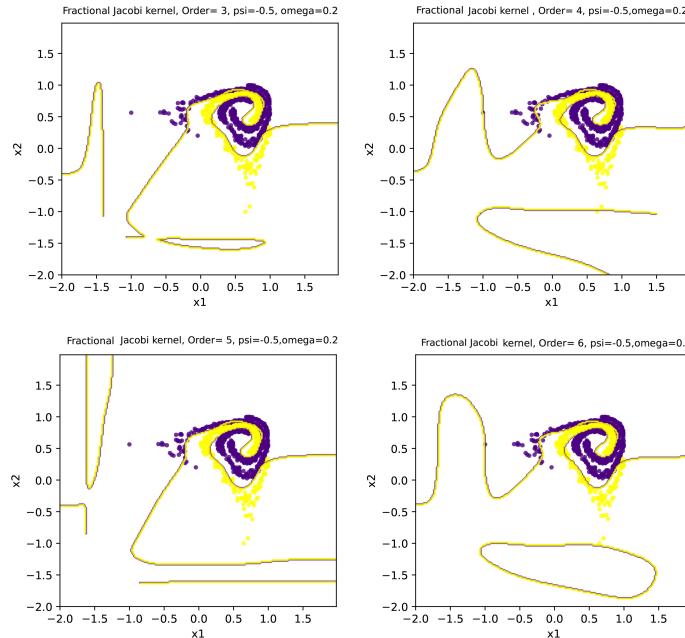


Fig. 6.12: Fractional Jacobi kernel applied to Spiral dataset, corresponding classifiers of order 3 to 6, at fixed parameters $\psi = -0.5$, $\omega = 0.2$, and fractional order=0.3

The following tables summarize experiment results of SVM classification on Spiral data-set. Using **OVA** method, all possible formations of classes 1-v-[2,3], 2-v-[1,3] and 3-v-[1,2] are chosen and relevant results are summarized in Tables 6.1, 6.2 and 6.3, respectively. In these experiments, the target was not getting the best accuracy scores. Therefore, reaching a higher accuracy score is technically possible through finding the best number of support vectors, setting the generalization parameter of SVM, and definitely trying different values for kernel's parameters. Table 6.1 is the complete accuracy comparison between introduced orthogonal kernels, RBF, and polynomial kernels on the binary classification of 1-v-[2,3] on Spiral dataset discussed in Chapter 3, which fractional Legendre kernel was closely accurate as 100%. However, the lowest accuracy score was 94% for the fractional Gegenbauer kernel. Table 6.2 expresses the summary of the second possible binary classification task on the Spiral dataset (2-v-[1,3]). The RBF and fractional Legendre kernels outperform other kernels with an accuracy score of 98%, while the second-best score is close to 93%. Finally, Table 6.3 summarizes the third form of binary classification

on the Spiral dataset which is 3-v-[1,3]. The fractional Legendre kernel has the best performance with 99% accuracy. However, the fractional Jacobi kernel yields an accuracy close to 97%.

Table 6.1: **Class 1-v-[2,3]**, comparison of RBF, polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, fractional Gegenbauer, Jacobi, fractional Jacobi kernels on the Spiral dataset. It is clear that RBF, fractional Chebyshev, and fractional Jacobi kernels closely outcome best results

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Psi(ψ) | Omega(ω) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-------------------------------|-----------------------------------|-----------------|
| RBF | 0.73 | - | - | - | - | - | - | 0.97 |
| Polynomial | - | 8 | - | - | - | - | - | 0.9533 |
| Chebyshev | - | - | 5 | - | - | - | - | 0.9667 |
| Fractional Chebyshev | - | - | 3 | 0.3 | - | - | - | 0.9733 |
| Legendre | - | - | 6 | - | - | - | - | 0.9706 |
| Fractional Legendre | - | - | 7 | 0.8 | - | - | - | 0.9986 |
| Gegenbauer | - | - | 6 | - | 0.3 | - | - | 0.9456 |
| Fractional Gegenbauer | - | - | 6 | 0.3 | 0.7 | - | - | 0.9533 |
| Jacobi | - | - | 3 | - | - | -0.8 | 0 | 0.96 |
| Fractional Jacobi | - | - | 7 | 0.7 | - | -0.5 | 0.6 | 0.9711 |

Table 6.2: **Class 2-v-[1,3]**, comparison of RBF, polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, fractional Gegenbauer, Jacobi, and fractional Jacobi kernels on the Spiral dataset. As it is clear, RBF kernel outperforms other kernels

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Psi(ψ) | Omega(ω) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-------------------------------|-----------------------------------|-----------------|
| RBF | 0.1 | - | - | - | - | - | - | 0.9867 |
| Polynomial | - | 5 | - | - | - | - | - | 0.9044 |
| Chebyshev | - | - | 6 | - | - | - | - | 0.9289 |
| Fractional Chebyshev | - | - | 6 | 0.8 | - | - | - | 0.9344 |
| Legendre | - | - | 8 | - | - | - | - | 0.9344 |
| Fractional Legendre | - | - | 8 | 0.4 | - | - | - | 0.9853 |
| Gegenbauer | - | - | 5 | - | 0.3 | - | - | 0.9278 |
| Fractional Gegenbauer | - | - | 4 | 0.6 | 0.6 | - | - | 0.9356 |
| Jacobi | - | - | 5 | - | - | -0.2 | 0.4 | 0.9144 |
| Fractional Jacobi | - | - | 3 | 0.3 | - | -0.2 | 0 | 0.9222 |

Table 6.3: **Class 3-v-[1,2]**, comparison of RBF, polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, fractional Gegenbauer, Jacobi, and fractional Jacobi kernels on the Spiral dataset. The RBF and polynomial kernels have the best accuracy and after them is the fractional Jacobi kernel

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Psi(ψ) | Omega(ω) | Accuracy |
|------------------------------|-------|-------|-------|-------------------|---------------------|---------------|-------------------|---------------|
| RBF | 0.73 | - | - | - | - | - | - | 0.9856 |
| Polynomial | - | 5 | - | - | - | - | - | 0.9856 |
| Chebyshev | - | - | 6 | - | - | - | - | 0.9622 |
| Fractional Chebyshev | - | - | 6 | 0.6 | - | - | - | 0.9578 |
| Legendre | - | - | 6 | - | - | - | - | 0.9066 |
| Fractional Legendre | - | - | 6 | 0.9 | - | - | - | 0.9906 |
| Gegenbauer | - | - | 6 | - | 0.3 | - | - | 0.9611 |
| Fractional Gegenbauer | - | - | 6 | 0.9 | 0.3 | - | - | 0.9644 |
| Jacobi | - | - | 5 | - | - | -0.8 | 0.3 | 0.96 |
| Fractional Jacobi | - | - | 7 | 0.9 | - | -0.8 | 0 | 0.9722 |

6.4.2 Three Monks's dataset

The Monks problem is a classic one introduced in 1991, according to *The MONK's Problems - A Performance Comparison of Different Learning algorithms* by S.B. Thrun et al. For more details, please refer to related section in chapter 3, i.e. (3.4.2).

The *Jacobi kernels* introduced in Eq. 6.30 and Eq. 6.35 are applied on datasets from *The three monks's problem* and the relevant are appended to Tables 6.4, 6.5, and 6.6. *Fractional Jacobi* kernel showed slightly better performance on these datasets in comparison with other fractional and ordinary kernels on monks's problem M1 and M2.

Table 6.4: Comparison of RBF, polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, fractional Gegenbauer, Legendre, fractional Legendre, Jacobi, and fractional Jacobi kernels on Monk's first problem. The fractional Gegenbauer, and fractional Jacobi kernels have the most desirable accuracy of 1

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Psi(ψ) | Omega(ω) | Accuracy |
|------------------------------|-------|-------|-------|-------------------|---------------------|---------------|-------------------|----------|
| RBF | 2.844 | - | - | - | - | - | - | 0.8819 |
| Polynomial | - | 3 | - | - | - | - | - | 0.8681 |
| Chebyshev | - | - | 3 | - | - | - | - | 0.8472 |
| Fractional Chebyshev | - | - | 3 | 1/16 | - | - | - | 0.8588 |
| Legendre | - | - | 3 | - | - | - | - | 0.8333 |
| Fractional Legendre | - | - | 3 | 0.8 | - | - | - | 0.8518 |
| Gegenbauer | - | - | 3 | - | -0.2 | - | - | 0.9931 |
| Fractional Gegenbauer | - | - | 3 | 0.7 | 0.2 | - | - | 1 |
| Jacobi | - | - | 4 | - | - | -0.2 | -0.5 | 0.9977 |
| Fractional Jacobi | - | - | 4 | 0.4 | - | -0.2 | -0.5 | 1 |

Table 6.5: Comparison of RBF, polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, fractional Gegenbauer, Legendre, fractional Legendre, Jacobi, and the fractional Jacobi kernels on The second Monk's problem. The fractional Legendre and fractional Jacobi Kernels have the best accuracy of 1

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Psi(ψ) | Omega(ω) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-------------------------------|-----------------------------------|-----------------|
| RBF | 5.5896 | - | - | - | - | - | - | 0.875 |
| Polynomial | - | 3 | - | - | - | - | - | 0.8657 |
| Chebyshev | - | - | 3 | - | - | - | - | 0.8426 |
| Fractional Chebyshev | - | - | 3 | 1/16 | - | - | - | 0.9653 |
| Legendre | - | - | 3 | - | - | - | - | 0.8032 |
| Fractional Legendre | - | - | 3 | 0.8 | - | - | - | 1 |
| Gegenbauer | - | - | 3 | - | 0.5 | - | - | 0.7824 |
| Fractional Gegenbauer | - | - | 3 | 0.1 | 0.5 | - | - | 0.9514 |
| Jacobi | - | - | 3 | - | - | -0.5 | -0.2 | 0.956 |
| Fractional Jacobi | - | - | 3 | 0.1 | - | -0.2 | -0.5 | 1 |

Table 6.6: Comparison of RBF, polynomial, Chebyshev, fractional Chebyshev, Gegenbauer, fractional Gegenbauer, Legendre, fractional Legendre, Jacobi, and fractional Jacobi kernels on the third Monk's problem. The Gegenbauer and fractional Gegenbauer kernels have the best accuracy score, then Jacobi, fractional Jacobi, RBF, and fractional Chebyshev Kernels

| | Sigma | Power | Order | Alpha(α) | Lambda(λ) | Psi(ψ) | Omega(ω) | Accuracy |
|------------------------------|--------------|--------------|--------------|-----------------------------------|-------------------------------------|-------------------------------|-----------------------------------|-----------------|
| RBF | 2.1586 | - | - | - | - | - | - | 0.91 |
| Polynomial | - | 3 | - | - | - | - | - | 0.875 |
| Chebyshev | - | - | 6 | - | - | - | - | 0.895 |
| Fractional Chebyshev | - | - | 5 | 1/5 | - | - | - | 0.91 |
| Legendre | - | - | 3 | - | - | - | - | 0.8472 |
| Fractional Legendre | - | - | 3 | 0.8 | - | - | - | 0.8379 |
| Gegenbauer | - | - | 4 | - | -0.2 | - | - | 0.9259 |
| Fractional Gegenbauer | - | - | 3 | 0.7 | -0.2 | - | - | 0.9213 |
| Jacobi | - | - | 5 | - | - | -0.5 | 0.0 | 0.919 |
| Fractional Jacobi | - | - | 4 | - | - | -0.5 | 0.0 | 0.9167 |

Finally, some kernels of support vector machine algorithms are summarized in Table 6.7. A brief explanation of introduced kernels in this book is also presented for each one that highlights the most notable characteristics of them and gives a suitable comparison list to glance at and compare the introduced orthogonal kernels of this book.

6.5 Summary and Conclusion

Jacobi orthogonal polynomials have been covered in this chapter which is the most general kind of classical orthogonal polynomials and has been used in many cases. In fact, the basics and the properties of these polynomials are explained, and also the ordinary Jacobi kernel function is constructed. Moreover, the fractional form of Jacobi polynomials and the relevant fractional Jacobi kernel function are introduced here which extended the applicability of such kernel by transforming the input data into a fractional space that has proved to leverage the accuracy of classification. Also, a comprehensive comparison is provided between the introduced Jacobi kernels and all other kernels discussed in Chapters 3, 4, and 5. The experiments showed the efficiency of Jacobi kernels that made them a suitable kernel function for kernel-based learning algorithms such as SVM.

Table 6.7: A summary of SVM kernels

| Kernel function | Description |
|-----------------------------|---|
| Legendre | The Legendre kernel is presented in [27] and has considerable merit, which does not require a weight function. Although it is immune to the explosion effect, it suffers from the annihilation effect. |
| Chebyshev | This kernel was presented by Ye et al. in [31]. They reported it as the first orthogonal polynomial kernel for SVM classifiers. Also, it is not immune to both explosion and annihilation effects. |
| Generalized Chebyshev | This kernel was proposed by Ozer et al. in [32], and it was the first orthogonal kernel with vector formulation. By introducing this kernel, they found an effective way to avoid the annihilation effect. |
| Modified Chebyshev | This kernel was introduced by Ozer et al. [32] and has the same characteristics as the generalized Chebyshev kernel. The vector form of this kernel has better performance in facing the nonlinear problems in comparison with the generalized Chebyshev kernel. |
| Chebyshev-Wavelet | Introduced by Jafarzadeh et al. at [16] using mercer rules, which in fact was the multiplications of Chebyshev and wavelet kernels and showed to have competitive accuracy. |
| Unified Chebyshev | Zhao et al. [33] introduced a new kernel through unifying Chebyshev kernels of the first and second kind. A new kernel with two notifiable properties, orthogonality, and adaptivity. This kernel was also shown to have low computational cost and competitive accuracy. |
| Gegenbauer | It was introduced in [34] by Padierna et al., it is not only immune to both annihilation and explosion effects, but also it involves Chebyshev and Legendre kernels and improves them as special cases. |
| Generalized Gegenbauer | Introduced by Yang et al. [35] originally to address an engineering problem of state recognition of bolted structures. showed considerable accuracy in that specific context. |
| Regularized Jacobi Wavelets | This kernel function was introduced by Abassa in [19]. Although such kernel has high time complexity, still provides competitive results compared to other kernels. |

References

1. Ezz-Eldien, S. S., Doha, E. H.: Fast and precise spectral method for solving pantograph type Volterra integro-differential equations. Numer. Algorithms **81**, 57–77 (2019)
2. Doha, E. H., Bhrawy, A. H., Ezz-Eldien, S. S.: Efficient Chebyshev spectral methods for solving multi-term fractional orders differential equations. Appl. Math. Model. **35**, 5662–5672 (2011)

3. Parand, K., Rad, J. A., Ahmadi, M.: A comparison of numerical and semi-analytical methods for the case of heat transfer equations arising in porous medium. *The European Physical Journal Plus* **131**, 1–15 (2016)
4. Parand, K., Moayeri, M. M., Latifi, S., Rad, J. A.: Numerical study of a multidimensional dynamic quantum model arising in cognitive psychology especially in decision making. *The European Physical Journal Plus* **134**, 109 (2019)
5. Moayeri, M. M., Rad, J. A., Parand, K.: Desynchronization of stochastically synchronized neural populations through phase distribution control: a numerical simulation approach. *Nonlinear Dynamics* **104**, 2363–2388 (2021)
6. Garnier, H., Mensler, M. I. C. H. E. L., Richard, A. L. A. I. N.: Continuous-time model identification from sampled data: implementation issues and performance evaluation. *Int. J. Control* **76**, 1337–1357 (2003)
7. Ping, Z., Ren, H., Zou, J., Sheng, Y., Bo, W.: Generic orthogonal moments: Jacobi–Fourier moments for invariant image description. *Pattern Recognit.* **40**, 1245–1254 (2007)
8. Kazem, S.: An integral operational matrix based on Jacobi polynomials for solving fractional-order differential equations. *Appl. Math. Model.* **37**, 1126–1136 (2013)
9. Shojaeizadeh, T., Mahmoudi, M., Darehmiraki, M.: Optimal control problem of advection-diffusion-reaction equation of kind fractal-fractional applying shifted Jacobi polynomials. *Chaos Solitons Fractals* **143**, 110568 (2021)
10. Bhrawy, A. H., Alofi, A. S.: Jacobi–Gauss collocation method for solving nonlinear Lane–Emden type equations. *Commun. Nonlinear Sci. Numer. Simul.* **17**, 62–70 (2012)
11. Bhrawy, A. H., Zaky, M. A.: Shifted fractional-order Jacobi orthogonal functions: application to a system of fractional differential equations. *Appl. Math. Model.* **40**, 832–845 (2016)
12. Bhrawy, A., Zaky, M.: A fractional-order Jacobi Tau method for a class of time-fractional PDEs with variable coefficients. *Math. Methods Appl. Sci.* **39**, 1765–1779 (2016)
13. Abdelkawy, M. A., Amin, A. Z., Bhrawy, A. H., Machado, J. A. T., Lopes, A. M.: Jacobi collocation approximation for solving multi-dimensional Volterra integral equations. *Int. J. Nonlinear Sci. Numer. Simul.* **18**, 411–425 (2017)
14. Nkengfack, L. C. D., Tchiotsop, D., Atangana, R., Louis-Door, V., Wolf, D.: Classification of EEG signals for epileptic seizures detection and eye states identification using Jacobi polynomial transforms-based measures of complexity and least-square support vector machine. *Inform. Med. Unlocked* **23**, 100536 (2021)
15. Morris, G. R., Abed, K. H.: Mapping a Jacobi iterative solver onto a high-performance heterogeneous computer. *IEEE Trans Parallel Distrib Syst* **24**, 85–91 (2012)
16. Jafarzadeh, S. Z., Aminian, M., Efati, S.: A set of new kernel function for support vector machines: An approach based on Chebyshev polynomials. In: *ICCKE* 412–416 (2013)
17. Szeg, G.: Orthogonal polynomials. American Mathematical Society, Rhode Island (1939)
18. Guo, B. Y., Shen, J., Wang, L. L.: Generalized Jacobi polynomials/functions and their applications. *Appl Numer Math* **59**, 1011–1028 (2009)
19. Nadira, A., Abdessamad, A., Mohamed, B. S.: Regularized Jacobi Wavelets Kernel for Support Vector Machines. *Stat. Optim. Inf. Comput.* **7**, 669–685 (2019)
20. Vapnik, V.: The nature of statistical learning theory. Springer, Berlin (2013)
21. Abdallah, N. B., Chouchene, F.: New recurrence relations for Wilson polynomials via a system of Jacobi type orthogonal functions. *J. Math. Anal. Appl.* **498**, 124978 (2021)
22. Askey, R., Wilson, J. A.: Some basic hypergeometric orthogonal polynomials that generalize Jacobi polynomials **319**, American Mathematical Soc. (1985)
23. Askey, R.: Orthogonal polynomials and special functions. Society for industrial and applied mathematics, Pennsylvania (1975)
24. Milovanovic, G. V., Rassias, T. M., Mitrinovic, D. S.: Topics In Polynomials: Extremal Problems, Inequalities, Zeros. World Scientific, Singapore (1994)
25. Hadian Rasanan, A. H., Bajalan, N., Parand, K., Rad, J. A.: Simulation of nonlinear fractional dynamics arising in the modeling of cognitive decision making using a new fractional neural network. *Math. Methods Appl. Sci.* **43**, 1437–1466 (2020)
26. Upneja, R., Singh, C.: Fast computation of Jacobi–Fourier moments for invariant image recognition. *Pattern Recognit.* **48**, 1836–1843 (2015)

27. Pan, Z. B., Chen, H., You, X. H.: Support vector machine with orthogonal Legendre kernel. International Conference on Wavelet Analysis and Pattern Recognition. 125–130 (2012)
28. Boyd, J. P.: Chebyshev and Fourier spectral methods. Courier Corporation, Massachusetts (2001)
29. Mastroianni, G., Milovanovic, G.: Interpolation processes: Basic theory and applications. Springer Science & Business Media, Berlin (2008)
30. Asghari, M., Hadian Rasanan, A.H., Gorgin, S., Rahmati, D., Parand, K.: FPGA-orthopoly: a hardware implementation of orthogonal polynomials. *Eng Comput* (2022) doi: 10.1007/s00366-022-01612-x
31. Ye, N., Sun, R., Liu, Y., Cao, L.: Support vector machine with orthogonal Chebyshev kernel. In 18th International Conference on Pattern Recognition (ICPR'06) **2** 752–755 (2006)
32. Ozer, S., Chen, C. H., Cirpan, H. A.: A set of new Chebyshev kernel functions for support vector machine pattern classification. *Pattern Recognit* **44**, 1435–1447 (2011)
33. Zhao, J., Yan, G., Feng, B., Mao, W., Bai, J.: An adaptive support vector regression based on a new sequence of unified orthogonal polynomials. *Pattern Recognit* **46**, 899–913 (2013)
34. Padíerna, L. C., Carpio, M., Rojas-Domínguez, A., Puga, H., Fraire, H.: A novel formulation of orthogonal polynomial kernel functions for SVM classifiers: the Gegenbauer family. *Pattern Recognit.* **84**, 211–225 (2018)
35. Yang, W., Zhang, Z., Hong, Y.: State recognition of bolted structures based on quasi-analytic wavelet packet transform and generalized Gegenbauer support vector machine. In 2020 IEEE International Instrumentation and Measurement Technology Conference (I2MTC) 1–6 (2020)
36. Moayeri, M. M., Hadian Rasanan, A. H., Latifi, S., Parand, K., Rad, J. A.: An efficient space-splitting method for simulating brain neurons by neuronal synchronization to control epileptic activity. *Eng Comput* (2020) doi: 10.1007/s00366-020-01086-9
37. Khodabandehlo, H. R., Shivanian, E., Abbasbandy, S.: Numerical solution of nonlinear delay differential equations of fractional variable-order using a novel shifted Jacobi operational matrix. *Eng Comput* (2021) doi: 10.1007/s00366-021-01422-7
38. Doha, E. H., Bhrawy, A. H., Ezz-Eldien, S. S.: A new Jacobi operational matrix: an application for solving fractional differential equations. *Appl. Math. Model.* **36**, 4931–4943 (2012)
39. Bhrawy, A. H.: A Jacobi spectral collocation method for solving multi-dimensional nonlinear fractional sub-diffusion equations. *Numer. Algorithms* **73**, 91–113 (2016)
40. Bhrawy, A. H., Doha, E. H., Saker, M. A., Baleanu, D.: Modified Jacobi–Bernstein basis transformation and its application to multi-degree reduction of Bézier curves. *J. Comput. Appl. Math.* **302**, 369–384 (2016)
41. Bhrawy, A. H., Hafez, R. M., Alzaidy, J. F.: A new exponential Jacobi pseudospectral method for solving high-order ordinary differential equations. *Adv Differ Equ* **2015**, 1–15 (2015)
42. Doman, B. G. S.: The classical orthogonal polynomials. World Scientific, Singapore (2015)
43. Tian, M., Wang, W.: Some sets of orthogonal polynomial kernel functions. *Appl. Soft Comput.* **61**, 742–756 (2017)
44. Imani, A., Aminataei, A., Imani, A.: Collocation method via Jacobi polynomials for solving nonlinear ordinary differential equations. *International Journal of Mathematics and Mathematical Sciences* **2011**, 673085 (2011)
45. Khader, M. M., Adel, M.: Chebyshev wavelet procedure for solving FLDEs. *Acta Appl. Math.* **158**, 1–10 (2018)
46. Razzaghi, M., Yousefi, S.: Legendre wavelets method for constrained optimal control problems. *Math. Methods Appl. Sci.* **25**, 529–539 (2002)
47. Elaydi, H. A., Abu Haya, A.: Solving optimal control problem for linear time invariant systems via Chebyshev wavelet. *Int. J. Electr. Eng.* **5**, (2012)
48. Bokhari, A., Amir, A., Bahri, S. M.: A numerical approach to solve quadratic calculus of variation problems. *Dyn. Contin. Discrete Impuls. Syst.* **25**, 427–440 (2018)

Part III

Applications of orthogonal kernels

Chapter 7

Solving Ordinary Differential Equations by LS-SVM

Mohsen Razzaghi and Simin Shekarpaz and Alireza Rajabi

Abstract In this chapter, we propose a machine learning method for solving a class of linear ordinary differential equations (ODEs) which is based on the least squares-support vector machines (LS-SVM) with collocation procedure. One of the most important and practical models in this category is Lane-Emden type equations. By using LS-SVM for solving these types of equations, the solution is expanded based on rational Legendre functions and the LS-SVM formulation is presented. Based on this, the linear problems are solved in dual form and a system of linear algebraic equations is concluded. Finally, by presenting some numerical examples, the results of the current method are compared with other methods. The comparison shows that the proposed method is fast and highly accurate with exponential convergence.

7.1 Introduction

Differential equations are a kind of mathematical equation that can be used for modeling many physical and engineering problems in real life, such as dynamics of oscillators, cosmology, and study of the solar system, the study of unsteady gases, fluid dynamics, and many other applications (see. e.g., [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]). In the meantime, Lane-Emden type equations are an important class of ordinary differential equations on the semi-infinite domain which was introduced

Mohsen Razzaghi
Department of Mathematics and Statistics, Mississippi State University, USA e-mail: razzaghi@math.msstate.edu

Simin Shekarpaz
Department of Applied Mathematics, Brown University, Providence, RI, 02912, USA, e-mail: Simin_Shekarpaz@Brown.edu

Reza Rajabi
Department of Computer and Data science, Faculty of Mathematical Sciences, Shahid Beheshti University, Tehran, Iran, e-mail: al.rajabi@mail.sbu.ac.ir

by Jonathan Homer Lane and by Emden as a model for the temperature of the sun [16, 17]. These equations have found many interesting applications in modeling many physical phenomena such as the theory of stellar structure, thermal behavior of a spherical cloud of gas, isothermal gas spheres, and the theory of thermionic currents [18, 19]. As one of the important applications, it can be said that in astrophysics, this equation describes the equilibrium density distribution in the self-gravitating sphere of polytropic isothermal gas.

The general form of the Lane-Emden equation is as follows [52, 39, 53]:

$$\begin{aligned} y''(x) + \frac{k}{x}y'(x) + f(x, y(x)) &= h(x), \\ y(x_0) = A, \quad y'(x_0) &= B, \end{aligned} \tag{7.1}$$

where A, B are constants and $f(x, y)$ and $h(x)$ are given functions of x and y .

Some of these equations do not have an exact solution and as a result of their singularity at $x = 0$, their numerical solutions are a challenge for scientists [40, 36]. Many semi-analytical and numerical approaches have been applied to solve the Lane-Emden equations, which can be presented as follows: In [20], a new perturbation technique based on an artificial parameter δ proposed to solve the Lane-Emden equation. Also, a non-perturbative analytical solution of this equation was derived in [21] by Adomian decomposition method (ADM). On the other hand, Mandelzweig et al. [22] used the quasilinearization method for solving the standard Lane-Emden equation, and Liao [23] produced an analytical framework based on the Adomian decomposition method for Lane-Emden type equations. The approach based on semi-analytical methods continued as He [24] obtained the analytical solutions to the problem by using Ritz's variational method. Also, in [25], the modified decomposition method for the analytic behavior of nonlinear differential equations was used, and Yildirim et al. [27] obtained approximate solutions of a class of Lane-Emden equations by homotopy perturbation method (HPM). Following this approach, Ramos [28] solved Lane-Emden equations by using a linearization method and a series solution has been suggested in [29]. Those solutions were obtained by writing this equation as a Volterra integral equation and assuming that the nonlinearities are smooth. Then, Dehghan [30, 31] proposed the Adomian decomposition method for differential equations with an alternate procedure to overcome the difficulty of singularity. Also, Aslanov [32] introduced an improved Adomian decomposition method for non-homogeneous, linear, and non-linear Lane-Emden type equations. On the other hand, Dehghan and Shakeri [33] used an exponential transformation for the Lane-Emden type equations to overcome the difficulty of a singular point at $x = 0$ and the resulting nonsingular problem solved by the variational iteration method (VIM). The use of two important analytical approaches, namely homotopy analysis method (HAM) and HPM, seems to have been so common. For example, Singh et al. [35] proposed an efficient analytic algorithm for the Lane-Emden type equations using modified homotopy analysis method. Also, Bataineh et al. [37] proposed a homotopy analysis method to obtain the analytical solutions of the singular IVPs of the Emden-Fowler type, and in [38], Chowdhury et al. applied an algorithm based

on the homotopy perturbation method to solve singular IVPs of the Emden-Fowler type. However, robust numerical approaches have also been used for these studies. In [26], a numerical method was provided for solving the Lane-Emden equations. In that work, by using the integral operator and Legendre wavelet approximation, the problem is converted into an integral equation. Also, Marzban et al. [34] applied the properties of hybrid of block-pulse functions together with the Lagrange interpolating polynomials for solving the nonlinear second-order initial value problems and the Lane-Emden equation. Then, Parand et al. [36] presented a numerical technique based on the rational Legendre approach to solving higher ordinary differential equations such as Lane-Emden. In [39], Parand et al. have applied an algorithm for the nonlinear Lane-Emden problems, using Hermite functions collocation methods. On the other hand, Pandey and Kumar [40] presented an efficient and accurate method for solving Lane-Emden type equations arising in astrophysics using Bernstein polynomials. Recently, in [41], the linear Lane-Emden equations solved by the Bessel collocation method and error of problem was also obtained. More recently, Parand and Khaleqi [42] also suggested rational Chebyshev of second kind collocation method to solve the Lane-Emden equation. For analysis of the Lane-Emden equation based on the Lie symmetry approach, we refer the interested reader to [43, 44]. This important model has been carefully examined from a simulation point of view with more numerical methods (see [45, 46]), but due to the weaknesses of the performance of these approaches in complex dynamic models, one of the current trends for solving and simulating dynamic models based on ODEs and partial differential equations (PDEs) is neural networks (NN). In fact, NN is a machine learning technique to obtain the numerical solution of higher-order ODE and partial differential equations (PDEs) which has been proposed by many researchers. For example, Lagaris et al [47] solved some well-known ODEs by using a neural network. Also, in [48], a hybrid numerical method based on an artificial neural network and optimization technique was studied for solving directly higher-order ODEs. More importantly, a regression-based artificial neural network method was analyzed by Chakravarty and Mall in [49] to solve ODEs where a single layer Chebyshev neural network was suggested to solve the Lane-Emden equation. On the other hand, Mall and Chacraverty [50] solved Emden-Fowler equations by using Chebyshev neural network. They also used a Legendre neural network to solve the Lane-Emden equation [51]. More recently, partial differential equation [53], system, and fractional [52] versions of the Lane-Emden equation are solved by orthogonal neural networks.

In this chapter, the LS-SVM algorithm together with the rational Legendre functions as kernel functions is used to solve the Lane-Emden equation. In our proposed method, the constraints are the collocation form of residual function. Then the coefficients of approximate solutions together with the error function are minimized. To obtain the solution of the minimization problem, the Lagrangian function is used and the problem is solved in dual form.

The remainder of this chapter is organized as follows. In Section 2, the LS-SVM formulation is discussed which is used to solve differential equations. The kernel trick is also introduced in Section 3 and their properties together with the operational matrix of differentiation are presented. In Section 4, the LS-SVM formulation of the

Lane-Emden equation is given and the problem is solved in dual form. Finally, the numerical results are presented which show the efficiency of the proposed technique.

7.2 LS-SVM formulation

We consider the general form of an m -th order initial value problem (IVP), which is as follows:

$$\begin{aligned} L[y(x)] - F(x) &= 0, \quad x \in [a, c], \\ y^{(i)}(a) &= p_i, \quad i = 0, 1, \dots, m-1, \end{aligned} \tag{7.2}$$

where L represents an ordinary differential operator, i.e.

$$L[y(x)] = L(x, y(x), y'(x), \dots, y^{(m)}(x)),$$

and $F(x)$ and $\{p_i\}_{i=0}^{m-1}$ are known. The aim is to find a learning solution to this equation by using the LS-SVM formulation.

We assume that we have training data $\{(x_i, y_i)\}_{i=0}^N$, in which $x_i \in \mathbb{R}$ is the input data and $y_i \in \mathbb{R}$ is output data. To solve the ODE, one-dimensional data is sufficient. Then, the solution to our regression problem is approximated by $y(x) = \mathbf{w}^T \boldsymbol{\varphi}(x)$, where $\mathbf{w} = [w_0, \dots, w_N]^T$, and $\boldsymbol{\varphi}(x) = [\varphi_0(x), \varphi_1(x), \dots, \varphi_N(x)]^T$ is a vector of arbitrary basis functions. It is worth mentioning that the basis functions are selected in a way that they satisfy the initial conditions. The primal LS-SVM model is as follows (see, e.g, [54]):

$$\begin{aligned} \text{minimize}_{\mathbf{w}, \boldsymbol{\varepsilon}} \quad & \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon}, \\ \text{subject to} \quad & y_i = \mathbf{w}^T \boldsymbol{\varphi}(x_i) + \varepsilon_i, \quad i = 0, 1, \dots, N, \end{aligned} \tag{7.3}$$

where ε_i is the error of the i -th training point, $\mathbf{w} \in \mathbb{R}^h$, $\boldsymbol{\varphi}(\cdot) : \mathbb{R} \rightarrow \mathbb{R}^h$ is a feature map, and h is the dimension of feature space. In this formulation, the term $\frac{1}{2} \mathbf{w}^T \mathbf{w}$ can be interpreted to obtain the stable solutions.

The problem with this formulation is that when solving the differential equation, we don't have the value of y_i and therefore, we put the differential equation in our model. The collocation form of equations is used as the constraint of our optimization problem.

To solve the optimization problem, the kernel trick is used and the problem is transformed into dual form.

7.2.1 Collocation form of LS-SVM

In the case that LS-SVM is used to solve a linear ODE, the approximate solution is obtained by solving the following optimization problem [55, 58]:

$$\begin{aligned} & \text{minimize}_{\mathbf{w}, \boldsymbol{\varepsilon}} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon}, \\ & \text{subject to} \quad L[\tilde{y}_i] - F(x_i) = \varepsilon_i, \quad i = 0, 1, \dots, N, \end{aligned} \quad (7.4)$$

where $\boldsymbol{\varepsilon} = [\varepsilon_0, \dots, \varepsilon_N]^T$ and \tilde{y} is the approximate solution, which is written in terms of the basis functions, to satisfy the initial conditions. The collocation points $\{x_i\}_{i=0}^N$ are also chosen to be the roots of rational Legendre functions in the learning phase.

Now we introduce a dual form of the problem:

Theorem 7.1 [55] Suppose that φ_i be the basis functions and $\boldsymbol{\alpha} = [\alpha_0, \alpha_1, \dots, \alpha_N]^T$ be the Lagrangian coefficients in the dual form of the minimization problem Eq. 7.4, then the solution of minimization problem is as follows:

$$\mathbf{w} = M\boldsymbol{\alpha} = \begin{pmatrix} L(\varphi_0(x_0)) & L(\varphi_0(x_1)) & \dots & L(\varphi_0(x_N)) \\ L(\varphi_1(x_0)) & \dots & \dots & L(\varphi_1(x_N)) \\ \dots & \dots & \dots & \dots \\ L(\varphi_N(x_0)) & L(\varphi_N(x_1)) & \dots & L(\varphi_N(x_N)) \end{pmatrix} \begin{pmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_N \end{pmatrix}, \quad \boldsymbol{\varepsilon} = \frac{\boldsymbol{\alpha}}{\gamma} = \begin{pmatrix} \frac{1}{\gamma}\alpha_0 \\ \frac{1}{\gamma}\alpha_1 \\ \vdots \\ \frac{1}{\gamma}\alpha_N \end{pmatrix},$$

where:

$$(M^T M + \frac{1}{\gamma} I)\boldsymbol{\alpha} = \mathbf{F}, \quad (7.5)$$

and $\mathbf{F} = [F(x_0), F(x_1), \dots, F(x_N)]^T$.

Proof First, we construct the Lagrangian function of our optimization problem as follows:

$$G = \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon} - \sum_{i=0}^N \alpha_i \left(\sum_{j=0}^N w_j L(\varphi_j(x_i)) - F_i - \varepsilon_i \right), \quad (7.6)$$

where $\alpha_i \geq 0$ ($i = 0, 1, \dots, N$) are Lagrange multipliers. To obtain the optimal solution, by using the Karush-Kuhn-Tucker (KKT) optimality conditions [55, 56, 57], one gets:

$$\begin{aligned}\frac{\partial G}{\partial w_k} &= w_k - \sum_{i=0}^N \alpha_i (L(\varphi_k(x_i))) = 0, \\ \frac{\partial G}{\partial \varepsilon_k} &= \gamma \varepsilon_k + \alpha_k = 0, \\ \frac{\partial G}{\partial \alpha_k} &= \sum_{j=0}^N w_j L(\varphi_j(x_k)) - F_k - \varepsilon_k = 0.\end{aligned}\tag{7.7}$$

So, the solution is obtained by the following relations,

$$\begin{aligned}w_k &= \sum_{i=0}^N M_{ki} \alpha_i, \quad (M_{ki} = (L(\varphi_k(x_i)))), \quad k = 0, 1, 2, \dots, N, \\ \mathbf{w} &= M\boldsymbol{\alpha},\end{aligned}\tag{7.8}$$

and by using the second relation in Eq. 7.7, one obtains:

$$\varepsilon_k = \frac{-\alpha_k}{\gamma}, \quad \Rightarrow \boldsymbol{\varepsilon} = \frac{-1}{\gamma} \boldsymbol{\alpha}.\tag{7.9}$$

Considering the third equation, results in,

$$\begin{aligned}\sum_{j=0}^N w_j L(\varphi_j(x_k)) - F_k - \varepsilon_k &= 0, \\ (M^T \mathbf{w})_k - F_k - \varepsilon_k &= 0, \quad \Rightarrow M^T \mathbf{w} - \boldsymbol{\varepsilon} = \mathbf{F}.\end{aligned}\tag{7.10}$$

So, by using Eqs. 7.8-7.10, we have:

$$M^T M \boldsymbol{\alpha} + \frac{1}{\gamma} \boldsymbol{\alpha} = \mathbf{F}, \quad \Rightarrow (M^T M + \frac{1}{\gamma} I) \boldsymbol{\alpha} = \mathbf{F},\tag{7.11}$$

where $F_i = F(x_i)$ and so, the vector of Lagrange multipliers $\boldsymbol{\alpha}$ is obtained. \square

7.3 Rational Legendre kernels

Kernel function has an efficient rule in using LS-SVM, therefore the choice of kernel function is important. These kernel functions can be constructed by using orthogonal polynomials. The operational matrices of orthogonal polynomials are sparse and the derivatives are obtained exactly which makes our method fast and it leads to well-posed systems. Given that the properties of Legendre polynomials, fractional Legendre functions, as well as the Legendre kernel functions are discussed in Chapter 4, we recommend that readers refer to that chapter of the book for review, and as a result, in this section, we will focus more on the rational Legendre kernels. In particular, Guo et al. [59] introduced a new set of rational Legendre functions which

are mutually orthogonal in $L_2(0, +\infty)$ with the weight function $w(x) = \frac{2L}{(L+x)^2}$, as follow:

$$RP_n(x) = P_n\left(\frac{x-L}{x+L}\right). \quad (7.12)$$

Thus,

$$\begin{aligned} RP_0(x) &= 1, & RP_1(x) &= \frac{x-L}{x+L}, \\ nRP_n(x) &= (2n-1)\left(\frac{x-L}{x+L}\right)RP_{n-1}(x) - (n-1)RP_{n-2}(x), & n \geq 2, \end{aligned} \quad (7.13)$$

where $\{P_n(x)\}$ are Legendre polynomials which were defined in Chapter 5. These functions are used to solve problems on semi-infinite domains, and they can also produce sparse matrices and have a high convergence rate [59, 60].

Since the range of Legendre polynomials is $[-1, 1]$, we have $|RP_n(x)| \leq 1$. The operational matrix of the derivative is also a lower Hessenberg matrix which can be calculated as [60, 36]:

$$D = \frac{1}{L}(D_1 + D_2), \quad (7.14)$$

where D_1 is a tridiagonal matrix

$$D_1 = \text{Diag}\left(\frac{7t^2 - i - 2}{2(2i+1)}, -i, \frac{i(i+1)}{2(2i+1)}\right), \quad i = 0, \dots, n-1,$$

and $D_2 = [d_{ij}]$ such that,

$$d_{ij} = \begin{cases} 0, & j \geq i-1, \\ (-1)^{i+j+1}(2j+1), & j < i-1. \end{cases}$$

The rational Legendre kernel function for non-vector data x and z is recommended as follow,

$$K(x, z) = \sum_{i=0}^N RP_i(x)RP_i(z). \quad (7.15)$$

This function is a valid SVM kernel, if that satisfies the conditions of the Mercer theorem (see. e.g., [54]).

Theorem 7.2 *To be a valid SVM kernel, for any finite function $g(x)$, the following integration should always be nonnegative for the given kernel function $K(x, z)$:*

$$\int \int K(x, z)g(x)g(z)dxdz \geq 0. \quad (7.16)$$

Proof Consider $g(x)$ to be a function and $K(x, z)$ that was defined in Eq. 7.15, then by using the Mercer condition, one obtains

$$\begin{aligned}
\int \int K(x, z)g(x)g(z)dx dz &= \int \int \sum_{i=0}^N RP_i(x)RP_i(z)g(x)g(z)dx dz \\
&= \sum_{i=0}^N \left(\int \int RP_i(x)RP_i(z)g(x)g(z)dx dz \right) \\
&= \sum_{i=0}^N \left(\int RP_i(x)g(x)dx \int RP_i(z)g(z)dz \right) \quad (7.17) \\
&= \sum_{i=0}^N \left(\int RP_i(x)g(x)dx \int RP_i(x)g(x)dx \right) \\
&= \sum_{i=0}^N \left(\int RP_i(x)g(x)dx \right)^2 \geq 0.
\end{aligned}$$

Therefore, the proposed function is a valid kernel. \square

7.4 Collocation form of LS-SVM for Lane-Emden type equations

In this section, we implement the proposed method to solve the Lane-Emden type equations. As we said before, these types of equations are an important class of ordinary differential equations in the semi-infinite domain. The general form of the Lane-Emden equation is as follows:

$$\begin{aligned}
y''(x) + \frac{k}{x}y'(x) + f(x, y(x)) &= h(x), \\
y(x_0) = A, \quad y'(x_0) = B,
\end{aligned} \quad (7.18)$$

where A, B are constants and $f(x, y(x))$ and $h(x)$ are given functions of x and y . Since the function $f(x, y(x))$ can be both linear or nonlinear, in this chapter, we assume that this function can be reformed to $f(x, y(x)) = f(x)y(x)$. In this case, there is no need to any linearization method and the LS-SVM method can be directly applied on the Eq. 7.18. For the nonlinear cases, quasi-linearization method can be applied on the Eq. 7.18 first and then the solution is approximated by using the LS-SVM algorithm. Now we can consider the LS-SVM formulation of this equation, which is:

$$\begin{aligned}
\text{minimize}_{\boldsymbol{w}, \boldsymbol{\varepsilon}} \quad & \frac{1}{2} \boldsymbol{w}^T \boldsymbol{w} + \frac{\gamma}{2} \boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon}, \\
\text{subject to} \quad & \frac{d^2 \tilde{y}_i}{dx^2} + \frac{k}{x_i} \left(\frac{d \tilde{y}_i}{dx} \right) + f(x_i) \tilde{y}(x_i) - h(x_i) = \varepsilon_i, \quad k > 0,
\end{aligned} \quad (7.19)$$

where $\tilde{y}_i = \tilde{y}(x_i)$ is the approximate solution. We use the Lagrangian function and then,

$$G = \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \boldsymbol{\varepsilon}^T \boldsymbol{\varepsilon} - \sum_{i=0}^N \alpha_i \left(\frac{d^2 \tilde{y}_i}{dx^2} + \frac{k}{x_i} \left(\frac{d \tilde{y}_i}{dx} \right) + f(x_i) \tilde{y}(x_i) - h(x_i) - \varepsilon_i \right). \quad (7.20)$$

So, by expanding the solution in terms of the rational Legendre kernels, one obtains:

$$\begin{aligned} G &= \frac{1}{2} \sum_{i=0}^N w_i^2 + \frac{\gamma}{2} \sum_{i=0}^N \varepsilon_i^2 - \sum_{i=0}^N \alpha_i \left(\sum_{j=0}^N w_j \varphi_j''(x_i) \right. \\ &\quad \left. + \frac{k}{x_i} \sum_{j=0}^N w_j \varphi_j'(x_i) + f(x_i) \left(\sum_{j=0}^N w_j \varphi_j(x_i) \right) - h(x_i) - \varepsilon_i \right), \end{aligned} \quad (7.21)$$

where $\{\varphi_j\}$ are the rational Legendre kernels. Then by employing the Karush-Kuhn-Tucker (KKT) optimality conditions for $0 \leq l \leq N$, we conclude that

$$\begin{aligned} \frac{\partial G}{\partial w_l} &= w_l - \sum_{i=0}^N \alpha_i \left(\varphi_l''(x_i) + \frac{k}{x_i} \varphi_l'(x_i) + f(x_i) \varphi_l(x_i) \right) = 0, \\ \frac{\partial G}{\partial \alpha_l} &= \sum_{j=0}^N w_j \varphi_j''(x_l) + \frac{k}{x_l} \sum_{j=0}^N w_j \varphi_j'(x_l) + f(x_l) \left(\sum_{j=0}^N w_j \varphi_j(x_l) \right) - h(x_l) - \varepsilon_l = 0, \\ \frac{\partial G}{\partial \varepsilon_l} &= \gamma \varepsilon_l + \alpha_l = 0, \end{aligned} \quad (7.22)$$

and, by using the above relations:

$$\begin{aligned} w_l &= \sum_{i=0}^N \alpha_i \left(\varphi_l''(x_i) + \frac{k}{x_i} \varphi_l'(x_i) + f(x_i) \varphi_l(x_i) \right) \\ &= \sum_{i=0}^N \alpha_i \psi_l(x_i), \end{aligned} \quad (7.23)$$

where $\psi_l(x_i) = \varphi_l''(x_i) + \frac{k}{x_i} \varphi_l'(x_i) f(x_i) \varphi_l(x_i)$. Now, we can show the matrix form as follows:

$$\begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \psi_0(x_0) & \psi_0(x_1) & \psi_0(x_2) & \dots & \psi_0(x_N) \\ \psi_1(x_0) & \psi_1(x_1) & \psi_1(x_2) & \dots & \psi_1(x_N) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \psi_N(x_0) & \psi_N(x_1) & \psi_N(x_2) & \dots & \psi_N(x_N) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix},$$

which can be summarized as $\mathbf{w} = M\boldsymbol{\alpha}$ where $M_{li} = \{\psi_l(x_i)\}_{0 \leq l, i \leq N}$. By using the third relation in Eq. 7.22,

$$\forall l : \epsilon_l = \frac{-\alpha_l}{\gamma}, \quad (7.24)$$

and substituting w_j in the second equation of Eq. 7.22, results in:

$$\sum_{j=0}^N \left(\sum_{i=0}^N \alpha_i \psi_j(x_i) \right) \psi_j(x_l) + \frac{\alpha_l}{\gamma} = h(x_l), \quad 0 \leq l \leq N, \quad (7.25)$$

and in matrix form, one gets,

$$\begin{bmatrix} \psi_0(x_0) & \psi_1(x_0) & \dots & \psi_N(x_0) \\ \psi_0(x_1) & \psi_1(x_1) & \dots & \psi_N(x_1) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_0(x_N) & \psi_1(x_N) & \dots & \psi_N(x_N) \end{bmatrix} \begin{bmatrix} \psi_0(x_0) & \psi_0(x_1) & \dots & \psi_0(x_N) \\ \psi_1(x_0) & \psi_1(x_1) & \dots & \psi_1(x_N) \\ \vdots & \vdots & \ddots & \vdots \\ \psi_N(x_0) & \psi_N(x_1) & \dots & \psi_N(x_N) \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix} \\ + \begin{bmatrix} \frac{1}{\gamma} & 0 & \dots & 0 \\ 0 & \frac{1}{\gamma} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{1}{\gamma} \end{bmatrix} \begin{bmatrix} \alpha_0 \\ \alpha_1 \\ \vdots \\ \alpha_N \end{bmatrix} = \begin{bmatrix} h_0 \\ h_1 \\ \vdots \\ h_N \end{bmatrix},$$

and can be written as $(\mathbf{M}^T \mathbf{M} + \frac{1}{\gamma} I)\alpha = h$ where $I = I_{N+1 \times N+1}$, $h_i = h(x_i)$ and $\mathbf{h} = [h_0, h_1, \dots, h_N]^T$.

Now, we need to define the derivatives of the kernel function $K(x_l, x_i) = [\varphi(x_l)]^T \varphi(x_i)$. Making use of Mercer's Theorem [61, 62], derivatives of the feature map can be written in terms of derivatives of the kernel function. Let us consider the following differential operator

$$\nabla^{m,n} = \frac{\partial^{m+n}}{\partial u^m \partial v^n},$$

which will be used in this section.

We define $\chi^{(m,n)}(u, v) = \nabla^{m,n} K(u, v)$ and $\chi_{l,i}^{(m,n)} = \nabla^{m,n} K(u, v)|_{u=x(l), v=x(i)}$, and Eq. 7.25 is written as follows,

$$\begin{aligned} h_l = \sum_{i=0}^N \alpha_i & [\chi_{l,i}^{(2,2)} + \frac{k}{x_l} \chi_{l,i}^{(1,2)} + f_l \chi_{l,i}^{(0,2)} + \frac{k}{x_i} \chi_{l,i}^{(2,1)} + \frac{k^2}{x_i x_l} \chi_{l,i}^{(1,1)} + \frac{k}{x_i} f_l \chi_{l,i}^{(0,1)} \\ & + f_i \chi_{l,i}^{(2,0)} + f_i \frac{k}{x_l} \chi_{l,i}^{(1,0)} + f_i f_l \chi_{l,i}^{(0,0)}] + \frac{\alpha_l}{\gamma}, \quad 0 \leq l \leq N. \end{aligned} \quad (7.26)$$

So we calculate $\{\alpha_i\}_{0 \leq i \leq N}$ and by substituting in Eq. 7.23, $\{w_l\}_{0 \leq l \leq N}$ is computed, and

$$\tilde{y} = \sum_{i=0}^n \alpha_i (\chi^{(2,0)}(x_i, x) + \frac{k}{x_i} \chi^{(1,0)}(x_i, x) + f_i \chi^{(0,0)}(x_i, x)), \quad (7.27)$$

will be the dual form of an approximate solution in the Lane-Emden equation.

7.5 Numerical Examples

In this section, the collocation form of least squares support vector regressions is applied for solving various form of Lane-Emden type equations based on the rational Legendre kernels.

The value of the regularization parameter γ is effective in the performance of the LS-SVM model. Based on experience the larger regularization parameter results in a smaller error. Therefore, the chosen value for γ is 10^8 in the presented examples which obtained by trial and error.

For solving this problem by using the proposed method, we train our algorithm by using the shifted roots of Legendre polynomials, and the testing points are considered to be the set of equidistant points on the domain. Then the training and testing errors are computed which have been defined by $e = y - \tilde{y}$. In order to show the efficiency and capability of the proposed method, the numerical approximations are compared with the other obtained results. Also, the convergence of our method is also illustrated by numerical results.

Test case 1: [63, 64, 65] Let us consider $f(x, y) = -2(2x^2 + 3)y$, $k = 2$, $h(x) = 0$, $A = 1$, and $B = 0$ in Eq. 7.18, then the linear Lane-Emden equation is as follows:

$$\begin{aligned} y''(x) + \frac{2}{x}y'(x) - 2(2x^2 + 3)y &= 0, \quad x \geq 0, \\ y(0) = 1, \quad y'(0) &= 0. \end{aligned} \tag{7.28}$$

The exact solution is $y(x) = e^{x^2}$. This type of equation has been solved with linearization, VIM, and HPM methods (see e.g., [63, 64, 65]).

By using the proposed method, the numerical results of this test case in $[0, 1]$ have been obtained which are depicted in Fig. 7.1 with 30 training points. The results and related training error function with 180 training points for solving the problem in $[0, 2]$ are also shown in Fig. 7.2.

The testing error by using 50 equidistant points for solving the problem in intervals $[0, 1]$ and $[0, 2]$ are displayed in Fig. 7.3 (a) and Fig. 7.3 (b), where the maximum testing error in this example is 2.23×10^{-13} and 4.46^{-14} , respectively. Moreover, the absolute errors for arbitrary testing data have been computed and presented in Table 7.1 for $N = 180$ and the optimal value of $L = 4.59$.

Table 7.1: The absolute errors of present method for testing points in $[0,2]$ with $N = 180$ and $L = 4.59$ (Test case 1)

| Testing data | Error | Exact value |
|--------------|--------------------------|-------------|
| 0.00 | 0.00000 | 1.000000000 |
| 0.01 | 2.2730×10^{-17} | 1.000100005 |
| 0.02 | 3.6336×10^{-18} | 1.000400080 |
| 0.05 | 1.0961×10^{-16} | 1.002503127 |
| 0.10 | 1.4026×10^{-16} | 1.010050167 |
| 0.20 | 1.1115×10^{-15} | 1.040810774 |
| 0.50 | 7.4356×10^{-16} | 1.284025416 |
| 0.70 | 2.8378×10^{-15} | 1.632316219 |
| 0.80 | 1.5331×10^{-15} | 1.896480879 |
| 0.90 | 8.3772×10^{-15} | 2.247907986 |
| 1.00 | 1.8601×10^{-14} | 2.718281828 |
| 1.1 | 4.0474×10^{-15} | 3.353484653 |
| 1.2 | 2.6672×10^{-14} | 4.220695816 |
| 1.5 | 3.9665×10^{-14} | 9.487735836 |
| 1.7 | 9.3981×10^{-15} | 17.99330960 |
| 1.8 | 4.1961×10^{-14} | 25.53372174 |
| 1.9 | 3.0212×10^{-14} | 36.96605281 |
| 2.0 | 3.6044×10^{-16} | 54.59815003 |

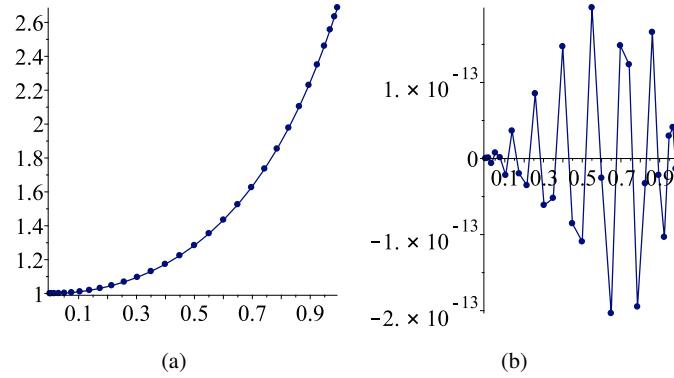


Fig. 7.1: (a) Numerical results for training points in $[0, 1]$ (b) Obtained training errors (Test case 1)

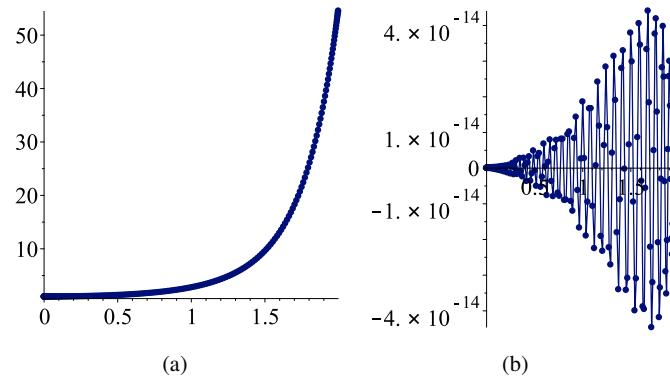


Fig. 7.2: (a) Numerical results for training points in $[0, 2]$ (b) Obtained training errors (Test case 1)

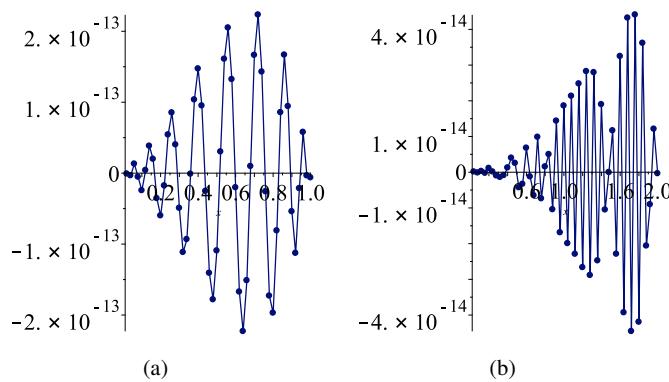


Fig. 7.3: Error function for testing points (a) in $[0, 1]$ with $M = 50$ and $N = 30$ and (b) in $[0, 2]$ with $M = 50$ and $N = 180$ (Test case 1)

The maximum norm of testing errors with different numbers of basis functions (training points) has also been presented in Table 7.2, which indicates the convergence of the method for solving this kind of linear Lane-Emden equation.

Table 7.2: Maximum norm of errors for testing data in $[0, 2]$ with $M = 50$ and different values of N (Test case 1)

| N | Error norm | N | Error norm |
|----|------------------------|-----|------------------------|
| 12 | 1.41×10^{-1} | 80 | 2.79×10^{-11} |
| 20 | 2.51×10^{-4} | 100 | 3.03×10^{-12} |
| 30 | 1.31×10^{-6} | 120 | 6.41×10^{-13} |
| 40 | 4.20×10^{-8} | 140 | 2.15×10^{-13} |
| 50 | 5.69×10^{-9} | 150 | 1.30×10^{-13} |
| 60 | 4.77×10^{-10} | 180 | 4.46×10^{-14} |

Test case 2: [63, 66, 67, 68] Considering $f(x, y) = xy$, $h(x) = x^5 - x^4 + 44x^2 - 30x$, $k = 8$, $A = 0$, and $B = 0$ in Eq. 7.18, we have the linear Lane-Emden equation,

$$\begin{aligned} y''(x) + \frac{8}{x}y'(x) + xy(x) &= x^5 - x^4 + 44x^2 - 30x, \quad x \geq 0, \\ y(0) = 0, \quad y'(0) &= 0, \end{aligned} \quad (7.29)$$

which has the exact solution $y(x) = x^4 - x^3$ and has been solved with linearization, HPM, HAM, and two-step ADM (TSADM) methods (see, e.g., [63, 66, 67, 68]). By applying Eqs. 7.26-7.27, the approximate solutions are calculated. The numerical solutions together with the training error function by using 30 points in $[0, 10]$ have been presented in Fig. 7.4.

The proposed algorithm is also tested with some arbitrary points in Table 7.3 for $N = 60$ and $L = 18$. Moreover, in Fig. 7.5, the error function in equidistant testing data has been plotted. The maximum norm of error by using 50 testing points is 7.77×10^{-12} .

In Table 7.4, the norm of testing error for different values of N has been obtained. We can see that, with increasing the number of training points, the testing error decreases, which shows the good performance and convergence of our algorithm to solve this example.

Table 7.3: The absolute errors of present method for testing points with $N = 60$ and $L = 18$ (Test case 2)

| Testing data | Error | Exact value |
|--------------|--------------------------|----------------|
| 0.00 | 0 | 0.00000000000 |
| 0.01 | 4.8362×10^{-16} | -0.0000009900 |
| 0.10 | 1.4727×10^{-15} | -0.0009000000 |
| 0.50 | 1.7157×10^{-15} | -0.0625000000 |
| 1.00 | 7.8840×10^{-15} | 0.00000000000 |
| 2.00 | 2.7002×10^{-15} | 8.00000000000 |
| 3.00 | 2.4732×10^{-14} | 54.000000000 |
| 4.00 | 2.2525×10^{-14} | 192.000000000 |
| 5.00 | 4.1896×10^{-14} | 500.000000000 |
| 6.00 | 6.3632×10^{-15} | 1080.000000000 |
| 7.00 | 5.4291×10^{-14} | 2058.000000000 |
| 8.00 | 7.0818×10^{-14} | 3584.000000000 |
| 9.00 | 1.0890×10^{-14} | 5832.000000000 |
| 10.00 | 6.5032×10^{-16} | 9000.000000000 |

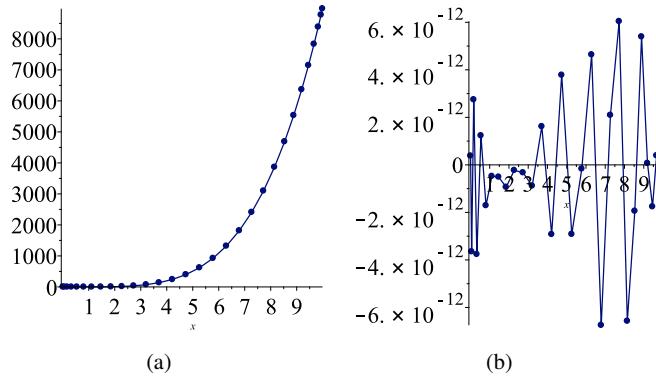


Fig. 7.4: (a) Numerical results for training points in $[0, 10]$ (b) Obtained training errors (Test case 2)

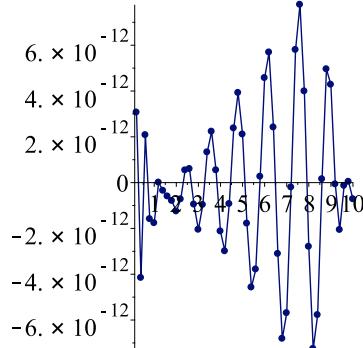


Fig. 7.5: Error function for 50 equidistant testing points with $N = 30$ and $L = 18$ (Test case 2)

Table 7.4: Maximum norm of testing errors obtained for $M = 50$ and $L = 18$ with different values of N (Test case 2)

| N | Error norm |
|----|------------------------|
| 8 | 3.78×10^{-1} |
| 12 | 2.14×10^{-4} |
| 20 | 1.31×10^{-9} |
| 30 | 7.77×10^{-12} |
| 40 | 1.32×10^{-12} |
| 50 | 2.30×10^{-13} |
| 60 | 7.08×10^{-14} |

Test case 3: [63, 64, 65, 68] In this test case, we consider $f(x, y) = y$, $h(x) = 6 + 12x + x^2 + x^3$, $k = 2$, $A = 0$, and $B = 0$ in the general form of the Lane-Emden equation, then we have:

$$\begin{aligned} y''(x) + \frac{2}{x}y'(x) + y(x) &= 6 + 12x + x^2 + x^3, \quad x \geq 0, \\ y(0) &= 0, \quad y'(0) = 0, \end{aligned} \tag{7.30}$$

which has the exact solution $y(x) = x^2 + x^3$. This example has also been solved in [63, 64, 65, 68] with linearization, VIM, HPM and TSADM methods, respectively.

The proposed method is used and the numerical solutions of this example are obtained in 30 training points which can be seen in Fig. 7.6. It should be noted that the optimal value of $L = 14$ has been used in this example. The testing error function is also shown in Fig. 7.7 with 50 equidistant points. In Table 7.5, the numerical

results in arbitrary testing data with $N = 60$ have been reported which show the efficiency of the LS-SVM model for solving this kind of problem. The maximum norm of testing errors with different values of N and $M = 50$ is recorded in Table 7.6 and so, the convergence of the LS-SVM model is concluded.

Table 7.5: The absolute errors of present method in testing points with $N = 60$ and $L = 14$ (Test case 3)

| Testing data | Error | Exact value |
|--------------|--------------------------|-------------------|
| 0.00 | 2.9657×10^{-28} | 0.000000000000 |
| 0.01 | 3.6956×10^{-17} | 0.0001010000 |
| 0.10 | 2.1298×10^{-16} | 0.0110000000 |
| 0.50 | 6.8302×10^{-17} | 0.3750000000 |
| 1.00 | 8.2499×10^{-17} | 2.000000000000 |
| 2.00 | 8.3684×10^{-17} | 12.000000000000 |
| 3.00 | 6.7208×10^{-17} | 36.000000000000 |
| 4.00 | 2.2338×10^{-17} | 80.000000000000 |
| 5.00 | 1.5048×10^{-16} | 150.000000000000 |
| 6.00 | 3.9035×10^{-17} | 252.000000000000 |
| 7.00 | 1.3429×10^{-16} | 392.000000000000 |
| 8.00 | 2.4277×10^{-17} | 576.000000000000 |
| 9.00 | 4.1123×10^{-17} | 810.000000000000 |
| 10.00 | 6.5238×10^{-18} | 1100.000000000000 |

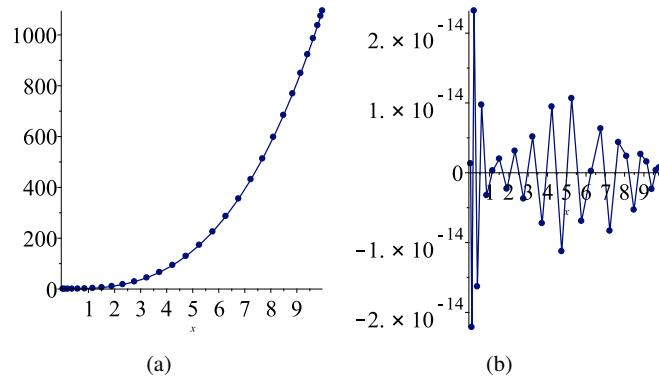


Fig. 7.6: (a) Numerical results with training points in $[0, 10]$ (b) obtained training errors (Test case 3)

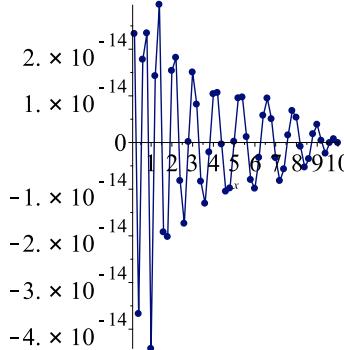


Fig. 7.7: Error function in equidistant testing points with $N = 30$ and $L = 14$ (Test case 3)

Table 7.6: Maximum norm of testing errors for $M = 50$ and $L = 14$ with different values of N (Test case 3)

| N | Error norm |
|----|------------------------|
| 8 | 4.88×10^{-2} |
| 12 | 5.85×10^{-5} |
| 20 | 9.15×10^{-11} |
| 30 | 4.42×10^{-14} |
| 40 | 2.61×10^{-15} |
| 50 | 3.89×10^{-16} |
| 60 | 1.82×10^{-16} |

Test case 4: [52, 53] (Standard Lane-Emden equation) Considering $f(x, y) = y^m$, $k = 2$, $h(x) = 0$, $A = 1$, and $B = 0$ in Eq. 7.18, then the standard Lane-Emden equation is:

$$\begin{aligned} y''(x) + \frac{2}{x}y'(x) + y^m(x) &= 0, \quad x \geq 0, \\ y(0) &= 1, \quad y'(0) = 0, \end{aligned} \tag{7.31}$$

where $m \geq 0$ is a constant. Substituting $m = 0, 1$ and 5 into Eq. 7.31 leads to the following exact solutions:

$$y(x) = 1 - \frac{1}{3!}x^2, \quad y(x) = \frac{\sin(x)}{x} \quad \text{and} \quad y(x) = (1 + \frac{x^2}{3})^{-\frac{1}{2}}, \tag{7.32}$$

respectively. By applying the LS-SVM formulation to solve the standard Lane-Emden equation, the approximate solutions are calculated. The numerical solutions of this example for $m = 0$ and 30 training points together with the error function are shown in Fig. 7.8. For testing our algorithm, based on 50 equidistant points, the obtained error function is presented in Fig. 7.9 with 30 training points. Moreover, the numerical approximations for arbitrary testing data have been reported in Table 7.7, which shows the accuracy of our proposed method.

Table 7.7: The absolute errors of proposed method in testing points with $m = 0$, $N = 60$ and $L = 30$ (Test case 4)

| Testing data | Error | Exact value |
|--------------|--------------------------|-------------|
| 0 | 0.00000 | 1.00000000 |
| 0.1 | 1.2870×10^{-22} | 0.99833333 |
| 0.5 | 4.3059×10^{-22} | 0.95833333 |
| 1.0 | 7.0415×10^{-22} | 0.83333333 |
| 5.0 | 1.1458×10^{-20} | -3.16666666 |
| 6.0 | 7.4118×10^{-21} | -5.00000000 |
| 6.8 | 1.5090×10^{-20} | -6.70666666 |

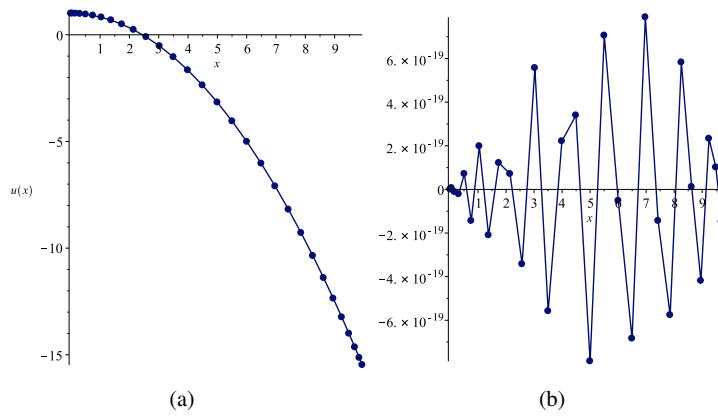


Fig. 7.8: Numerical results with training points in $[0, 10]$ (a) and obtained training errors (b) for $m = 0$ (Test case 4)

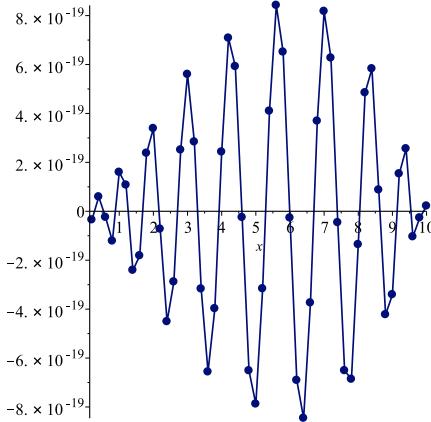


Fig. 7.9: Error function in equidistant testing points with $m = 0$, $N = 30$ and $L = 30$ (Test case 4)

The numerical results for $m = 1$ have been presented in Fig. 7.10 with 30 training points. Its related error function has also been plotted. The testing error function is displayed in Fig. 7.11, where the maximum norm of error is equal to 1.56×10^{-12} . The numerical results for arbitrary values of testing data are also shown in Table 7.8 for $m = 1$ and $N = 60$.

Table 7.8: The absolute errors of present method in testing points with $m = 1$, $N = 60$ and $L = 30$ (Test case 4)

| Testing data | Error | Exact value |
|--------------|------------------------|---------------|
| 0 | 1.80×10^{-32} | 1.0000000000 |
| 0.1 | 6.28×10^{-21} | 0.9983341665 |
| 0.5 | 1.52×10^{-20} | 0.9588510772 |
| 1.0 | 4.27×10^{-20} | 0.8414709848 |
| 5.0 | 2.86×10^{-19} | -0.1917848549 |
| 6.0 | 3.71×10^{-19} | -0.0465692497 |
| 6.8 | 1.52×10^{-19} | 0.0726637280 |

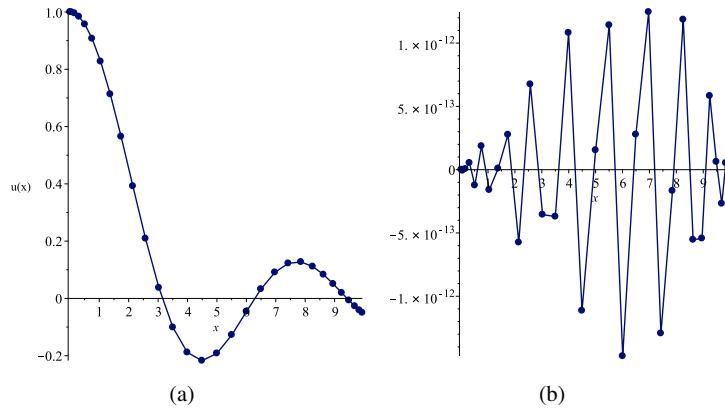


Fig. 7.10: Numerical results with training points in $[0, 10]$ (a) and obtained training errors (b) for $m = 1$ (Test case 4)

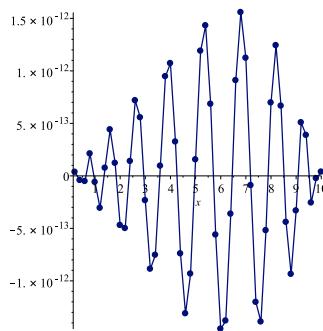


Fig. 7.11: Error function in equidistant testing points with $m = 1$, $N = 30$ and $L = 30$ (Test case 4)

By using this model and for $m = 0, 1$, the maximum norm of errors for different numbers of training points have been reported in Table 7.9, which show the convergence of the method.

Table 7.9: Maximum norm of testing errors for $M = 50$ and $L = 30$ with different values of N (Test case 4)

| N | Error norm ($m = 0$) | Error norm ($m = 1$) |
|-----|------------------------|------------------------|
| 8 | 7.99×10^{-7} | 9.42×10^{-3} |
| 12 | 4.30×10^{-9} | 6.10×10^{-4} |
| 20 | 6.78×10^{-18} | 2.60×10^{-9} |
| 30 | 8.46×10^{-19} | 1.56×10^{-12} |
| 40 | 8.71×10^{-20} | 2.91×10^{-15} |
| 50 | 5.81×10^{-20} | 1.73×10^{-17} |
| 60 | 1.54×10^{-20} | 3.72×10^{-19} |

Conclusion

In this chapter, we developed the least squares support vector machine model for solving various forms of Lane-Emden type equations. For solving the this problem, the collocation LS-SVM formulation was applied and the problem was solved in dual form. The rational Legendre functions were also employed to construct kernel function, because of their good properties to approximate functions on semi-infinite domains.

We used the shifted roots of Legendre polynomial for training our algorithm and equidistance points as the testing points. The numerical results by applying the training and testing points, show the accuracy of our proposed model. Moreover, the exponential convergence of our proposed method was achieved by choosing the different numbers of training points and basis functions. In other words, when the number of training points is increased, the norm of errors decreased exponentially.

References

1. Liu, Q. X., Liu, J. K., Chen, Y. M.: A second-order scheme for nonlinear fractional oscillators based on Newmark- β algorithm. *J. Comput. Nonlinear Dyn.* **13**, 084501 (2018)
2. Rodrigues, C., Simoes, F. M., da Costa, A. P., Froio, D., Rizzi, E.: Finite element dynamic analysis of beams on nonlinear elastic foundations under a moving oscillator. *Eur J Mech A Solids* **68**, 9–24 (2018)
3. Anderson, D., Yunes, N., Barausse, E.: Effect of cosmological evolution on Solar System constraints and on the scalarization of neutron stars in massless scalar-tensor theories. *Phys. Rev. D* **94**, 104064 (2016)
4. Khouri, J., Sakstein, J., Solomon, A. R.: Superfluids and the cosmological constant problem. *J. Cosmol. Astropart. Phys.* **2018**, 024 (2018)
5. Farzaneh-Gord, M., Rahbari, H. R.: Unsteady natural gas flow within pipeline network, an analytical approach. *J. Nat. Gas Sci. Eng.* **28**, 397–409 (2016)

6. Lusch, B., Kutz, J. N., Brunton, S. L.: Deep learning for universal linear embeddings of nonlinear dynamics. *Nat. Commun* **9**, 1–10 (2018)
7. Bristeau, M. O., Pironneau, O., Glowinski, R., Periaux, J., Perrier, P.: On the numerical solution of nonlinear problems in fluid dynamics by least squares and finite element methods (I) least square formulations and conjugate gradient solution of the continuous problems. *Comput Methods Appl Mech Eng* **17**, 619–657 (1979)
8. Parand, K., Abbasbandy, S., Kazem, S., Rad, J. A.: A novel application of radial basis functions for solving a model of first-order integro-ordinary differential equation. *Communications in Nonlinear Science and Numerical Simulation* **16**, 4250–4258 (2011)
9. Parand, K., Rad, J. A.: Exp-function method for some nonlinear PDE's and a nonlinear ODE's. *Journal of King Saud University-Science* **24**, 1–10 (2012)
10. Kazem, S., Rad, J. A., Parand, K., Abbasbandy, S.: A new method for solving steady flow of a third-grade fluid in a porous half space based on radial basis functions. *Zeitschrift für Naturforschung A* **66**, 591–598 (2011)
11. Parand, K., Hossayni, S. A., Rad, J. A.: An operation matrix method based on Bernstein polynomials for Riccati differential equation and Volterra population model. *Applied Mathematical Modelling* **40**, 993–1011 (2016)
12. Kazem, S., Rad, J. A., Parand, K., Shaban, M., Saberi, H.: The numerical study on the unsteady flow of gas in a semi-infinite porous medium using an RBF collocation method. *International Journal of Computer Mathematics* **89**, 2240–2258 (2012)
13. Parand, K., Nikarya, M., Rad, J. A., Baharifard, F.: A new reliable numerical algorithm based on the first kind of Bessel functions to solve Prandtl-Blasius laminar viscous flow over a semi-infinite flat plate. *Zeitschrift für Naturforschung A* **67** 665–673 (2012)
14. Parand, K., Lotfi, Y., Rad, J. A.: An accurate numerical analysis of the laminar two-dimensional flow of an incompressible Eyring-Powell fluid over a linear stretching sheet. *The European Physical Journal Plus* **132**, 1–21 (2017)
15. Abbasbandy, S., Modarrespoor, D., Parand, K., Rad, J. A.: Analytical solution of the transpiration on the boundary layer flow and heat transfer over a vertical slender cylinder. *Quaestiones Mathematicae* **36**, 353–380 (2013)
16. Lane, H. J.: On the theoretical temperature of the sun, under the hypothesis of a gaseous mass maintaining its volume by its internal heat, and depending on the laws of gases as known to terrestrial experiment. *Am. J. Sci.* **2**, 57–74 (1870)
17. Emden, R.: *Gaskugeln: Anwendungen der mechanischen Warmetheorie auf kosmologische und meteorologische Probleme*, BG Teubner, (1907)
18. Chandrasekhar, S., Chandrasekhar, S.: *An introduction to the study of stellar structure* (Vol. 2). North Chelmsford, Courier Corporation, (1957)
19. Wood, D. O.: *The Emission of Electricity from Hot Bodies. Monographs on physics*, Longmans, Green and Company, (1921)
20. Bender, C. M., Milton, K. A., Pinsky, S. S., Simmons Jr, L. M.: A new perturbative approach to nonlinear problems. *J. Math. Phys.* **30**, 1447–1455 (1989)
21. Shawagfeh, N. T.: Nonperturbative approximate solution for Lane–Emden equation. *J. Math. Phys.* **34**, 4364–4369 (1993)
22. Mandelzweig, V. B., Tabakin, F.: Quasilinearization approach to nonlinear problems in physics with application to nonlinear ODEs. *Comput. Phys. Commun.* **141**, 268–281 (2001)
23. Liao, S.: A new analytic algorithm of Lane–Emden type equations. *Appl. Math. Comput.* **142**, 1–16 (2003)
24. He, J. H.: Variational approach to the Lane–Emden equation. *Appl. Math. Comput.* **143**, 539–541 (2003)
25. Wazwaz, A. M.: The modified decomposition method for analytic treatment of differential equations. *Appl. Math. Comput.* **173**, 165–176 (2006)
26. Yousefi, S. A.: Legendre wavelets method for solving differential equations of Lane–Emden type. *Appl. Math. Comput.* **181**, 1417–1422 (2006)
27. Yıldırım, A., Özış, T.: Solutions of singular IVPs of Lane–Emden type by Homotopy perturbation method. *Phys. Lett. A* **369**, 70–76 (2007)

28. Ramos, J. I.: Linearization techniques for singular initial-value problems of ordinary differential equations. *Appl. Math. Comput.* **161**, 525–542 (2005)
29. Ramos, J. I.: Series approach to the Lane–Emden equation and comparison with the Homotopy perturbation method. *Chaos Solitons Fractals* **38**, 400–408 (2008)
30. Dehghan, M., Tatari, M.: The use of Adomian decomposition method for solving problems in calculus of variations. *Math. Probl. Eng.* **2006**, 1–12 (2006)
31. Dehghan, M., Shakeri, F.: The use of the decomposition procedure of Adomian for solving a delay differential equation arising in electrodynamics. *Phys. Scr.* **78**, 065004 (2008)
32. Aslanov, A.: Determination of convergence intervals of the series solutions of Emden–Fowler equations using polytropes and isothermal spheres. *Phys. Lett. A* **372**, 3555–3561 (2008)
33. Dehghan, M., Shakeri, F.: Approximate solution of a differential equation arising in astrophysics using the variational iteration method. *New Astron.* **13**, 53–59 (2008)
34. Marzban, H. R., Tabrizidooz, H. R., Razzaghi, M.: Hybrid functions for nonlinear initial-value problems with applications to Lane–Emden type equations. *Phys. Lett. A* **372**, 5883–5886 (2008)
35. Singh, O. P., Pandey, R. K., Singh, V. K.: An analytic algorithm of Lane–Emden type equations arising in astrophysics using modified Homotopy analysis method. *Comput. Phys. Commun.* **180**, 1116–1124 (2009)
36. Parand, K., Shahini, M., Dehghan, M.: Rational Legendre pseudospectral approach for solving nonlinear differential equations of Lane–Emden type. *J. Comput. Phys.* **228**, 8830–8840 (2009)
37. Bataineh, A. S., Noorani, M. S. M., Hashim, I.: Homotopy analysis method for singular IVPs of Emden–Fowler type. *Commun Nonlinear Sci Numer Simul* **14**, 1121–1131 (2009)
38. Chowdhury, M. S. H., Hashim, I.: Solutions of Emden–Fowler equations by Homotopy-perturbation method. *Nonlinear Anal. Real World Appl.* **10**, 104–115 (2009)
39. Parand, K., Dehghan, M., Rezaei, A. R., Ghaderi, S. M.: An approximation algorithm for the solution of the nonlinear Lane–Emden type equations arising in astrophysics using Hermite functions collocation method. *Comput. Phys. Commun.* **181**, 1096–1108 (2010)
40. Parand, K., Pirkhedri, A.: Sinc-collocation method for solving astrophysics equations. *New Astron.* **15**, 533–537 (2010)
41. Yüzbaşı, Ş., Sezer, M.: An improved Bessel collocation method with a residual error function to solve a class of Lane–Emden differential equations. *Math. comput. model.* **57**, 1298–1311 (2013)
42. Parand, K., Khaleqi, S.: The rational Chebyshev of second kind collocation method for solving a class of astrophysics problems. *Eur. Phys. J. Plus* **131**, 1–24 (2016)
43. Kara, A. H., Mahomed, F. M.: Equivalent lagrangians and the solution of some classes of non-linear equations... *Int J Non Linear Mech* **27**, 919–927 (1992)
44. Kara, A. H., Mahomed, F. M.: A note on the solutions of the Emden–Fowler equation. *Int J Non Linear Mech* **28**, 379–384 (1993)
45. Hossayni, S. A., Rad, J. A., Parand, K., Abbasbandy S.: Application of the exact operational matrices for solving the Emden–Fowler equations, arising in astrophysics. *International Journal of Industrial Mathematics* **7**, 351–374 (2015)
46. Parand, K., Nikarya, M., Rad, J. A.: Solving non-linear Lane–Emden type equations using Bessel orthogonal functions collocation method. *Celestial Mechanics and Dynamical Astronomy* **116**, 97–107 (2013)
47. Lagaris, I. E., Likas, A., Fotiadis, D. I.: Artificial neural networks for solving ordinary and partial differential equations. *IEEE trans. neural netw.* **9**, 987–1000 (1998)
48. Malek, A., Beidakhti, R. S.: Numerical solution for high order differential equations using a hybrid neural network—optimization method. *Appl. Math. Comput.* **183**, 260–271 (2006)
49. Mall, S., Chakraverty, S.: Chebyshev neural network based model for solving Lane–Emden type equations. *Appl. Math. Comput.* **247**, 100–114 (2014)
50. Mall, S., Chakraverty, S.: Numerical solution of nonlinear singular initial value problems of Emden–Fowler type using Chebyshev Neural Network method. *Neurocomputing* **149**, 975–982 (2015)
51. Mall, S., Chakraverty, S.: Application of Legendre neural network for solving ordinary differential equations. *Appl. Soft Comput.* **43**, 347–356 (2016)

52. Hadian Rasanan, A. H., Rahmati, D., Gorgin, S., Parand, K.: A single layer fractional orthogonal neural network for solving various types of Lane–Emden equation. *New Astron.* **75**, 101307, (2020)
53. Omidi, M., Arab, B., Hadian Rasanan, A. H., Rad, J. A., Parand, K.: Learning nonlinear dynamics with behavior ordinary/partial/system of the differential equations: looking through the lens of orthogonal neural networks. *Eng Comput* 1–20 (2021)
54. Suykens, J. A. K., Van Gestel, T., De Brabanter, J., De Moor, B., Vandewalle, J.: Least squares support vector machines. World Scientific, New Jersey (2002)
55. Mehrkanoon, S., Falck, T., Suykens, J. A.: Approximate solutions to ordinary differential equations using least squares support vector machines. *IEEE Trans Neural Netw Learn Syst* **23**, 1356–1367 (2012)
56. Mehrkanoon, S., Suykens, J. A.: LS-SVM based solution for delay differential equations. In *Journal of Physics: Conference Series* **410**, 012041 (2013)
57. Cortes, C., Vapnik, V.: Support-vector networks. *Machine learning* **20**, 273–297 (1995)
58. Pakniyat, A., Parand, K., Jani, M.: Least squares support vector regression for differential equations on unbounded domains. *Chaos Solitons Fractals*, **151**, 111232 (2021)
59. Guo, B. Y., Shen, J., Wang, Z. Q.: A rational approximation and its applications to differential equations on the half line. *J. Sci. Comput.* **15**, 117–147 (2000)
60. Parand, K., Razzaghi, M.: Rational Legendre approximation for solving some physical problems on semi-infinite intervals. *Phys. Scr.* **69**, 353–357 (2004)
61. Vapnik, V.: Statistical Learning Theory. Wiley, New York (1998)
62. Lázaro, M., Santamaría, I., Pérez-Cruz, F., Artés-Rodríguez, A.: Support vector regression for the simultaneous learning of a multivariate function and its derivatives. *Neurocomputing* **69**, 42–61 (2005)
63. Ramos, J. I.: Linearization techniques for singular initial-value problems of ordinary differential equations. *Appl. Math. Comput.* **161**, 525–542 (2005)
64. Yıldırım, A., Özış, T.: Solutions of singular IVPs of Lane–Emden type by the variational iteration method. *Nonlinear Anal Theory Methods Appl* **70**, 2480–2484 (2009)
65. Chowdhury, M. S. H., Hashim, I.: Solutions of a class of singular second-order IVPs by Homotopy-perturbation method. *Phys. Lett. A* **365**, 439–447 (2007)
66. Chowdhury, M. S. H., Hashim, I.: Solutions of Emden–Fowler equations by Homotopy-perturbation method. *Nonlinear Anal. Real World Appl.* **10**, 104–115 (2009)
67. Bataineh, A. S., Noorani, M. S. M., Hashim, I.: Homotopy analysis method for singular IVPs of Emden–Fowler type. *Commun Nonlinear Sci Numer Simul* **14**, 1121–1131 (2009)
68. Zhang, B. Q., Wu, Q. B., Luo, X. G.: Experimentation with two-step Adomian decomposition method to solve evolution models. *Appl. Math. Comput.* **175**, 1495–1502 (2006)
69. Wazwaz, A. M.: A new algorithm for solving differential equations of Lane–Emden type. *Appl. Math. Comput.* **118**, 287–310 (2001)
70. Liao, S.: A new analytic algorithm of Lane–Emden type equations. *Appl. Math. Comput.* **142**, 1–16 (2003)
71. Singh, O. P., Pandey, R. K., Singh, V. K.: An analytic algorithm of Lane–Emden type equations arising in astrophysics using modified Homotopy analysis method. *Comput. Phys. Commun.* **180**, 1116–1124 (2009)
72. Ramos, J. I.: Series approach to the Lane–Emden equation and comparison with the homotopy perturbation method. *Chaos Solitons Fractals* **38**, 400–408 (2008)
73. Aslanov, A.: A generalization of the Lane–Emden equation. *Int J Comput Math* **85**, 1709–1725 (2008)
74. Horedt, G. P.: Polytropes: applications in astrophysics and related fields. Klawer Academic Publishers, New York, (2004)

Chapter 8

Solving Partial Differential Equations by LS-SVM

Mohammad Mahdi Moayeri and Mohammad Hemami

Abstract In recent years, much attention has been paid to machine learning-based numerical approaches due to their applications in solving difficult high dimensional problems. In this chapter, a numerical method based on support vector machines is proposed to solve second-order time-dependent partial differential equations. This method is called the least squares support vector machines (LS-SVM) collocation approach. In this approach, first, the time dimension is discretized by the Crank-Nicolson algorithm, then, the optimal representation of the solution is obtained in the primal setting. Using KKT optimality conditions, the dual formulation is derived, and at the end, the problem is converted to a linear system of algebraic equations that can be solved by standard solvers. The Fokker-Planck and generalized Fitzhugh-Nagumo equations are considered as test cases to demonstrate the proposed effectiveness of the scheme. Moreover, two kinds of orthogonal kernel functions are introduced for each example, and their performances are compared.

8.1 Introduction

Nowadays, most mathematical models of problems in science have more than one independent variables which usually represents time and space variables [43, 75, 76, 5]. These models lead to partial differential equations (PDEs). However, these equations usually do not have exact/analytical solutions due to their complexity. Thus, numerical methods can help us to approximate the solution of equations and simulate models. Let us consider the following general form of a second-order PDE:

Mohammad Mahdi Moayeri

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran. e-mail: mahdi.myr@gmail.com

Mohammad Hemami

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran. e-mail: gaslakh@gmail.com

$$A \frac{\partial^2 u}{\partial x^2} + B \frac{\partial^2 u}{\partial x \partial y} + C \frac{\partial^2 u}{\partial y^2} + D \frac{\partial u}{\partial x} + E \frac{\partial u}{\partial y} + Fu + G = 0, \quad (8.1)$$

where, A, B, C, D, E, F , and G can be a constant, or a function of x , y , and u . It is clear when $G = 0$, we have a homogeneous equation, and otherwise, we have a nonhomogeneous equation. According to these coefficients, there are three kinds of PDEs [1]. This equation is called elliptic when $B^2 - 4AC < 0$, parabolic when $B^2 - 4AC = 0$, and hyperbolic when $B^2 - 4AC > 0$. In science, most of the problems which include time dimension, are parabolic or hyperbolic such as heat and wave equations [1]. In addition, the Laplace equation is also an example of the elliptic type equation[2, 3]. On the other hand, if A, B, C, D, E or F be a function of dependent variable u , the equation converts to a nonlinear PDE which is usually more complex than the linear one.

Since analytical and semi-analytical methods can be used only for simple differential equations, scientists have developed various numerical approaches to solve more complex and nonlinear ordinary/partial differential equations. These approaches can be divided into six general categories as:

- Finite difference method (FDM)[1, 4]: The intervals are discretized to a finite number of steps and the derivatives are approximated by finite difference formulas. Usually, the approximate solution of a problem is obtained by solving a system of algebraic equations obtained by employing FDM on PDE. Some examples of this method are implicit method[5, 6], explicit method[1, 4], Crank-Nicolson method[7, 8], etc. [9, 10, 11].
- Finite element method (FEM)[12, 13]: This approach is based on mesh generating. In fact, FEM subdivides the space into smaller parts called finite elements. Then, the variation calculus and low order local basis functions are used in these elements. Finally, all sets of element equations are combined into a global system of equations for the final calculation. Different kinds of FEMs are standard FEM[14, 15], nonconforming FEM[16, 17], mixed FEM[18, 19], discontinuous FEM[20, 21], etc. [24, 25, 26, 27].
- Finite volume method (FVM)[21, 23]: FVM is similar to FEM and needs mesh construction. In this approach, these mesh elements are called control volumes. In fact, in this method, volume integrals are used with the divergence theorem. An additional feature is the local conservative of the numerical fluxes; that is, the numerical flux is conserved from one discretization cell to its neighbour. Due to this property, FVM is appropriate for the models where the flux is of importance[28, 29]. Actually, a finite volume method evaluates exact expressions for the average value of the solution over some volume, and uses this data to construct approximations of the solution within cells [30]. Some famous FVMs are cell-centered FVM[31, 32], vertex-centered FVM[33, 34], Petrov-Galerkin FVM[35, 36], etc. [37, 38, 39].

- Spectral method [88]: Unlike previous approaches, the spectral method is a high-order global method. In the spectral method, the solution of the differential equation is considered as a finite sum of basis functions as $\sum_{i=0}^n a_i \phi_i(x)$. Usually, basis functions $\phi_i(x)$ are one of the orthogonal polynomials because of their orthogonality properties which makes the calculations easier [138]. Now, different strategies are developed to calculate the coefficients in the sum in order to satisfy the differential equation as well as possible. Generally, in simple geometries for smooth problems, the spectral methods offer exponential rates of convergence/spectral accuracy[40, 41]. This method is divided into three main categories: collocation [106, 107], Galerkin [42, 90], and Petrov-Galerkin methods [89, 108], etc. [43, 44, 45, 46, 47].
- Meshfree method [48, 49]: The meshfree method is used to establish a system of algebraic equations for the whole problem domain without the use of a pre-defined mesh, or uses easily generable meshes in a much more flexible or freer manner. It can be said that meshfree methods essentially use a set of nodes scattered within the problem domain as well as on the boundaries to represent the problem domain and its boundaries. The field functions are then approximated locally using these nodes [49]. Meshfree methods have been considered by many researchers in the last decade due to their high flexibility and high accuracy of the numerical solution[50, 51]. However, there are still many challenges and questions about these methods that need to be answered [52, 53], such as optimal shape parameter selection in methods based on radial basis functions and enforcing boundary conditions in meshfree methods. Some of meshfree methods are radial basis function approach (RBF)[54, 55, 56, 57, 58, 59, 60], radial basis function generated finite difference (RBF-FD)[61, 62], meshfree local Petrov-Galerkin (MLPG)[50, 63, 64, 65, 66], element free Galerkin (EFG)[67, 68], meshfree local radial point interpolation method (MLRPIM)[69, 70, 71, 72], etc. [73, 74, 75, 76].
- Machine learning-based methods[77, 78]: With the growth of available scientific data and improving machine learning approaches, recently, researchers of scientific computing have been trying to develop machine learning and deep learning algorithms for solving differential equations[79, 80]. Especially, they have had some successful attempts to solve some difficult problems that common numerical methods are not well able to solve such as PDEs by noisy observations or very high-dimension PDEs. Some of the recent machine learning-based numerical methods are physics informed neural networks (PINN)[81, 82], neural network collocation method (NNCM)[83, 84], deep Galerkin method (DGM)[85, 86], deep Ritz method (DRM)[91, 92], differential equation generative adversarial networks (DEGAN)[95, 96], fractional orthogonal neural networks[97, 98], least squares support vector machines (LS-SVM)[99, 100], etc. [87, 101, 102].

Each of these methods has its advantages and disadvantages; for instance, the spectral method has a spectral convergence for the smooth functions, but it is not a good choice for problems with a complex domain. On the other hand, the finite element method can handle domain complexity and singularity easily. Moreover,

scientists have combined these methods with each other to introduce new efficient numerical approaches; for example, spectral element approach [103], finite volume spectral element method [104], radial basis function-finite difference method (RBF-FD) [105], deep RBF collocation method [109].

In this chapter, we utilize a machine learning-based method using support vector machines called the least squares support vector machines collocation algorithm [110, 113] in conjunction with the Crank-Nicolson method to solve well-known second-order PDEs. In this algorithm, modified Chebyshev and Legendre orthogonal functions are considered as kernels. In the proposed method, first, the temporal dimension is discretized by the Crank-Nicolson approach. With the aid of the LS-SVM collocation method, the obtained equation at each time step is converted to an optimization problem in a primal form whose constraints are the discretized equation with its boundary conditions. Then, the KKT optimality condition is utilized to derive the dual form. Afterward, the problem is minimized in dual form using the Lagrange multipliers method [115]. Finally, the problem is converted to a linear system of algebraic equations that can be solved by a standard method such as QR or LU decomposition.

8.2 LS-SVM method for solving second-order partial differential equations

We consider a time-dependent second-order PDE with Dirichlet boundary condition in the following form:

$$\frac{\partial u}{\partial t} = A(x, t, u)u + B(x, t, u)\frac{\partial u}{\partial x} + C(x, t, u)\frac{\partial^2 u}{\partial x^2}, \quad (8.2)$$

$$x \in \Omega \subset \mathbb{R} \text{ and } t \in [0, T].$$

There are two strategies to solve this PDE by SVM. The first way is to use the LS-SVM approach for both time and space dimensions simultaneously as proposed by the authors in [110]. Another approach is to use a semi-discrete method, i.e. in order to solve Eq. 8.2, first, the time discretization method is recalled and then, LS-SVM is applied to solve the problem. Details of the proposed algorithm are described below.

8.2.1 Temporal discretization

The Crank-Nicolson method is chosen for time discretization due to its good convergence and unconditional stability. According to finite-difference methods, first, derivative formula by applying the Crank-Nicolson approach on Eq. 8.2, we have:

$$\begin{aligned} \frac{u^{i+1}(x) - u^i(x)}{\Delta t} &= \frac{1}{2} \left(A(x, t_i, u^i) u^{i+1}(x) + B(x, t_i, u^i) \frac{\partial u^{i+1}}{\partial t} + C(x, t_i, u^i) \frac{\partial^2 u^{i+1}}{\partial x^2} \right) \\ &+ \frac{1}{2} \left(A(x, t_i, u^i) u^i(x) + B(x, t_i, u^i) \frac{\partial u^i}{\partial t} + C(x, t_i, u^i) \frac{\partial^2 u^i}{\partial x^2} \right), \quad i = 0, \dots, m-1, \end{aligned} \quad (8.3)$$

where $u^i(x) = u(x, t_i)$ and $t_i = i\Delta t$. Moreover, $\Delta t = T/m$ is the size of time steps. This equation can be rewritten as:

$$\begin{aligned} u^{i+1}(x) - \frac{\Delta t}{2} \left(A(x, t_i, u^i) u^{i+1}(x) + B(x, t_i, u^i) \frac{\partial u^{i+1}}{\partial t} + C(x, t_i, u^i) \frac{\partial^2 u^{i+1}}{\partial x^2} \right) &= \\ u^i(x) + \frac{\Delta t}{2} \left(A(x, t_i, u^i) u^i(x) + B(x, t_i, u^i) \frac{\partial u^i}{\partial t} + C(x, t_i, u^i) \frac{\partial^2 u^i}{\partial x^2} \right). \end{aligned} \quad (8.4)$$

Now, the LS-SVM algorithm is applied on Eq. 8.4 to find the solution of the problem Eq. 8.2.

8.2.2 LS-SVM collocation method

At each time step j we have training data $\{x_i^j, u_i^j\}_{i=0}^n$, where $x_i^j \in \Omega$ and u_i^j are the input and output data, respectively. The solution of the problem in time step j is approximated as $\hat{u}^j(x) = \sum_{i=1}^n w_i \varphi(x) + b = \mathbf{w}^T \varphi(x) + b$, where $\{\varphi(x)\}$ are arbitrary basis functions. Consider

$$r^i(x) = u^i(x) + \frac{\Delta t}{2} \left(A(x, t_i, u^i) u^i(x) + B(x, t_i, u^i) \frac{\partial u^i}{\partial t} + C(x, t_i, u^i) \frac{\partial^2 u^i}{\partial x^2} \right).$$

The solution of $u^j(x)$ is obtained by solving the following optimization problem:

$$\text{minimize}_{\mathbf{w}, e} \quad \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \mathbf{e}^T \mathbf{e} \quad (8.5)$$

$$\text{s.t. } \tilde{A}_i \left[\mathbf{w}^T \varphi(x_i) + b \right] + \tilde{B}_i \mathbf{w}^T \varphi'(x_i) + \tilde{C}_i \mathbf{w}^T \varphi''(x_i) + r^j(x_i) + e_i \quad i = 0, \dots, n,$$

$$\begin{aligned} \mathbf{w}^T \varphi(x_0) &= p_1, \\ \mathbf{w}^T \varphi(x_n) &= p_2, \end{aligned}$$

where $\tilde{A}_i = 1 - \frac{\Delta t}{2} A_i$, $\tilde{B}_i = -\frac{\Delta t}{2} B_i$, and $\tilde{C}_i = -\frac{\Delta t}{2} C_i$. The collocation points $\{x_i\}_{i=0}^n$ are the Gauss-Lobatto Legendre or Chebyshev points [111, 112]. Now, the dual form representation of the problem is expressed. In addition, the Lagrangian of the problem Eq. 8.5 is as follows:

$$G = \frac{1}{2} \mathbf{w}^T \mathbf{w} + \frac{\gamma}{2} \mathbf{e}^T \mathbf{e} - \sum_{i=2}^n \alpha_i \left[\mathbf{w}^T (\tilde{A}_i \varphi(x_i) + \tilde{B}_i \varphi'(x_i) + \tilde{C}_i \varphi''(x_i)) - r^j(x_i) - e_i \right] - \beta_1 (\mathbf{w}^T \varphi(x_0) + b - p_1) - \beta_2 (\mathbf{w}^T \varphi(x_n) + b - p_2), \quad (8.6)$$

where $\{\alpha_i\}_{i=2}^n$, β_1 , and β_2 are Lagrange multipliers. Then, the KKT optimality conditions, for $l = 2, \dots, n$, are as follows:

$$\begin{aligned} \frac{\partial G}{\partial \mathbf{w}} = 0 \rightarrow \mathbf{w} &= \sum_{i=2}^{n-1} \alpha_i \left[\tilde{A}_i \varphi(x_i) + \tilde{B}_i \varphi'(x_i) + \tilde{C}_i \varphi''(x_i) \right] + \beta_1 \varphi(x_0) + \beta_2 \varphi(x_n), \\ \frac{\partial G}{\partial b} = 0 \rightarrow \sum_{i=2}^{n-1} \alpha_i \tilde{A}_i + \beta_1 + \beta_2 &= 0, \\ \frac{\partial G}{\partial e_l} = 0 \rightarrow e_l &= \frac{\alpha_l}{\gamma}, \\ \frac{\partial G}{\partial \alpha_l} = 0 \rightarrow \mathbf{w}^T (\tilde{A}_l \varphi(x_l) + \tilde{B}_l \varphi'(x_l) + \tilde{C}_l \varphi''(x_l)) + \tilde{A}_l b - e_l &= r^j(x_l), \\ \frac{\partial G}{\partial \beta_1} = 0 \rightarrow \mathbf{w}^T \varphi(x_0) + b &= p_1, \\ \frac{\partial G}{\partial \beta_2} = 0 \rightarrow \mathbf{w}^T \varphi(x_n) + b &= p_2. \end{aligned} \quad (8.7)$$

By substituting the first and third equations in the fourth one, the primal variables are eliminated. Now, since the multiplications of the basis functions appear in themselves or their derivatives, we need to define the derivatives of the kernel function $K(x_i, x_j) = \varphi(x_i)^T \varphi(x_j)$. According to Mercer's theorem, derivatives of the feature map can be written in terms of derivatives of the kernel function. So, we use differential operator $\nabla^{m,n}$ which is defined in the previous chapters 3 and 4. So, in time step k , it can be written that:

$$\begin{aligned} \sum_{j=2}^{n-1} \alpha_j \left[\tilde{A}_j \left(\chi_{j,i}^{(0,0)} \tilde{A}_j + \chi_{j,i}^{(1,0)} \tilde{B}_j + \chi_{j,i}^{(2,0)} \tilde{C}_j \right) + \tilde{B}_j \left(\chi_{j,i}^{(0,1)} \tilde{A}_j + \chi_{j,i}^{(1,1)} \tilde{B}_j + \chi_{j,i}^{(2,1)} \tilde{C}_j \right) + \right. \\ \left. \tilde{C}_j \left(\chi_{j,i}^{(0,2)} \tilde{A}_j + \chi_{j,i}^{(1,2)} \tilde{B}_j + \chi_{j,i}^{(2,2)} \tilde{C}_j \right) \right] + \beta_1 \left(\chi_{1,i}^{(0,0)} \tilde{A}_i + \chi_{1,i}^{(0,1)} \tilde{B}_i + \chi_{1,i}^{(0,2)} \tilde{C}_i \right) + \\ \beta_2 \left(\chi_{n,i}^{(0,0)} \tilde{A}_i + \chi_{1,i}^{(0,n)} \tilde{B}_i + \chi_{n,i}^{(0,2)} \tilde{C}_i \right) + \frac{\alpha_i}{\gamma} + \tilde{A}_i b = r_i^k, \quad i = 2, \dots, n. \end{aligned}$$

$$\sum_{j=2}^{n-1} \alpha_j \left[\chi_{j,1}^{(0,0)} \tilde{A}_j + \chi_{j,1}^{(1,0)} \tilde{B}_j + \chi_{j,1}^{(2,0)} \tilde{C}_j \right] + \chi_{1,1}^{(0,0)} \beta_1 + \chi_{n,1}^{(0,0)} \beta_2 + b = p_1,$$

$$\sum_{j=2}^{n-1} \alpha_j \left[\chi_{j,n}^{(0,0)} \tilde{A}_j + \chi_{j,n}^{(1,0)} \tilde{B}_j + \chi_{j,n}^{(2,0)} \tilde{C}_j \right] + \chi_{1,n}^{(0,0)} \beta_1 + \chi_{n,n}^{(0,0)} \beta_2 + b = p_2,$$

$$\sum_{j=2}^{n-1} \alpha_j \tilde{A}_j + \beta_1 + \beta_2 = 0. \quad (8.8)$$

According to the aforementioned equations, the following linear system is defined:

$$\left[\begin{array}{c|c|c|c} \chi_{1,1}^{(0,0)} & \mathcal{M}_1 & \chi_{1,n}^{(0,0)} & 1 \\ \hline \mathcal{P}_1 & \mathcal{K} + \frac{1}{\gamma} I & \mathcal{P}_n & \mathcal{A}^T \\ \hline \chi_{n,1}^{(0,0)} & \mathcal{M}_2 & \chi_{n,n}^{(0,0)} & 1 \\ \hline 1 & \mathcal{A} & 1 & 0 \end{array} \right] \begin{bmatrix} \beta_1 \\ \alpha \\ \beta_2 \\ b \end{bmatrix} = \begin{bmatrix} p_1 \\ \mathbf{r}^k \\ p_2 \\ 0 \end{bmatrix}, \quad (8.9)$$

where

$$\begin{aligned} \mathcal{A} &= [\tilde{A}_{2:n-1}], \quad \mathcal{B} = [\tilde{B}_{2:n-1}], \quad \mathcal{C} = [\tilde{C}_{2:n-1}], \\ \mathcal{M}_1 &= \chi_{1,2:n-1}^{(0,0)} \text{Diag}(\mathcal{A}) + \chi_{1,2:n-1}^{(1,0)} \text{Diag}(\mathcal{B}) + \chi_{1,2:n-1}^{(2,0)} \text{Diag}(\mathcal{C}), \\ \mathcal{P}_1 &= \text{Diag}(\mathcal{A}) \chi_{2:n-1,1}^{(0,0)} + \text{Diag}(\mathcal{B}) \chi_{2:n-1,1}^{(0,1)} + \text{Diag}(\mathcal{C}) \chi_{2:n-1,1}^{(0,2)}, \end{aligned}$$

$$\begin{aligned} \mathcal{K} &= \text{Diag}(\mathcal{A}) \left(\chi^{(0,0)} \text{Diag}(\mathcal{A}) + \chi^{(1,0)} \text{Diag}(\mathcal{B}) + \chi_{j,i}^{(2,0)} \text{Diag}(\mathcal{C}) \right) + \\ &\quad \text{Diag}(\mathcal{B}) \left(\chi^{(0,1)} \text{Diag}(\mathcal{A}) + \chi^{(1,1)} \text{Diag}(\mathcal{B}) + \chi^{(2,1)} \text{Diag}(\mathcal{C}) \right) + \\ &\quad \text{Diag}(\mathcal{C}) \left(\chi^{(0,2)} \text{Diag}(\mathcal{A}) + \chi^{(1,2)} \text{Diag}(\mathcal{B}) + \chi^{(2,2)} \text{Diag}(\mathcal{C}) \right), \end{aligned}$$

$$\begin{aligned} \mathcal{P}_n &= \text{Diag}(\mathcal{A}) \chi_{2:n-1,n}^{(0,0)} + \text{Diag}(\mathcal{B}) \chi_{2:n-1,n}^{(0,1)} + \text{Diag}(\mathcal{C}) \chi_{2:n-1,n}^{(0,2)}, \\ \mathcal{M}_2 &= \chi_{n,2:n-1}^{(0,0)} \text{Diag}(\mathcal{A}) + \chi_{n,2:n-1}^{(1,0)} \text{Diag}(\mathcal{B}) + \chi_{n,2:n-1}^{(2,0)} \text{Diag}(\mathcal{C}), \\ \mathbf{r}^k &= [r_i^k]_{2:n-1}^T. \end{aligned}$$

By solving system Eq. 8.9, $\{\alpha_i\}_{i=2}^{n-1}$, β_1 , β_2 , and b are calculated. Then, the solution in time step k , in the dual form can be obtained as follows:

$$\begin{aligned} \hat{u}^k(x) &= \sum_{i=2}^{n-1} \alpha_i \left(\tilde{A}(x_i) \chi^{(0,0)}(x_i, x) + \tilde{B}(x_i) \chi^{(1,0)}(x_i, x) + \tilde{C}(x_i) \chi^{(2,0)}(x_i, x) \right) \\ &\quad + \beta_1 \chi^{(0,0)}(x_0, x) + \beta_2 \chi^{(0,0)}(x_n, x) + b. \end{aligned}$$

Going through all the above steps, we can say that the solution of PDE Eq. 8.2 at each time step is approximated.

8.3 Numerical simulations

In this part, the proposed approach is applied to several examples. Fokker-Planck and generalized Fitzhugh-Nagumo equations are provided as test cases and each test case contains three different examples. These examples have exact solutions, so we can calculate the exact error to evaluate the precision of the proposed method. Moreover, we solve examples by different kernel functions and compare their obtained solutions to each other.

Modified Chebyshev/Legendre kernels of orders three and six are used in the following examples and defined as follows [116]

$$K(x, z) = \frac{\sum_{i=0}^n \mathcal{L}_i(x) \mathcal{L}_i^T(z)}{\exp(\delta \|x - z\|^2)}, \quad (8.10)$$

where $\{\mathcal{L}_i(\cdot)\}$ are Chebyshev or Legendre polynomials, n is the order of polynomials that in our examples, it is 3 or 6. In addition, δ is the decaying parameter.

In order to illustrate the accuracy of the proposed method, L_2 and root-mean-square (RMS) errors are computed as follows:

$$\begin{aligned} RMS &= \sqrt{\frac{\sum_{i=1}^n (\hat{u}(x_i) - u(x_i))^2}{n}}, \\ L_2 &= \sqrt{\sum_{i=1}^n |\hat{u}(x_i) - u(x_i)|^2}. \end{aligned}$$

8.3.1 Fokker-Planck equation

Fokker-Planck equations have various applications in astrophysics[117], biology[118], chemical physics[119], polymer [120], circuit theory[121], dielectric relaxation [124], economics [125], electron relaxation in gases [126], nucleation [127], optical bistability [128], quantum optics [129], reactive systems [130], solid-state physics [121], finance [122], cognitive psychology [123], etc [145, 146]. In fact, this equation is derived from the description of the Brownian motion of particles [131, 132]. For instance, one of the equations describing the Brownian motion in potential is the Kramer equation which is a special case of the Fokker-Planck equation [133, 134]. The general form of the Fokker-Planck equation is:

$$\frac{\partial u}{\partial t} = \left[-\frac{\partial}{\partial x} \psi_1(x, t, u) + \frac{\partial^2}{\partial x^2} \psi_2(x, t, u) \right] u, \quad (8.11)$$

in which ψ_1 and ψ_2 are the drift and diffusion coefficients, respectively. If these coefficients depend on x and t , the PDE is called forward Kolmogorov equation[135, 145]. There is another type of Fokker-Planck equation similar to the forward Kolmogorov

equation called backward Kolmogorov equation [145, 136] that is in the form [147]:

$$\frac{\partial u}{\partial t} = \left[-\psi_1(x, t) \frac{\partial}{\partial x} + \psi_2(x, t) \frac{\partial^2}{\partial x^2} \right] u. \quad (8.12)$$

Moreover, if ψ_1 and ψ_2 are dependent on u in addition to time and space, then we have the nonlinear Fokker-Planck equation which has important applications in biophysics[137], neuroscience[5, 6], engineering[146], laser physics[139], nonlinear hydrodynamics[140], plasma physics[141], pattern formation[142], and so on[143, 144].

There are some fruitful studies that explored the Fokker-Planck equation by classical numerical methods. For example, Vanaja [148] presented an iterative algorithm for solving this model. In [149], researchers employed the finite difference approach to the two-dimensional Fokker-Planck equation. In 2006, Dehghan and Tatari [150] developed He's variational iteration method (VIM) to approximate the solution of this equation. Moreover, Tatari et al. [151] investigated the application of the Adomian decomposition method for solving different types of Fokker-Planck equations. One year later, Lakestani and Dehghan [152] obtained the numerical solution of the Fokker-Planck equation using cubic B-spline scaling functions. In addition, Kazem et al. [146] proposed a meshfree approach to solve linear and nonlinear Fokker-Planck equations. Recently, a pseudo-spectral method was applied to approximate the solution of the Fokker-Planck equation with high accuracy [147].

By simplifying Eq. 8.11, the Fokker-Planck equation takes the form of Eq. 8.2. It is worth mentioning that if our equation is linear, then $A(x, t, u) = 0$. In the following, we present a number of various linear and nonlinear examples. These examples are defined over $\Omega = [0, 1]$.

It should be noted that the average CPU times for all examples of the Fokker-Planck equation and the Generalized FHN equation are about 2.48 and 2.5 seconds depending on the number of spacial nodes and the time steps. In addition, the order of time complexity for the proposed method is $O(Mn^3)$, in which M is the number of time steps and n is the number of spacial nodes.

8.3.1.1 Example 1

Consider Eq. 8.11 with $\psi_1(x) = -1$ and $\psi_2(x) = 1$, and initial condition $f(x) = x$. Using these fixed values we can write $A(x, t, u) = 0$, $B(x, t, u) = 1$, and $C(x, t, u) = 1$. The exact solution of this test problem is $u(x, t) = x + t$ [146, 152].

In this example, we consider $n = 15$, $\Delta t = 0.001$, $\gamma = 10^{15}$. Fig. 8.1 shows the obtained result of function $u(x, t)$ by third-order Chebyshev kernel as an example. Also, the L_2 and RMS errors of different kernels at various times are represented in Tables 8.1 and 8.2. Note that in these tables, the number next to the polynomial name indicates its order. Moreover, the value of the decaying parameter for each kernel is specified in the tables.

Table 8.1: Numerical absolute errors (L_2) of the method for Example 1 with different kernels

| t | Chebyshev-3 ($\delta = 3$) | Chebyshev-6 ($\delta = 1.5$) | Legendre-3 ($\delta = 2$) | Legendre-6 ($\delta = 1$) |
|------|---------------------------------|-----------------------------------|--------------------------------|--------------------------------|
| 0.01 | 2.4178e-04 | 0.0010 | 1.4747e-04 | 4.0514e-04 |
| 0.25 | 3.8678e-04 | 0.0018 | 2.5679e-04 | 8.1972e-04 |
| 0.5 | 3.8821e-04 | 0.0018 | 2.5794e-04 | 8.2382e-04 |
| 0.7 | 3.8822e-04 | 0.0018 | 2.5795e-04 | 8.2385e-04 |
| 1 | 3.8822e-04 | 0.0018 | 2.5795e-04 | 8.2385e-04 |

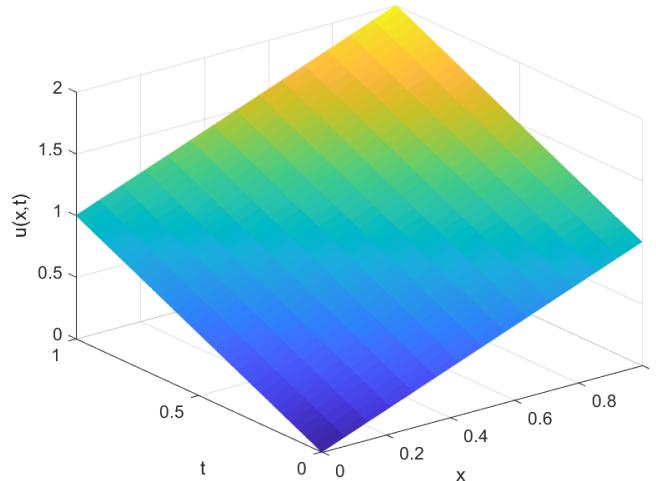


Fig. 8.1: Approximated solution of Example 1 by the third order Chebyshev kernel and $n = 10$, $\Delta t = 10^{-4}$

8.3.1.2 Example 2

Consider the backward Kolmogorov equation (8.12) with $\psi_1(x) = -(x+1)$, $\psi_2(x, t) = x^2 \exp(t)$, and initial condition $f(x) = x + 1$. This equation has the exact solution $u(x, t) = (x+1) \exp(t)$ [146, 152]. In this example, we have $A(x, t, u) = 0$, $B(x, t, u) = (x+1)$, and $C(x, t, u) = x^2 \exp(t)$. The parameters are set as $n = 10$, $\Delta t = 0.0001$, and $\gamma = 10^{15}$.

Table 8.2: RMS errors of the method for Example 1 with different kernels

| t | Chebyshev-3 ($\delta = 3$) | Chebyshev-6 ($\delta = 1.5$) | Legendre-3 ($\delta = 3$) | Legendre-6 ($\delta = 1$) |
|------|---------------------------------|-----------------------------------|--------------------------------|--------------------------------|
| 0.01 | 6.2427e-05 | 2.6776e-04 | 3.8077e-05 | 1.0461e-04 |
| 0.25 | 9.9866e-05 | 0.0013 | 6.6303e-05 | 2.1165e-04 |
| 0.5 | 1.0024e-04 | 0.0013 | 6.6600e-05 | 2.1271e-04 |
| 0.7 | 1.0024e-04 | 0.0013 | 6.6602e-05 | 2.1272e-04 |
| 1 | 1.0024e-04 | 0.0013 | 6.6602e-05 | 2.1272e-04 |

The obtained solution by the proposed method (with third-order Legendre kernel) is demonstrated in Fig. 8.2. Additionally, Tables 8.3 and 8.4 depict the numerical absolute errors and RMS errors of the presented method with different kernels. It can be deduced that the Legendre kernel is generally a better option for this example.

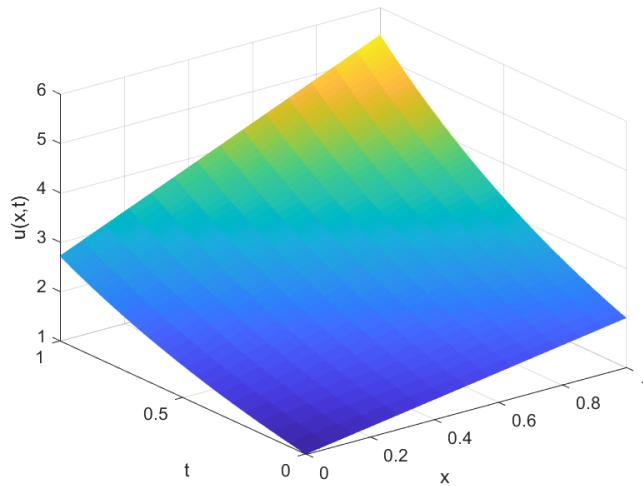


Fig. 8.2: Approximated solution of Example 2 by the third order Legendre kernel and $n = 15$, $\Delta t = 0.0001$

8.3.1.3 Example 3

Consider the nonlinear Fokker-Planck equation with $\psi_1(x, t, u) = \frac{7}{2}u$, $\psi_2(x, t, u) = xu$, and initial condition $f(x) = x$. By reconsidering this equation, it can be concluded

Table 8.3: Numerical absolute errors (L_2) of the method for Example 2 with different kernels

| t | Chebyshev-3 ($\delta = 3$) | Chebyshev-6 ($\delta = 1.5$) | Legendre-3 ($\delta = 3$) | Legendre-6 ($\delta = 1.5$) |
|------|---------------------------------|-----------------------------------|--------------------------------|----------------------------------|
| 0.01 | 3.6904e-04 | 0.0018 | 2.0748e-04 | 7.1382e-04 |
| 0.25 | 6.0740e-04 | 0.0043 | 2.7289e-04 | 0.0020 |
| 0.5 | 8.6224e-04 | 0.0064 | 3.6194e-04 | 0.0031 |
| 0.7 | 0.0011 | 0.0083 | 4.5389e-04 | 0.0040 |
| 1 | 0.0016 | 0.0116 | 6.3598e-04 | 0.0055 |

Table 8.4: RMS errors of the method for Example 2 with different kernels

| t | Chebyshev-3 ($\delta = 3$) | Chebyshev-6 ($\delta = 1.5$) | Legendre-3 ($\delta = 3$) | Legendre-6 ($\delta = 1$) |
|------|---------------------------------|-----------------------------------|--------------------------------|--------------------------------|
| 0.01 | 9.5285e-05 | 4.7187e-04 | 5.3572e-05 | 1.8431e-04 |
| 0.25 | 1.5683e-04 | 0.0011 | 7.0461e-05 | 5.1859e-04 |
| 0.5 | 2.2263e-04 | 0.0017 | 9.3453e-05 | 7.8761e-04 |
| 0.7 | 2.8540e-04 | 0.0021 | 1.1719e-04 | 0.0010 |
| 1 | 4.0064e-04 | 0.0030 | 1.6421e-04 | 0.0014 |

that $A(x, t, u) = 0$, $B(x, t, u) = -3u + 2xu_x$, and $C(x, t, u) = 2xu$. The exact solution of this problem is $u(x, t) = \frac{x}{t+1}$ [146, 152]. In order to overcome the nonlinearity of the problem, we use the value of u and its derivatives in the previous steps. For this example we set $n = 20$, $\Delta t = 0.0001$, and $\gamma = 10^{12}$.

The approximated solution of this example is displayed in Fig. 8.3. Also, the numerical errors for Example 3 are demonstrated in Tables 8.5 and 8.6.

Table 8.5: Numerical absolute errors (L_2) of the method for Example 3 with different kernels

| t | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|------------------|------------------|------------------|------------------|
| | ($\delta = 4$) | ($\delta = 3$) | ($\delta = 4$) | ($\delta = 4$) |
| 0.01 | 3.0002e-04 | 0.0010 | 2.1898e-04 | 5.0419e-04 |
| 0.25 | 3.0314e-04 | 0.0011 | 2.1239e-04 | 4.5307e-04 |
| 0.5 | 2.1730e-04 | 7.8259e-04 | 1.4755e-04 | 3.3263e-04 |
| 0.7 | 1.7019e-04 | 6.2286e-04 | 1.1327e-04 | 2.6523e-04 |
| 1 | 1.2321e-04 | 4.5854e-04 | 8.0082e-05 | 1.9611e-04 |

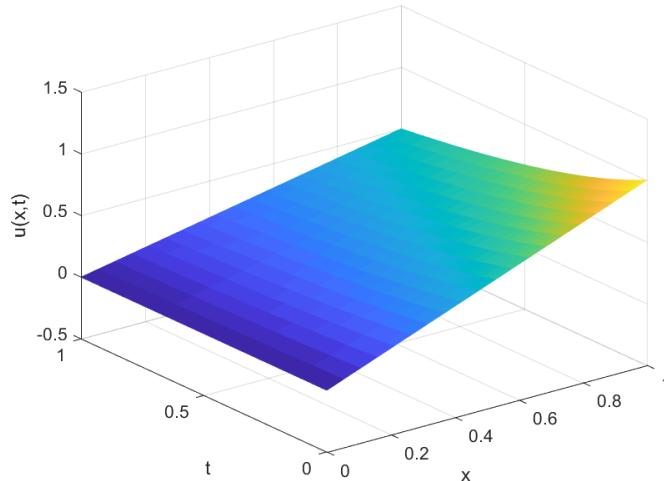


Fig. 8.3: Approximated solution of Example 3 by the sixth order Legendre kernel and $n = 20$, $\Delta t = 0.0001$

8.3.2 Generalized Fitzhugh-Nagumo equation

The Fitzhugh-Nagumo (FHN) model is based on electrical transmission in a nerve cell at the axon surface [169, 170]. In other words, this model is a simplified equation of the famous Hodgkin-Huxley (HH) model [171], because it uses a simpler structure to interpret the electrical transmission at the surface of the neuron as opposed to the complex HH design[74, 73, 179, 45]. In electrical analysis, the HH model is

Table 8.6: RMS errors of the method for Example 3 with different kernels

| t | Chebyshev-3 ($\delta = 4$) | Chebyshev-6 ($\delta = 1.5$) | Legendre-3 ($\delta = 4$) | Legendre-6 ($\delta = 4$) |
|------|---------------------------------|-----------------------------------|--------------------------------|--------------------------------|
| 0.01 | 6.7087e-05 | 2.2583e-04 | 4.8965e-05 | 1.1274e-04 |
| 0.25 | 6.7784e-05 | 2.3724e-04 | 4.7492e-05 | 1.0131e-04 |
| 0.5 | 4.8591e-05 | 1.7499e-04 | 3.2994e-05 | 7.4379e-05 |
| 0.7 | 3.8055e-05 | 1.3927e-04 | 2.5328e-05 | 5.9307e-05 |
| 1 | 2.7551e-05 | 1.0253e-04 | 1.7907e-05 | 4.3852e-05 |

simulated using a combination of a capacitor, resistor, rheostat, and current source elements[74, 73, 45, 179], while the FHN model uses a combination of resistor, inductor, capacitor, and diode elements[74, 73, 45, 179]. Also, the HH model involves three channels of sodium, potassium, and leak in the generation of electrical signals and is simulated by a rheostat and battery, while the FHN model simulates behavior by only one diode and inductor [74, 73, 45, 179]. For this reason, this model has attracted the attention of many researchers and it has been used in various other fields of flame propagation[153], logistic population growth[154], neurophysiology[155], branching Brownian motion process[156], autocatalytic chemical reaction[157], and nuclear reactor theory[158], etc. [159, 160, 161, 162, 163, 164, 165]. The classical Fitzhugh–Nagumo equation is given by [168, 159]

$$\frac{\partial u}{\partial t} = \frac{\partial^2 u}{\partial x^2} - u(1-u)(\rho - u), \quad (8.13)$$

where $0 \leq \rho \leq 1$, and $u(x, t)$ is the unknown function depending on the temporal variable t and the spatial variable x . It should be noted that this equation combines diffusion and nonlinearity which is controlled by the term $u(1-u)(\rho - u)$. The generalized FHN equation can be represented as follows [158]

$$\frac{\partial u}{\partial t} = -v(t) \frac{\partial u}{\partial x} + \mu(t) \frac{\partial^2 u}{\partial x^2} + \eta(t)u(1-u)(\rho - u). \quad (8.14)$$

It seems clear that in this model, when we assume that $v(t) = 0$, $\mu(t) = 1$, and $\eta(t) = -1$, we can conclude that model in relation Eq. 8.13.

Different types of FHN equations have been studied numerically by several researchers, as shown in Table 8.7

Table 8.7: Numerical methods used to solve different types of FHN models

| Authors | Method | Type of FHN | Year |
|-----------------------------|-----------------------------------|------------------------|------|
| Li and Guo [172] | First integral method | 1D-FHN | 2006 |
| Abbasbandy [159] | Homotopy analyses method | 1D-FHN | 2008 |
| Olmos and Shizgal [173] | Pseudospectral method | 1D- and 2D-FHN systems | 2009 |
| Hariharan et al. [174] | Haar wavelet method | 1D-FHN | 2010 |
| Van Gorder et al. [175] | Variational formulation | Nagumo-Telegraph | 2010 |
| Dehghan et al. [11] | Homotopy perturbation method | 1D-FHN | 2010 |
| Bhrawy [158] | Jacobi-Gauss-Lobatto collocation | Generalized FHN | 2013 |
| Jiwari et al. [165] | Polynomial quadrature method | Generalized FHN | 2014 |
| Moghaderi and Dehghan [177] | two-grid finite difference method | 1D- and 2D-FHN systems | 2016 |
| Kumar [178] et al. | q-homotopy analyse method | 1D fractional FHN | 2018 |
| Hemami et al. [74] | CS-RBF method | 1D- and 2D-FHN systems | 2019 |
| Hemami et al. [73] | RBF-FD method | 2D-FHN systems | 2020 |
| Moayeri et al. [179] | Generalized Lagrange method | 1D- and 2D-FHN systems | 2020 |
| Moayeri et al. [45] | Legendre spectral element | 1D- and 2D-FHN systems | 2020 |
| Abdel-Aty et al. [180] | Improved B-spline method | 1D fractional FHN | 2020 |

8.3.2.1 Example 1

Consider non-classical FHN model Eq. 8.14 with $v(t) = 0$, $\mu(t) = 1$, $\eta(t) = -1$, $\rho = 2$, initial condition $u(x, 0) = \frac{1}{2} + \frac{1}{2} \tanh(\frac{x}{2\sqrt{2}})$, and domain over $(x, t) \in [-10, 10] \times [0, 1]$. In this example, we have $A(x, t, u) = -\rho + (1 - \rho)u - u^2$, $B(x, t, u) = 0$ and $C(x, t, u) = -1$. So the exact solution of this test problem is $u(x, t) = \frac{1}{2} + \frac{1}{2} \tanh(\frac{x - \frac{2\rho-1}{2}t}{2\sqrt{2}})$ [158, 165, 166].

In this example, we set $n = 40$, $\Delta t = 2.5e10^{-4}$, $\gamma = 10^{15}$. Fig. 8.4 shows the obtained result of function $u(x, t)$ by sixth order Legendre kernel as an example. Also, the L_2 and RMS errors of different kernels at various times are represented in Tables 8.8 and 8.9.

Table 8.8: Numerical absolute errors (L_2) of the method for Example 1 with different kernels

| t | δ | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|----------|-------------|-------------|------------|------------|
| 0.01 | 30 | 9.8330e-07 | 9.6655e-07 | 9.8167e-07 | 9.7193e-07 |
| 0.25 | 30 | 2.4919e-05 | 2.4803e-05 | 2.5029e-05 | 2.4956e-05 |
| 0.50 | 30 | 5.1011e-05 | 5.0812e-05 | 5.1289e-05 | 5.1132e-05 |
| 0.07 | 30 | 7.2704e-05 | 7.2357e-05 | 7.3108e-05 | 7.2815e-05 |
| 1.00 | 30 | 1.0648e-04 | 1.0576e-04 | 1.0706e-04 | 1.0644e-04 |
| 0.01 | 50 | 4.7375e-06 | 1.4557e-06 | 3.9728e-06 | 1.3407e-06 |
| 0.25 | 50 | 5.2206e-05 | 2.6680e-05 | 4.4305e-05 | 2.6237e-05 |
| 0.50 | 50 | 6.9759e-05 | 5.1818e-05 | 6.3688e-05 | 5.1808e-05 |
| 0.07 | 50 | 8.6685e-05 | 7.3064e-05 | 8.2108e-05 | 7.3286e-05 |
| 1.00 | 50 | 1.1599e-04 | 1.0624e-04 | 1.1299e-04 | 1.0675e-04 |

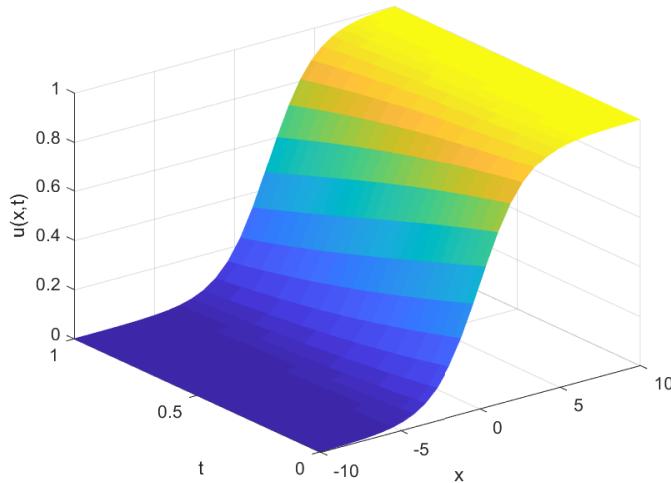


Fig. 8.4: Approximated solution of Example 1 by the sixth order Legendre kernel and $n = 40$, $\Delta t = 2.5 \times 10^{-4}$

8.3.2.2 Example 2

Consider the Fisher type of classical FHN equation with $v(t) = 0$, $\mu(t) = 1$, $\eta(t) = -1$, $\rho = \frac{1}{2}$, initial condition $u(x, 0) = \frac{3}{4} + \frac{1}{4} \tanh(\frac{\sqrt{2}}{8}x)$, and domain over $(x, t) \in$

Table 8.9: RMS errors of the method for Example 1 with different kernels

| t | δ | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|----------|-------------|-------------|------------|------------|
| 0.01 | 30 | 1.5547e-07 | 1.5283e-07 | 1.5522e-07 | 1.5368e-07 |
| 0.25 | 30 | 7.8801e-07 | 7.8433e-07 | 7.9149e-07 | 7.8917e-07 |
| 0.50 | 30 | 1.1406e-06 | 1.1362e-06 | 1.1469e-06 | 1.1433e-06 |
| 0.07 | 30 | 1.3740e-06 | 1.3674e-06 | 1.3816e-06 | 1.3761e-06 |
| 1.00 | 30 | 1.6836e-06 | 1.6723e-06 | 1.6927e-06 | 1.6829e-06 |
| 0.01 | 50 | 7.4906e-07 | 2.3017e-07 | 6.2816e-07 | 2.1198e-07 |
| 0.25 | 50 | 1.6509e-06 | 8.4368e-07 | 1.4010e-06 | 8.2970e-07 |
| 0.50 | 50 | 1.5598e-06 | 1.1587e-06 | 1.4241e-06 | 1.1585e-06 |
| 0.07 | 50 | 1.6382e-06 | 1.3808e-06 | 1.5517e-06 | 1.3850e-06 |
| 1.00 | 50 | 1.8340e-06 | 1.6798e-06 | 1.7865e-06 | 1.6878e-06 |

$[0, 1] \times [0, 1]$. So, we have $A(x, t, u) = -\rho + (1 - \rho)u - u^2$, $B(x, t, u) = 0$ and $C(x, t, u) = -1$. This equation has the exact solution $u(x, t) = \frac{1+\rho}{2} + (\frac{1}{2} - \frac{\rho}{2}) \tanh(\sqrt{2}(1-\rho)\frac{x}{4} + \frac{1-\rho^2}{4}t)$ [163, 165, 167].

In this example, we assume $n = 40$, $\Delta t = 2.5e10^{-4}$, and $\gamma = 10^{15}$. Fig. 8.5 shows the obtained result of function $u(x, t)$ by sixth-order Legendre kernel as an example. Also, the L_2 and RMS errors of different kernels at various times are represented in Tables 8.10 and 8.11.

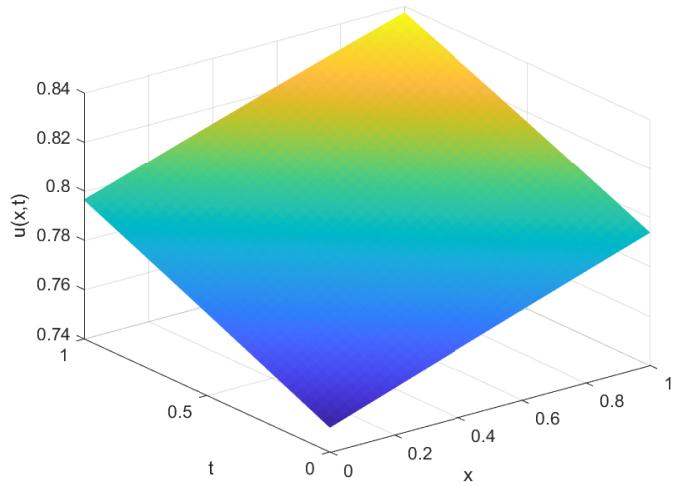


Fig. 8.5: Approximated solution of Example 2 by the sixth order Legendre kernel and $n = 40$, $\Delta t = 2.5e10^{-4}$

Table 8.10: Numerical absolute errors (L_2) of the method for Example 2 with different kernels

| t | δ | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|----------|-------------|-------------|------------|------------|
| 0.01 | 8e3 | 2.1195e-06 | 1.7618e-06 | 1.7015e-06 | 1.5138e-06 |
| 0.25 | 8e3 | 3.5893e-06 | 3.1130e-06 | 2.0973e-06 | 2.7182e-06 |
| 0.50 | 8e3 | 3.5383e-06 | 2.9330e-06 | 2.0790e-06 | 2.6138e-06 |
| 0.07 | 8e3 | 3.4609e-06 | 2.9069e-06 | 1.8524e-06 | 3.1523e-06 |
| 1.00 | 8e3 | 3.3578e-06 | 2.7996e-06 | 2.3738e-06 | 2.4715e-06 |
| 0.01 | 1e4 | 9.1689e-06 | 9.2317e-06 | 9.1726e-06 | 9.1931e-06 |
| 0.25 | 1e4 | 1.5310e-05 | 1.5417e-05 | 1.5318e-05 | 1.5352e-05 |
| 0.50 | 1e4 | 1.5089e-05 | 1.5195e-05 | 1.5097e-05 | 1.5130e-05 |
| 0.07 | 1e4 | 1.4869e-05 | 1.4974e-05 | 1.4876e-05 | 1.4909e-05 |
| 1.00 | 1e4 | 1.4472e-05 | 1.4575e-05 | 1.4479e-05 | 1.4512e-05 |

Table 8.11: RMS errors of the method for Example 2 with different kernels

| t | δ | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|----------|-------------|-------------|------------|------------|
| 0.01 | 8e3 | 3.3513e-07 | 2.7856e-07 | 2.6904e-07 | 2.3935e-07 |
| 0.25 | 8e3 | 5.6752e-07 | 4.9222e-07 | 3.3162e-07 | 4.2978e-07 |
| 0.50 | 8e3 | 5.5945e-07 | 4.6375e-07 | 3.2871e-07 | 4.1328e-07 |
| 0.07 | 8e3 | 5.4722e-07 | 4.5962e-07 | 2.9290e-07 | 4.9843e-07 |
| 1.00 | 8e3 | 5.3091e-07 | 4.4266e-07 | 3.7532e-07 | 3.9077e-07 |
| 0.01 | 1e4 | 1.4497e-06 | 1.4597e-06 | 1.4503e-06 | 1.4536e-06 |
| 0.25 | 1e4 | 2.4208e-06 | 2.4376e-06 | 2.4220e-06 | 2.4273e-06 |
| 0.50 | 1e4 | 2.3858e-06 | 2.4025e-06 | 2.3870e-06 | 2.3923e-06 |
| 0.07 | 1e4 | 2.3510e-06 | 2.3675e-06 | 2.3520e-06 | 2.3574e-06 |
| 1.00 | 1e4 | 2.2882e-06 | 2.3045e-06 | 2.2893e-06 | 2.2946e-06 |

8.3.2.3 Example 3

Consider the nonlinear time-dependent generalized FitzHugh-Nagumo equation with $v(t) = \cos(t)$, $\mu(t) = \cos(t)$, $\eta(t) = -2 \cos(t)$, $\rho = \frac{3}{4}$, initial condition $u(x, 0) = \frac{3}{8} + \frac{3}{8} \tanh(\frac{3x}{8})$, and domain over $(x, t) \in [-10, 10] \times [0, 1]$. In addition, we have $A(x, t, u) = -\rho + (1 - \rho)u - u^2$, $B(x, t, u) = 1$ and $C(x, t, u) = -1$. This equation has the exact solution $u(x, t) = \frac{\rho}{2} + \frac{\rho}{2} \tanh(\frac{\rho}{2}(x - (3 - \rho) \sin(t)))$ [158, 165, 168]. In this example, we set $n = 40$, $\Delta t = 2.5e10^{-4}$, and $\gamma = 10^{15}$. Fig. 8.6 shows the obtained result of function $u(x, t)$ by sixth order Legendre kernel as an example. Also, the L_2 and RMS errors of different kernels at various times are represented in Tables 8.12 and 8.13.

Table 8.12: Numerical absolute errors (L_2) of the method for Example 3 with different kernels

| t | δ | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|----------|-------------|-------------|------------|------------|
| 0.01 | 80 | 4.6645e-05 | 1.1548e-04 | 3.2880e-05 | 2.3228e-04 |
| 0.25 | 80 | 2.3342e-04 | 1.7331e-04 | 1.8781e-04 | 3.2000e-03 |
| 0.50 | 80 | 2.4288e-04 | 1.7697e-04 | 1.9577e-04 | 4.7000e-03 |
| 0.07 | 80 | 2.4653e-04 | 1.6815e-04 | 1.9935e-04 | 5.4000e-03 |
| 1.00 | 80 | 2.5033e-04 | 1.4136e-04 | 2.0333e-04 | 5.6000e-03 |
| 0.01 | 100 | 6.2498e-05 | 1.1966e-04 | 4.8828e-05 | 1.9291e-04 |
| 0.25 | 100 | 3.6141e-04 | 1.9420e-04 | 2.9325e-04 | 2.5000e-03 |
| 0.50 | 100 | 3.7364e-04 | 1.9940e-04 | 3.0328e-04 | 3.5000e-03 |
| 0.07 | 100 | 3.7937e-04 | 1.9260e-04 | 3.0863e-04 | 3.9000e-03 |
| 1.00 | 100 | 3.8635e-04 | 1.7084e-04 | 3.1527e-04 | 4.0000e-03 |

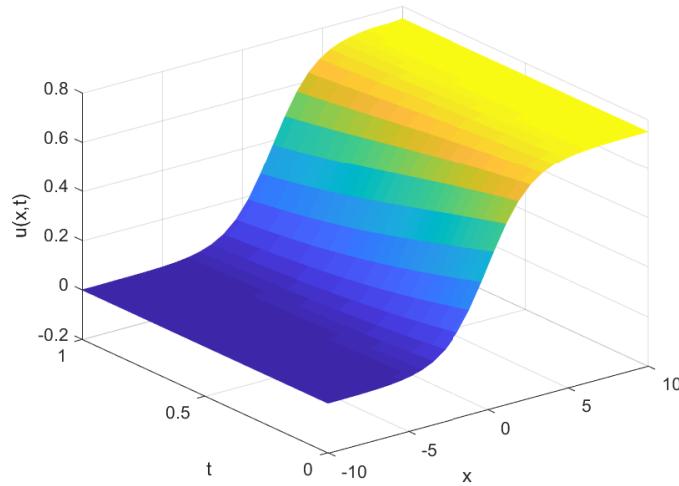
Fig. 8.6: Approximated solution of Example 3 by the sixth order Legendre kernel and $n = 40$, $\Delta t = 2.5 \times 10^{-4}$

Table 8.13: RMS errors of the method for Example 3 with different kernels

| t | δ | Chebyshev-3 | Chebyshev-6 | Legendre-3 | Legendre-6 |
|------|----------|-------------|-------------|------------|------------|
| 0.01 | 80 | 7.3752e-06 | 1.8259e-05 | 5.1988e-06 | 3.6726e-05 |
| 0.25 | 80 | 3.6907e-05 | 2.7403e-05 | 2.9695e-05 | 5.0071e-04 |
| 0.50 | 80 | 3.8403e-05 | 2.7981e-05 | 3.0954e-05 | 7.4712e-04 |
| 0.07 | 80 | 3.8980e-05 | 2.6586e-05 | 3.1521e-05 | 8.4603e-04 |
| 1.00 | 80 | 3.9580e-05 | 2.2351e-05 | 3.2149e-05 | 8.8481e-04 |
| 0.01 | 100 | 9.8818e-06 | 1.8921e-05 | 7.7204e-06 | 3.0502e-05 |
| 0.25 | 100 | 5.7144e-05 | 3.0706e-05 | 4.6367e-05 | 3.9744e-04 |
| 0.50 | 100 | 5.9077e-05 | 3.1528e-05 | 4.7954e-05 | 5.5947e-04 |
| 0.07 | 100 | 5.9983e-05 | 3.0452e-05 | 4.8798e-05 | 6.1710e-04 |
| 1.00 | 100 | 6.1087e-05 | 2.7012e-05 | 4.9849e-05 | 6.2629e-04 |

8.4 Conclusion

In this chapter, we introduced a machine learning-based numerical algorithm called the least squares support vector machine approach to solve partial differential equations. First, the temporal dimension was discretized by the Crank-Nicolson method. Then, the collocation LS-SVM approach was applied to the obtained equations at each time step. By employing the dual form, the problem was converted to a system of algebraic equations that can be solved by standard solvers. The modified Chebyshev and Legendre orthogonal kernel functions were used in LS-SVM formulations. In order to evaluate the effectiveness of the proposed method, it was applied to two well-known second-order partial differential equations, i.e. Fokker-Planck and generalized Fitzhugh-Nagumo equations. The obtained results from various orthogonal kernels were reported in terms of numerical absolute error and root-mean-square error. It can be concluded that the proposed approach has acceptable accuracy and is effective to solve linear and nonlinear partial differential equations.

References

1. Smith, G. D.: Numerical Solutions of Partial Differential Equations Finite Difference Methods. 3rd Edition, Oxford University Press, New York (1985)
2. Lindqvist, P.: Notes on the stationary p-Laplace equation. Springer International Publishing, Berlin (2019)
3. Kogut, P. I., Olha, P. Kupenko.: On optimal control problem for an ill-posed strongly nonlinear elliptic equation with p -Laplace operator and L^1 -type of nonlinearity. Discret Cont. Dyn-B **24**, 1273–1295 (2019)
4. Strikwerda, J. C.: Finite difference schemes and partial differential equations. Society for Industrial and Applied Mathematics, Pennsylvania (2004)

5. Hemami, M., Rad, J. A., Parand, K.: Phase distribution control of neural oscillator populations using local radial basis function meshfree technique with application in epileptic seizures: A numerical simulation approach. *Commun. Nonlinear SCI. Numer. Simul.* **103**, 105961 (2021)
6. Moayeri, M. M., Rad, J. A., Parand, K.: Desynchronization of stochastically synchronized neural populations through phase distribution control: a numerical simulation approach. *Nonlinear Dyn.*, **104**, 2363–2388 (2021)
7. Liu J, Hao Y.: Crank–Nicolson method for solving uncertain heat equation. *Soft Comput.*, **26**, 937–945 (2022)
8. Abazari R, Yildirim K.: Numerical study of Sivashinsky equation using a splitting scheme based on Crank-Nicolson method. *Math. Method. Appl. Sci.*, **16**, 5509–5521 (2019)
9. Trefethen, L. N.: Finite difference and spectral methods for ordinary and partial differential equations. Cornell University, New York (1996)
10. Meerschaert, M. M., Tadjeran, C.: Finite difference approximations for two-sided space-fractional partial differential equations. *Appl. Numer. Math.* **56**, 80–90 (2006)
11. Dehghan, M., Taleei, A.: A compact split-step finite difference method for solving the nonlinear Schrödinger equations with constant and variable coefficients. *Comput. Phys. Commun.* **181**, 80–90 (2010)
12. Bath, K. J., Wilson, E.: Numerical methods in finite element analysis. Prentice Hall, New Jersey (1976)
13. Ottosen, N., Petersson, H., Saabye, N.: Introduction to the Finite Element Method. Prentice Hall, New Jersey (1992)
14. Hughes, T.J.: The finite element method: linear static and dynamic finite element analysis. Courier Corporation, (2012)
15. Zienkiewicz, O.C., Taylor, R.L., Zhu, J.Z.: The finite element method: its basis and fundamentals. Elsevier, (2005)
16. Carstensen, C., Köhler, K.: Nonconforming FEM for the obstacle problem. *IMA J. Numer. Anal.* **37**, 64–93 (2017)
17. Wilson, P., Teschemacher, T., Bucher, P., Wüchner, R.: Non-conforming FEM-FEM coupling approaches and their application to dynamic structural analysis. *Eng. Struct.* **241**, 112342 (2021)
18. Bruggi, M., Venini, P.: NA mixed FEM approach to stress-constrained topology optimization. *Int. J. Numer. Methods Eng.* **73**, 1693–1714 (2008)
19. Bossavit, A., Véríté, J.C.: A mixed FEM-BIEM method to solve 3-D eddy-current problems. *IEEE Trans. Magn.* **18**, 431–435 (1982)
20. Kanschat G.: Multilevel methods for discontinuous Galerkin FEM on locally refined meshes. *Comput. struct.* **82**, 2437–2445 (2004)
21. Chien, C.C., Wu, T.Y.: A particular integral BEM/time-discontinuous FEM methodology for solving 2-D elastodynamic problems. *Int. J. Solids Struct.* **38**, 289–306 (2001)
22. LeVeque RJ.: Finite volume methods for hyperbolic problems. Cambridge university press, (2002)
23. Eymard, R., Gallouët, T., Herbin, R.: Finite volume methods. *Handbook of numerical analysis.* **7**, 713–1018 (2000)
24. Pozrikidis, C.: *Introduction to Finite and Spectral Element Methods Using MATLAB*. 2nd Edition, Oxford CRC Press (2014) .
25. Wang, C. H., Feng, Y. Y., Yue, K., Zhang, X. X.: Discontinuous finite element method for combined radiation-conduction heat transfer in participating media. *Int. Commun. Heat Mass.* **108**, 104287 (2019)
26. Yeganeh, S., Mokhtari, R., Hesthaven, J.S.: Space-dependent source determination in a time-fractional diffusion equation using a local discontinuous Galerkin method. *Bit Numer. Math.* **57**, 685–707 (2017)
27. Zhao, Y., Chen, P., Bu, W., Liu, X., Tang, Y.: Two Mixed Finite Element Methods for Time-Fractional Diffusion Equations. *J. Sci. Comput.* **70**, 407–428 (2017)
28. Zhao, D.H., Shen, H.W., Lai, J.S. III, G.T.: Approximate Riemann solvers in FVM for 2D hydraulic shock wave modeling. *J. Hydraulic Eng.* **122**, 692–702 (1996)

29. Kassab, A., Divo, E., Heidmann, J., Steinþorsson, E., Rodriguez, F.: BEM/FVM conjugate heat transfer analysis of a three-dimensional film cooled turbine blade. *Int. J. Numer. Methods heat fluid flow.* **13**, 581–610 (2003)
30. Fallah, N.A., Bailey, C., Cross, M., Taylor, G.A.: Comparison of finite element and finite volume methods application in geometrically nonlinear stress analysis. *Appl. Math. Model.* **24**, 439–455 (2000)
31. Ghidaglia JM, Kumbaro A, Le Coq G.: On the numerical solution to two fluid models via a cell centered finite volume method. *Eur. J. Mech. B Fluids.* **20**, 841–867 (2001)
32. Bertolazzi E, Manzini G.: A cell-centered second-order accurate finite volume method for convection-diffusion problems on unstructured meshes. *Math. Models Methods Appl. Sci.* **14**, 1235–1260 (2001)
33. Asouti, V.G., Trompoukis, X.S., Kampolis, I.C., Giannakoglou, K.C.: Unsteady CFD computations using vertex-centered finite volumes for unstructured grids on graphics processing units. *Int. J. Numer. Methods Fluids.* **67**, 232–246 (2011)
34. Zhang, Z., Zou, Q.: Some recent advances on vertex centered finite volume element methods for elliptic equations. *Sci. China Math.* **56**, 2507–2522 (2013)
35. Dubois F.: Finite volumes and mixed Petrov-Galerkin finite elements: The unidimensional problem. *Numer. Methods Partial Diff. Eq. Int. J.* **16**, 335–360 (2000)
36. Moosavi, M.R., Khelil, A.: Accuracy and computational efficiency of the finite volume method combined with the meshless local Petrov–Galerkin in comparison with the finite element method in elasto-static problem. *ICCES.* **5**, 211–238 (2008)
37. Zhao DH, Shen HW, Lai JS, III GT.: Approximate Riemann solvers in FVM for 2D hydraulic shock wave modeling. *J. Hydraulic Eng.* **122**, 692–702 (1996)
38. Liu, F., Zhuang, P., Turner, I., Burrage, K., Anh, V. : A new fractional finite volume method for solving the fractional diffusion equation. *Appl. Math. Model.* **38**, 3871–3878 (2014)
39. Fallah, N.: A cell vertex and cell centred finite volume method for plate bending analysis. *Comput. Methods Appl. Mech. Eng.* **193**, 3457–3470 (2004)
40. Spalart, P.R., Moser, R.D., Rogers, M.M.: Spectral methods for the Navier-Stokes equations with one infinite and two periodic directions. *J. Comput. Phys.* **96**, 297–324 (1991)
41. Mai-Duy N.: An effective spectral collocation method for the direct solution of high-order ODEs. *Commun. Numer. methods Eng.* **22**, 627–642 (2006)
42. Shen J.: Efficient spectral-Galerkin method I. Direct solvers of second-and fourth-order equations using Legendre polynomials. *SIAM J. Sci. Comput.* **15**, 1489–1505 (1994)
43. Moayeri, M. M., Rad, J. A., Parand, K.: Desynchronization of stochastically synchronized neural populations through phase distribution control: a numerical simulation approach. *Nonlinear Dyn.* **104**, 2363–2388 (2021)
44. Kopriva, D.: Implementing Spectral Methods for Partial Differential Equations, Springer, Berlin (2009)
45. Moayeri, M.M., Rad, J.A., Parand, K.: Dynamical behavior of reaction–diffusion neural networks and their synchronization arising in modeling epileptic seizure: A numerical simulation study. *Comput. Math. with Appl.* **80**, 1887–1927 (2020)
46. Delkhosh, M., Parand, K.: A new computational method based on fractional Lagrange functions to solve multi-term fractional differential equations. *Numer. Algor.* **88**, 729–766 (2021)
47. Latifi, S., Delkhosh, M.: Generalized Lagrange Jacobi-Gauss-Lobatto vs Jacobi-Gauss-Lobatto collocation approximations for solving (2 + 1)-dimensional Sine-Gordon equations. *Math. Methods Appl. Sci.* **43**, 2001–2019 (2020)
48. Fasshauer, G.E.: Meshfree approximation methods with MATLAB., World Scientific, Singapore (2007)
49. Liu, G.R.: Mesh Free Methods: Moving beyond the Finite Element Method, CRC press, Florida (2003)
50. Rad, J.A., Parand, K.: Numerical pricing of American options under two stochastic factor models with jumps using a meshless local Petrov–Galerkin method. *Appl. Numer. Math.* **115**, 252–274 (2017)
51. Dehghan, M., Shokri, A.: A numerical method for solution of the two-dimensional sine-Gordon equation using the radial basis functions. *Math. Comput. Simul.* **79**, 700–715 (2008)

52. Abbasbandy, S., Shirzadi, A.: A meshless method for two-dimensional diffusion equation with an integral condition. *Eng. Anal. Bound. Elem.* **34**, 1031–1037 (2010)
53. Abbasbandy, S., Shirzadi, A.: MLPG method for two-dimensional diffusion equation with Neumann's and non-classical boundary conditions. *Appl. Numer. Math.* **61**, 170–180 (2011)
54. Rad, J.A., Kazem, S., Parand K.: A numerical solution of the nonlinear controlled Duffing oscillator by radial basis functions. *Comput. Math. with Appl.* **64**, 2049–2065 (2012)
55. Parand, K., Hemami, M., Hashemi-Shahraki, S.: Two Meshfree Numerical Approaches for Solving High-Order Singular Emden–Fowler Type Equations. *Int. J. Appl. Comput. Math.* **3**, 521–546 (2017)
56. Rad, J. A., Kazem, S., Parand, K.: Optimal control of a parabolic distributed parameter system via radial basis functions. *Communications in Nonlinear Science and Numerical Simulation* **19**, 2559–2567 (2014)
57. Kazem, S., Rad, J. A.: Radial basis functions method for solving of a non-local boundary value problem with Neumann's boundary conditions. *Applied Mathematical Modelling* **36**, 2360–2369 (2012)
58. Kazem, S., Rad, J. A., Parand, K.: A meshless method on non-Fickian flows with mixing length growth in porous media based on radial basis functions: A comparative study. *Comput. Math. Appl.* **64**, 399–412 (2012)
59. Rashedi, K., Adibi, H., Rad, J. A., Parand, K.: Application of meshfree methods for solving the inverse one-dimensional Stefan problem. *Eng. Anal. Bound. Elem.* **40**, 1–21 (2014)
60. Parand, K., Rad, J. A.: Kansa method for the solution of a parabolic equation with an unknown spacewise-dependent coefficient subject to an extra measurement. *Comput. Phys. Commun.* **184**, 582–595 (2013)
61. Mohammadi, V., Dehghan, M., De Marchi, S.: Numerical simulation of a prostate tumor growth model by the RBF-FD scheme and a semi-implicit time discretization. *J. Comput. Appl. Math.* **388**, 113314 (2021)
62. Abbaszadeh, M., Dehghan, M.: Simulation flows with multiple phases and components via the radial basis functions-finite difference (RBF-FD) procedure: Shan-Chen model. *Eng. Anal. Bound. Elements* **119**, 151–161 (2020)
63. Abbaszadeh, M., Dehghan, M.: Direct meshless local Petrov–Galerkin method to investigate anisotropic potential and plane elastostatic equations of anisotropic functionally graded materials problems. *Eng. Anal. Bound. Elements* **118**, 188–201 (2020)
64. Rad, J. A., Parand, K., Abbasbandy, S.: Pricing European and American options using a very fast and accurate scheme: the meshless local Petrov–Galerkin method. *Proceedings of the National Academy of Sciences, India Section A: Physical Sciences* **85**, 337–351 (2015)
65. Rad, J. A., Parand, K.: Numerical pricing of American options under two stochastic factor models with jumps using a meshless local Petrov–Galerkin method. *Applied Numerical Mathematics* **115**, 252–274 (2017)
66. Rad, J. A., Parand, K.: Pricing American options under jump-diffusion models using local weak form meshless techniques. *International Journal of Computer Mathematics* **94**, 1694–1718 (2017)
67. Belytschko, T., Lu, Y.Y., Gu, L.: Element-free Galerkin methods. *Int. J. Numer. Methods Eng.* **37**, 229–256 (1994)
68. Dehghan, M., Narimani, N.: The element-free Galerkin method based on moving least squares and moving Kriging approximations for solving two-dimensional tumor-induced angiogenesis model. *Eng. Comput.* **36**, 1517–1537 (2020)
69. Liu, G.R., Zhang, G.Y., Gu, Y., Wang, Y.Y.: A meshfree radial point interpolation method (RPIM) for three-dimensional solids. *Comput. Mech.* **36**, 421–430 (2005)
70. Liu, G.R., Gu, Y.T.: A local radial point interpolation method (LRPIM) for free vibration analyses of 2-D solids. *J. Sound vibration* **246**, 29–46 (2001)
71. Rad, J. A., Ballestra, L. V.: Pricing European and American options by radial basis point interpolation. *Applied Mathematics and Computation* **251**, 363–377 (2015)
72. Rad, J. A., Parand, K., Abbasbandy, S.: Local weak form meshless techniques based on the radial point interpolation (RPI) method and local boundary integral equation (LBIE)

- method to evaluate European and American options. *Communications in Nonlinear Science and Numerical Simulation* **22**, 1178–1200 (2015)
73. Hemami, M., Rad, J.A., Parand, K.: The use of space-splitting RBF-FD technique to simulate the controlled synchronization of neural networks arising from brain activity modeling in epileptic seizures. *J. Comput. Sci.* **42**, 101090 (2020)
 74. Hemami, M., Parand, K., Rad, J.A.: Numerical simulation of reaction–diffusion neural dynamics models and their synchronization/desynchronization: Application to epileptic seizures. *Comput. Math. with Appl.* **78**, 3644–3677 (2019)
 75. Mohammadi, V., Dehghan, M.: A meshless technique based on generalized moving least squares combined with the second-order semi-implicit backward differential formula for numerically solving time-dependent phase field models on the spheres. *Appl. Numer. Math.* **153**, 248–275 (2020)
 76. Mohammadi, V., Dehghan, M.: Simulation of the phase field Cahn–Hilliard and tumor growth models via a numerical scheme: Element-free Galerkin method. *Comput. Methods Appl. Mech. Eng.* **345**, 919–950 (2019)
 77. Cortes, C., Vapnik, V.: Support-vector networks, *Machine learning*, **20**, 273–297 (1995)
 78. Jordan, M.I., Mitchell, T.M.: Machine learning: Trends, perspectives, and prospects. *Science*, **349**, 255–260 (2015)
 79. Aarts, L.P., Van Der Veer, P.: Neural network method for solving partial differential equations, *Neural Proc. Letters*, **14**, 261–271 (2001)
 80. Cheung, K.C., See, S.: Recent advance in machine learning for partial differential equation, *CCF Trans. High Perform. Comput.* **3**, 298–310 (2021)
 81. Raissi, M., Perdikaris, P., Karniadakis, G.E.: Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *J. Comput. Phys.* **378**, 686–797 (2019)
 82. Cai, S., Mao, Z., Wang, Z., Yin, M., Karniadakis, G.E.: Physics-informed neural networks (PINNs) for fluid mechanics: A review. *Acta Mechanica Sinica*. 1–12 (2022)
 83. Brink, A.R., Najera-Flores, D.A.: Martinez C. The neural network collocation method for solving partial differential equations. *Neural Comput. App.* **33**, 5591–5608 (2021)
 84. Liaqat, A., Fukuwara, M., Takeda, T.: Application of neural network collocation method to data assimilation. *Computer Phys. Commun.* **141**, 350–364 (2001)
 85. Sirignano, J., Spiliopoulos, K.: DGM: A deep learning algorithm for solving partial differential equations. *J. Comput. phys.* **375**, 1339–1364 (2018)
 86. Saporito, Y.F., Zhang, Z.: Path-Dependent Deep Galerkin Method: A Neural Network Approach to Solve Path-Dependent Partial Differential Equations. *SIAM J. Financ. Math.s.*, **12**, 912–40 (2021)
 87. Pang, G., Lu, L., Karniadakis, G.E.: fPINNs: Fractional Physics-Informed Neural Networks. *SIAM J. Sci. Comput.* **41**, A2603–A2626 (2019)
 88. Shizgal, B.: Spectral methods in chemistry and physics. *Scietif. Comput.* Springer (2015)
 89. Gamba, I. M., Rjasanow, S.: Galerkin–Petrov approach for the Boltzmann equation. *J. Comput. Phys.* **366**, 341–365 (2018)
 90. Chen, Y., Yi, N., Liu, W.: A Legendre–Galerkin spectral method for optimal control problems governed by elliptic equations. *SIAM J. Numer. Anal.* **46**, 2254–2275 (2008)
 91. E, W., Yu, B.:The Deep Ritz Method: A Deep Learning-Based Numerical Algorithm for Solving Variational Problems. *Commun. Math. Stat.* **6**, 1–12 (2018)
 92. Lu, Y., Lu, J., Wang, M.:The Deep Ritz Method: A priori generalization analysis of the deep Ritz method for solving high dimensional elliptic partial differential equations. Conference on Learning Theory, PMLR, 3196–3241 (2021)
 93. Yang, L., Zhang, D., Karniadakis, G.E.: Physics-informed generative adversarial networks for stochastic differential equations. *SIAM J. Scientif. Comput.* **46**, 292–317 (2020)
 94. Qin C., Wu, Y., Springenberg, J.T., Brock, A., Donahue, J., Lillicrap, T., Kohli, P.: Training generative adversarial networks by solving ordinary differential equations. *Advances in Neural Information Processing Systems*. **33**, 5599–5609 (2020)

95. Lee, Y.Y., Ruan, S.J., Chen, P.C.: Predictable Coupling Effect Model for Global Placement Using Generative Adversarial Networks with an Ordinary Differential Equation Solver. *IEEE Transactions on Circuits and Systems II: Express Briefs.* 1–5 (2021)
96. Kadeethumm T., O’Malley, D., Fuhg, J.N., Choi, Y., Lee, J., Viswanathan, H.S., Bouklas, N.: A framework for data-driven solution and parameter estimation of PDEs using conditional generative adversarial networks. *Nature Comput. Sci.* **1**, 819–829 (2021)
97. Hadian-Rasanan, A.H., Rahmati, D., Girgin, S., Parand, K.: A single layer fractional orthogonal neural network for solving various types of Lane–Emden equation. *New Astron.* **75**, 101307 (2019)
98. Hadian-Rasanan, A.H., Bajalan, Parand, K., Rad, J.A.: Simulation of nonlinear fractional dynamics arising in the modeling of cognitive decision making using a new fractional neural network. *Math. Methods Appl. Sci.* **43**, 1437–1466 (2020)
99. Shivanian, E., Hajimohammadi, Z., Baharifard, F., Parand, K., Kazemi, R.: A novel learning approach for different profile shapes of convecting-radiating fins based on shifted Gegenbauer LSSVM. *New Math. Natural Comput.* 1–27, (2022)
100. Hajimohammadi, Z., Shekarpaz, S., Parand, K. The novel learning solutions to nonlinear differential models on a semi-infinite domain. *Engineering with Computers*, 1–18 (2022)
101. Parand, K., Aghaei, A.A., Jani, M., Ghodsi, A.: Parallel LS-SVM for the numerical simulation of fractional Volterra’s population model. *Alexandria Eng. J.* **60**, 5637–5647 (2021)
102. Hajimohammadi, Z., Parand, K.: Numerical learning approximation of time-fractional sub diffusion model on a semi-infinite domain. *Chaos Solitons Frac.* **142**, 110435 (2021)
103. Karniadakis, G.E., Sherwin, S.J.: *Spectral/hp Element Methods for Computational Fluid Dynamics*, Oxford University press, New York (2005)
104. Shakeri, F., Dehghan, M.: A finite volume spectral element method for solving magnetohydrodynamic (MHD) equations. *Appl. Numer. Math.* **61**, 1–23 (2011)
105. Mohammadi, V., Dehghan, M., De Marchi, S.: Numerical simulation of a prostate tumor growth model by the RBF-FD scheme and a semi-implicit time discretization. *J. Comput. Appl. Math.* **388**, 113314 (2021)
106. Zayernouri, M., Karniadakis, G. E.: Fractional spectral collocation method. *SIAM J Sci Comput.* **36**, A40–A62 (2014)
107. Zayernouri, M., Karniadakis, G. E.: Fractional spectral collocation methods for linear and nonlinear variable order FPDEs. *J. Comput. Phys.* **293**, 312–338 (2015)
108. Zayernouri, M., Ainsworth, M., Karniadakis, G. E.: A unified Petrov–Galerkin spectral method for fractional PDEs. *Comput. Methods Appl. Mech. Eng.*, **283**, 1545–1569 (2015)
109. Saha, P., Mukhopadhyay, S.: A Deep Learning-based Collocation Method for Modeling Unknown PDEs from Sparse Observation. arxiv.org/pdf/2011.14965.pdf (2020)
110. Mehrkanoon, S., Suykens, J.A.K.: Learning solutions to partial differential equations using LS-SVM. *Neurocomputing*. **159**, 105–116 (2015)
111. Bhrawy, A.H., Baleanu, D.: A spectral Legendre–Gauss–Lobatto collocation method for a space-fractional advection diffusion equations with variable coefficients. *Reports Math. Phys.* **72**, 219–233 (2013)
112. Heydari, M. H., Avazzadeh, Z.: Chebyshev–Gauss–Lobatto collocation method for variable-order time fractional generalized Hirota–Satsuma coupled KdV system. *Eng. Comput.*, 1–10 (2020)
113. Mehrkanoon, S., Suykens, J.A.K.: Approximate Solutions to Ordinary Differential Equations Using Least Squares Support Vector Machines. *IEEE Trans. Neural Netw. Learn. Syst.* **23**, 1356–1362 (2012)
114. Vapnik, V.: *Statistical Learning Theory*. Wiley, New York. (1998)
115. Parand, K., Aghaei, A.A., Jani, M., Ghodsi, A.: A new approach to the numerical solution of Fredholm integral equations using least squares-support vector regression. *Math Comput. Simul.* **180**, 114–128 (2021)
116. Ozer, S., Chen, C.H., Cirpan, H.A.: A set of new Chebyshev kernel functions for support vector machine pattern classification. *Pattern Recognit.* **44**, 1435–1447 (2011)
117. Chavanis, P.H.: Nonlinear mean-field Fokker–Planck equations and their applications in physics, astrophysics and biology. *Comptes. Rendus. Physique.* **7**, 318–330 (2006)

118. Frank, T.D., Beek, P.J., Friedrich, R.: Fokker-Planck perspective on stochastic delay systems: Exact solutions and data analysis of biological systems. *astrophysics and biology. Phys. Review E.* **68**, 021912 (2003)
119. Grima, R., Thomas, P., Straube, A.V.: How accurate are the nonlinear chemical Fokker-Planck and chemical Langevin equations?. *astrophysics and biology. J. chemical phys.* **135**, 084103 (2011)
120. Chauvière, C., Lozinski, A.: Simulation of dilute polymer solutions using a Fokker-Planck equation. *Comput. fluids.* **33**, 687–696 (2004)
121. Kumar, S.: Numerical computation of time-fractional Fokker-Planck equation arising in solid state physics and circuit theory. *Z NATURFORSCH A.* **68**, 777–784 (2013)
122. Rad, J. A., Höök, J., Larsson, E., Sydow, L. V.: Forward deterministic pricing of options using Gaussian radial basis functions. *Journal of Computational Science* **24**, 209–217 (2018)
123. Hadian-Rasanan, A. H., Rad, J. A., Sewell, D. K.: Are there jumps in evidence accumulation, and what, if anything, do they reflect psychologically? An analysis of Lévy-Flights models of decision-making. *PsyArXiv* (2021) doi: 10.31234/osf.io/vy2mh
124. Tanimura, Y.: Stochastic Liouville, Langevin, Fokker-Planck, and master equation approaches to quantum dissipative systems. *J. Phys. Society Japan.* **75**, 082001 (2006)
125. Furioli, G., Pulvirenti, A., Terraneo, E., Toscani, G.: Fokker-Planck equations in the modeling of socio-economic phenomena. *Math. Models Methods Appl. Sci.* **27**, 115–158 (2017)
126. Braglia, G.L., Caraffini, G.L., Diligenti, M.: A study of the relaxation of electron velocity distributions in gases. *Il Nuovo Cimento B* **62**, 139–168 (1981)
127. Reguera, D., Rubí, J.M., Pérez-Madrid, A.: Fokker-Planck equations for nucleation processes revisited. *Physica A: Stat. Mech. Appl.* **259**, 10–23 (1998)
128. Gronchi, M., Lugiato A.: Fokker-Planck equation for optical bistability. *Lettore Al Nuovo Cimento.* **23**, 593–8 (1973)
129. D'ARIANO GM, Macchiavello C, Moroni S.: On the Monte Carlo Simulation Approach to Fokker-Planck Equations in Quantum Optics. *Modern Phys. Letters B.* **8**, 239–246 (1994)
130. De Decker, Y., Nicolis, G.: On the Fokker-Planck approach to the stochastic thermodynamics of reactive systems. *Physica A: Stat. Mech. Appl.* **553**, 124269 (2020)
131. Ullersma, P.: An exactly solvable model for Brownian motion: II. derivation of the Fokker-Planck equation and the master equation. *Physica.* **32**, 56–73 (1966)
132. Uhlenbeck, G.E., Ornstein, L.S.: A On the theory of the Brownian motion. *Phys. rev.* **36**, 823–841 (1930)
133. Jiménez-Aquino, J.I., Romero-Bastida, M.: Fokker-Planck-Kramers equation for a Brownian gas in a magnetic field. *Phys. Rev. E.* **74**, 041117 (2006)
134. Friedrich, R., Jenko, F., Baule, A., Eule, S.: Exact solution of a generalized Kramers-Fokker-Planck equation retaining retardation effects. *Phys. Rev. E.* **74**, 041103 (2006)
135. Conze, A., Lantos, N., Pironneau, O.: The forward Kolmogorov equation for two dimensional options. *Commun. Pure Appl. Anal.* **8**, 195 (2009)
136. Flandoli, F., Zanco, G.: An infinite-dimensional approach to path-dependent Kolmogorov equations. *Annals Probab.* **44**, 2643–2693 (2016)
137. Xing J, Wang H, Oster G.: From continuum Fokker-Planck models to discrete kinetic models. *Biophysical J.* **89**, 1551–1563 (2005)
138. Asghari, M., Hadian Rasanan, A.H., Gorgin, S., Rahmati, D., Parand, K.: FPGA-orthopoly: a hardware implementation of orthogonal polynomials. *Eng Comput* (inpress) (2022)
139. Blackmore, R., Weinert, U., Shizgal, B.: Discrete ordinate solution of a Fokker-Planck equation in laser physics. *Transport Theory Stat. Phy.* **15**, 181–210 (1986)
140. Zubarev, D.N., Morozov, V.G.: Statistical mechanics of nonlinear hydrodynamic fluctuations. *Physica A: Stat. Mech. Appl.* **120**, 411–467 (1983)
141. Peeters, A.G., Strintzi, D.: The Fokker-Planck equation, and its application in plasma physics. *Annalen der Physik.* **17**, 142–157 (2008)
142. Bengfort M, Malchow H, Hilker FM.: The Fokker-Planck law of diffusion and pattern formation in heterogeneous environments. *J. math. biology.* **73**, 683–704 (2016)
143. Barkai E.: Fractional Fokker-Planck equation, solution, and application. *Phys. Rev. E.* **63**, 046118 (2001)

144. Tsurui A, Ishikawa H.: Application of the Fokker-Planck equation to a stochastic fatigue crack growth model. *Structural Safety*. **63**, 15–29 (1986)
145. Risken, H.: *The Fokker-Planck Equation: Method of Solution and Applications*, Springer Verlag, Berlin (1989)
146. Kazem, S., Rad, J.A., Parand, K.: Radial basis functions methods for solving Fokker-Planck equation. *Eng. Anal. Bound. Elem.* **36**, 181–189 (2012)
147. Parand, K., Latifi, S., Moayeri, M.M., Delkhosh, M.: Generalized Lagrange Jacobi Gauss-Lobatto (GLJGL) Collocation Method for Solving Linear and Nonlinear Fokker-Planck Equations. *Eng. Anal. Bound. Elem.* **69**, 519–531 (2018)
148. Vanaja, V.: Numerical solution of a simple Fokker-Planck equation. *Appl. Numer. Math.* **9**, 533–540 (1992)
149. Zorzano, M.P., Mais, H., Vazquez, L.: Numerical solution of two dimensional Fokker-Planck equations. *Appl. Math. Comput.* **98**, 109–117 (1999)
150. Dehghan, M., Tatari, M.: Numerical solution of two dimensional Fokker-Planck equations. *Phys. Scr.* **74**, 310–316 (2006)
151. Tatari, M., Dehghan, M., Razzaghi, M.: Application of the Adomian decomposition method for the Fokker-Planck equation. *Phys. Scr.* **45**, 639–650 (2007)
152. Lakestani, M., Dehghan, M.: Numerical solution of Fokker-Planck equation using the cubic B-spline scaling functions. *Numer. Method. Part. D. E.* **25**, 418–429 (2008)
153. Van Gorder, R.A.: Gaussian waves in the Fitzhugh-Nagumo equation demonstrate one role of the auxiliary function $H(x, t)$ in the homotopy analysis method. *Commun. Nonlinear Sci. Numer. Simul.* **17**, 1233–1240 (2012)
154. Appadu, A.R., Agbavon, K.M.: Comparative study of some numerical methods for FitzHugh-Nagumo equation. *AIP Conference Proceedings*, AIP Publishing LLC, **2116**, 030036 (2019)
155. Aronson, D.G., Weinberger, H.F.: Nonlinear diffusion in population genetics, combustion, and nerve pulse propagation. *Partial differential equations and related topics*, Springer, Berlin, Heidelberg, 5–49 (1975)
156. Ali, H., Kamrujjaman, M., Islam, M.S.: Numerical computation of FitzHugh-Nagumo equation: A novel Galerkin finite element approach. *Int. J. Math. Research.* **9**, 20–27 (2020)
157. İnan B.: A finite difference method for solving generalized FitzHugh-Nagumo equation. *AIP Conference Proceedings*, AIP Publishing LLC, **1926**, 020018 (2018)
158. Bhrawy, A.H.: A Jacobi-Gauss-Lobatto collocation method for solving generalized Fitzhugh-Nagumo equation with time-dependent coefficients. *Appl. Math. Comput.* **222**, 255–264 (2013)
159. Abbasbandy, S.: Soliton solutions for the Fitzhugh-Nagumo equation with the homotopy analysis method. *Applied Mathematical Modelling*, **32**, 2706–2714 (2008)
160. Abdusalam, H.A.: Analytic and approximate solutions for Nagumo telegraph reaction diffusion equation. *Appl. Math. Comput.* **157**, 515–522 (2004)
161. Aronson, D.G., Weinberger, H.F.: Multidimensional nonlinear diffusion arising in population genetics. *Adv. Math.* **30**, 33–76 (1978)
162. Browne, P., Momoniat, E., Mahomed, F.M.: A generalized Fitzhugh-Nagumo equation. *Nonlinear Anal. Theory Methods Appl.* **68**, 1006–1015 (2008)
163. Kawahara, T., Tanaka, M.: Interactions of traveling fronts: an exact solution of a nonlinear diffusion equation. *Phys. Lett. A* **97**, 311–314 (1983)
164. Li, H., Guo, Y.: New exact solutions to the Fitzhugh-Nagumo equation. *Appl. Math. Comput.* **180**, 524–528 (2006)
165. Jiwari, R., Gupta, R. K., Kumar, V.: Polynomial differential quadrature method for numerical solutions of the generalized Fitzhugh-Nagumo equation with time-dependent coefficients. *Ain Shams Eng. J.* **5**, 1343–1350 (2014)
166. Wazwaz, A.M.: The tanh-coth method for solitons and kink solutions for nonlinear parabolic equations. *Appl. Math. Comput.* **188**, 1467–75 (2007)
167. Wazwaz, A.M.: Gorguis A. An analytic study of Fisher's equation by using adomian decomposition method. *Appl. Math. Comput.* **154**, 609–20 (2004)
168. Triki, H., Wazwaz, A.M.: On soliton solutions for the Fitzhugh-Nagumo equation with time-dependent coefficients. *Appl. Math. Model.* **37**, 3821–8 (2013)

169. FitzHugh R.: Impulses and physiological states in theoretical models of nerve membrane. *Biophysical J.* **1**, 445–466 (1961)
170. Gordon, A., Vugmeister, B.E., Dorfman, S., Rabitz, H.: Impulses and physiological states in theoretical models of nerve membrane. *Biophysical J.* **233**, 225–242 (1999)
171. Hodgkin, A.L., Huxley A.F.: Currents carried by sodium and potassium ions through the membrane of the giant axon of Loligo. *J. Physiology.* **116**, 449–72 (1952)
172. Li, H., Guo, Y.: New exact solutions to the FitzHugh-Nagumo equation. *Appl. Math. Comput.* **180**, 524–528 (2006)
173. Olmos, D., Shizgal B.D.: Pseudospectral method of solution of the Fitzhugh-Nagumo equation. *Math. Comput. Simul.* **79**, 2258–2278 (2009)
174. Hariharan, G., Kannan, K.: Haar wavelet method for solving FitzHugh-Nagumo equation. *Int. J. Math. Comput. Sci.* **4**, 909–913 (2010)
175. Van Gorder, R.A., Vajravelu K.: A variational formulation of the Nagumo reaction-diffusion equation and the Nagumo telegraph equation. *Nonlinear Anal.: Real World Appl.* **11**, 2957–2962 (2010)
176. Dehghan, M., Manafian Heris, J., Saadatmandi A.: Application of semi-analytic methods for the Fitzhugh-Nagumo equation, which models the transmission of nerve impulses. *Math. Methods Appl. Sci.* **33**, 1384–1398 (2010)
177. Moghaderi, H., Dehghan, M.: Mixed two-grid finite difference methods for solving one-dimensional and two-dimensional Fitzhugh-Nagumo equations. *Math. Methods Appl. Sci.* **40**, 1170–1200 (2016)
178. Kumar, D., Singh, J., Baleanu, D.: A new numerical algorithm for fractional Fitzhugh-Nagumo equation arising in transmission of nerve impulses. *Nonlinear Dyn.* **91**, 307–317 (2018)
179. Moayeri, M.M., Hadian-Rasanan, A. H., Latifi, S., Parand, K., Rad, J. A.: An efficient space-splitting method for simulating brain neurons by neuronal synchronization to control epileptic activity. *Eng. Comput.* 1–28 (2020)
180. Abdel-Aty, A. H., Khater, M., Baleanu, D., Khalil, E. M., Bouslimi, J., Omri, M.: Abundant distinct types of solutions for the nervous biological fractional FitzHugh-Nagumo equation via three different sorts of schemes. *Adv. Difference Eq.* **476**, 1–17 (2020)

Chapter 9

Solving Integral Equations by LS-SVR

Kourosh Parand and Alireza Afzal Aghaei and Mostafa Jani and Reza Sahleh

Abstract The other important type of problems in science and engineering are integral equations. Thus developing precise numerical algorithms for approximating the solution of these problems is one of the main questions of scientific computing. In this chapter, the least squares support vector algorithm is utilized for developing a numerical algorithm for solving various types of integral equations. The robustness also the convergence of the proposed method is discussed in this chapter by providing several numerical examples.

9.1 Introduction

Any equation with an unknown function under the integral sign is called an integral equation. These equations frequently appear in science and engineering, for instance, different mathematical models such as diffraction problems [1], scattering in quantum mechanics [2], plasticity [3], conformal mapping [4], water waves [5], and Volterra's population model [7] are expressed as integral equations [9, 10, 11, 16, 18, 20, 21]. Recently, the applications of integral equations in machine learning problems have

Kourosh Parand

Department of Statistics and Actuarial Science, University of Waterloo, Canada e-mail:
k_parand@sbu.ac.ir

Alireza Afzal Aghaei

Department of Computer and Data Science, Faculty of Mathematical Sciences, Shahid Beheshti
University, Tehran, Iran, e-mail: alirezaafzalaghaei@gmail.com

Mostafa Jani

Department of Computer and Data Science, Faculty of Mathematical Sciences, Shahid Beheshti
University, Tehran, Iran e-mail: mostafa.jani@gmail.com

Reza Sahleh

Department of Computer and Data Science, Faculty of Mathematical Sciences, Shahid Beheshti
University, Tehran, Iran e-mail: Navid.sahleh@gmail.com

also been discussed by researchers [22, 8, 23]. Furthermore, integral equations are closely related to differential equations, and in some cases, these equations can be converted to each other. Integro-differential equations are also a type of integral equations, in which not only the unknown function is placed under the integral operator, but also its derivatives appear in the equation. In some cases, the partial derivative of the unknown function may appear in the equation. In this case, the equation is called the partial integro-differential equation. Distributed fractional differential equations are also a type of fractional differential equations, which are very similar to integral equations. In these equations, the derivative fraction of the unknown function appears under the integral in such a way that the fractional derivative order is the same as the integral variable.

Due to the importance and wide applications of integral equations, many researchers have developed efficient methods for solving these types of equations [24, 25, 26, 27, 28, 29, 35, 36]. In this chapter, starting with the explanation of integral equations, and in the following, a new numerically efficient method based on the least-squares support vector regression approach is proposed for simulation of some integral equations.

9.2 Integral equations

Integral equations are divided into different categories based on their properties. However, there are three main types of integral equations [36], i.e. Fredholm, Volterra, and Volterra-Fredholm integral equations. While the Fredholm integral equations have constant integration bounds, in Volterra integral equations at least one of the bounds depends on the independent variable. Volterra-Fredholm integral equations also include both types of equations. These equations themselves are divided into subcategories such as linear/non-linear, homogeneous/inhomogeneous, first/second kind, etc. In the next section, different classes of these equations are presented and discussed.

9.2.1 Fredholm integral equations

Fredholm integral equations are an essential class of integral equations used in image processing [6] and reinforcement learning problems [22, 23]. Since analytical methods for finding the exact solution are only available in specific cases, various numerical techniques have been developed to approximate the exact solution of these types of equations as well as a larger class of these models such as Hammerstein, system of equations, multi-dimensional equations, and non-linear equations. Here, a brief overview of some numerical methods for these integral equations is given. The local radial basis function method was presented in [47] for solving Fredholm integral equations. Also, Bahmanpour et al. developed the Müntz wavelets method [55] to

simulate these models. On the other hand, Newton–Raphson, and Newton–Krylov methods were used for solving one and two-dimensional nonlinear Fredholm integral equations [64]. In another work, the least squares support vector regression method was proposed to solve Fredholm integral equations [66]. To see more, the interested reader can see [30, 31, 32, 33, 34].

For any $a, b, \lambda \in \mathbb{R}$, the following equation is called a Fredholm integral equation [36, 37, 38]

$$u(x) = f(x) + \lambda \int_a^b K(x, t)u(t)dt,$$

where $u(x)$ is the unknown function and $K(x, t)$ is the kernel of the equation.

9.2.2 Volterra integral equations

Volterra integral equations are another category of integral equations. This kind of equation appears in many scientific applications, such as population dynamics [40], the spread of epidemics [41], and semi-conductor devices [42]. Also, these equations can be obtained from initial value problems. Different numerical methods are proposed to solve these types of equations, for example, a collocation method using Sinc and rational Legendre functions proposed to solve Volterra's population model [67]. In another work, the least squares support vector regression method was proposed to solve Volterra integral equations [72]. Also, Runge–Kutta method was implemented to solve the second kind of linear Volterra integral equations [68]. To see more, the interested reader can see [44, 45, 46]. In this category, the equation is defined as follows [36, 37, 38]:

$$u(x) = f(x) + \lambda \int_{g(x)}^{h(x)} K(x, t)u(t)dt, \quad (9.1)$$

where $h(x)$ and $g(x)$ are known functions.

9.2.3 Volterra-Fredholm integral equations

Volterra-Fredholm integral equations are a combination of the Fredholm and Volterra integral equations. These equations are obtained from parabolic boundary value problems [36, 37, 38] and can also be derived from spatiotemporal epidemic modeling [65, 54]. Several numerical methods have been proposed for solving the Volterra-Fredholm integral model. For example, Brunner has developed a Spline collocation method for solving nonlinear Volterra-Fredholm integral equation appeared in spatio-temporal development of epidemic [65], Maleknejad and colleagues have utilized a collocation method based on orthogonal triangular functions for this kind of prob-

lems [49], and Legendre wavelet collocation method is applied to these problems by Yousefi and Razzaghi [50]. To see more, the interested reader can see [48, 51, 52, 53].

In a one-dimensional linear case, these equations can be written as follows [36, 37, 38]

$$u(x) = f(x) + \lambda_1 \int_a^b K_1(x, t)u(t)dt + \lambda_2 \int_{g(x)}^{h(x)} K_2(x, t)u(t)dt,$$

which include Fredholm and Volterra integral operators.

Remark 9.1 (First and second kind integral equations) It should be noted that if $u(x)$ only appears under the integral sign, it is called the first kind, otherwise, if the unknown function appears inside and outside the integral sign, it is named the second kind. For instance, the Eq. 9.1 is a second kind Volterra integral equation and the equation below $f(x) = \lambda \int_{g(x)}^{h(x)} K(x, t)u(t)dt$ is the first kind.

Remark 9.2 (Linear and nonlinear integral equations) Suppose we have the integral equation $\psi(u(x)) = f(x) + \lambda \int_{g(x)}^{h(x)} K(x, t)\phi(u(t))dt$. If either $\psi(x)$ or $\phi(x)$ be a non-linear function, the equation is called non-linear. In the case of the first kind Volterra integral equation, we have $f(x) = \lambda \int_{g(x)}^{h(x)} K(x, t)\phi(u(t))dt$.

Remark 9.3 (Homogeneous and inhomogeneous integral equations) In the field of integral equations, the function $f(x)$, which appeared in previous equations, is defined as the data function. This function plays an important role in determining the solution of the integral equation. It is known if the function $f(x)$ is on the function range of $K(x, t)$, then this equation has a solution. For instance, if the kernel is in the form of $K(x, t) = \sin(x) \sin(t)$, the function $f(x)$ should be a coefficient of $\sin(x)$ [27]. Otherwise, the equation has no solution [36]. Due to the importance of function $f(x)$ in integral equations, scientists have categorized equations based on the existence or non-existence of this function. If there is a function $f(x)$ in the integral equation, it is called inhomogeneous, otherwise, it is identified as homogeneous. In other words, the Fredholm integral equation of the second kind $u(x) = f(x) + \lambda \int_a^b K(x, t)u(t)dt$ is inhomogeneous and equation $u(x) = \lambda \int_a^b K(x, t)u(t)dt$ is homogeneous.

9.2.4 Integro-differential equations

As mentioned at the beginning of this chapter, integro-differential equations are integral equations in which the derivatives of the unknown function also appear in the equation. For instance, the equation

$$\begin{aligned} \frac{d^n u(x)}{dx^n} &= f(x) + \lambda \int_a^b K(x, t)u(t)dt, \\ \left. \frac{d^k u}{dx^k} \right|_{x_0} &= b_k \quad 0 \leq k \leq n-1, \end{aligned}$$

is the second kind Fredholm integro-differential equation, which n is a positive integer and b_k is the initial value for determining the unknown function. Several numerical methods have been proposed for solving the model discussed in here. For example, in 2006, several finite difference schemes including the forward Euler explicit, the backward Euler implicit, the Crank–Nicolson implicit, and Crandall’s implicit have been developed for solving partial integro-differential equations by Dehghan [56]. In another research, Dehghan and Saadatmandi have utilized Chebyshev finite difference algorithm for both linear and nonlinear Fredholm integro-differential equations. To see more, the interested reader can see [58, 59, 60, 61, 62, 63].

9.2.5 Multi-dimensional integral equations

Due to the existence of different variables, the study and modeling of physical problems usually lead to the creation of multi-dimensional cases [12]. Partial differential equations and multi-dimensional integral equations are the most famous examples of modeling these problems. The general form of these equations are defined as follows:

$$\mu u(\mathbf{x}) = f(\mathbf{x}) + \lambda \int_S K(\mathbf{x}, \mathbf{t})\sigma(u(\mathbf{t}))dt, \quad \mathbf{x}, \mathbf{t} \in S \subset \mathbb{R}^n, \quad (9.2)$$

where $\mathbf{x} = (x_1, x_2, \dots, x_n)$, $\mathbf{t} = (t_1, t_2, \dots, t_n)$, and λ is the eigenvalue of the integral equation. Also, for convenience in defining the first and second kind equations, the constant $\mu \in \mathbb{R}$ has been added to the left side of the equation. If this constant is zero, it is called the first kind equation, otherwise, it is the second kind. In the general case, this type of integral equation cannot be evaluated usually using exact closed-form solutions, and powerful computational algorithms are required. However, some works are available on numerical simulations of this model. For example, [73] moving least squares method for one and two-dimensional Fredholm integral equations, Legendre collocation method for Volterra–Fredholm integral equations [13], and Jacobi collocation method for multi-dimensional Volterra integral equations [14]. To see more, the interested reader can see [15, 17, 19].

9.2.6 System of integral equations

Another type of problem in integral equations is systems of equations. These systems appear in the form of several equations and several unknowns which their equations

are integral. The general form of these systems are defined as follows

$$u_i(x) = f_i(x) + \int_a^b \sum_{j=1}^n K_{ij}(x, t)v_{ij}(t)dt, \quad i = 1, \dots, n.$$

For example, the following is a system of two equations and two unknowns of the Fredholm integral equations

$$\begin{cases} u_1(x) = f_1(x) + \int_a^b (K_{11}(x, t)v_{11}(t) + K_{12}(x, t)v_{12}(t))dt, \\ u_2(x) = f_2(x) + \int_a^b (K_{21}(x, t)v_{21}(t) + K_{22}(x, t)v_{22}(t))dt. \end{cases}$$

Remark 9.4 (Relationship between differential equations and integral equations) Differential equations and integral equations are closely related, and some of them can be converted to another. Sometimes it is beneficial to convert a differential equation into an integral equation because this approach can prevent the instability of numerical solvers in differential equations [36, 37, 38]. Fredholm integral equations can be converted to differential equations with boundary values, and Volterra integral equations can be converted to differential equations with initial values [72].

9.3 LS-SVR for solving IEs

In this section, an efficient method for the numerical simulation of integral equations is proposed. Using the ideas behind weighted residual methods, the proposed technique introduces two different algorithms named collocation LS-SVR (CLS-SVR) and Galerkin LS-SVR (GLS-SVR). For the sake of simplicity, here we denote an integral equation in the operator form

$$\mathcal{N}(u) = f, \quad (9.3)$$

in which

$$\mathcal{N}(u) := \mu u - \mathcal{K}_1(u) - \mathcal{K}_2(u).$$

Here, $\mu \in \mathbb{R}$ is a constant which specifies the first or second kind the integral equation for $\mu = 0$ and $\mu \neq 0$, respectively. The operators \mathcal{K}_1 and \mathcal{K}_2 , are Fredholm and Volterra integral operators, respectively. These operators are defined as

$$\mathcal{K}_1(u) = \lambda_1 \int_{\Delta_1} K_1(\mathbf{x}, \mathbf{t})u(\mathbf{t})d\mathbf{t},$$

and

$$\mathcal{K}_2(u) = \lambda_2 \int_{\Delta_2} K_2(\mathbf{x}, \mathbf{t})u(\mathbf{t})d\mathbf{t}.$$

The proposed method can solve a wide range of integral equations, so we split the subject into three sections, based on the structure of the unknown function which should be approximated.

9.3.1 One-dimensional case

In order to approximate the solution of Eq. 9.3 using the LS-SVR formulations, some training data is needed. In contrast to the LS-SVR, in which there is a set of labeled training data, there are no labels for any arbitrary set of training data in solving Eq. 9.3. To handle this problem, the approximate solution is expanded as a linear combination of unknown coefficients, and some basis functions φ_i for $i = 1, \dots, d$

$$u(x) \simeq \tilde{u}(x) = w^T \varphi(x) + b = \sum_{i=1}^d w_i \varphi_i(x) + b. \quad (9.4)$$

In a general form, the LS-SVR primal problem can be formulated as:

$$\begin{aligned} \min_{w, e} \quad & \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e \\ \text{s.t.} \quad & \langle \mathcal{N}(\tilde{u}) - f, \psi_k \rangle = e_k, \quad k = 1, \dots, n, \end{aligned} \quad (9.5)$$

in which, n is the number of training points, $\{\psi_k\}_{k=1}^n$ is a set of test functions in the test space, and $\langle \cdot, \cdot \rangle$ is the inner product of two functions. In order to take advantage of the kernel trick, the dual form of this optimization problem is constructed. If the operator \mathcal{N} is linear, the optimization problem of Eq. 9.5 is convex, and the dual form can be derived easily. To do so, we denote the linear operators as \mathcal{L} and construct the Lagrangian function

$$\mathfrak{L}(w, e, \alpha) = \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e - \sum_{k=1}^n \alpha_k [\langle \mathcal{L}\tilde{u} - f, \psi_k \rangle - e_k], \quad (9.6)$$

in which $\alpha_k \in \mathbb{R}$ are Lagrangian multipliers. The conditions for the optimality of the Eq. 9.6 yield

$$\begin{cases} \frac{\partial \mathfrak{L}}{\partial w_k} = 0 \rightarrow w_k = \sum_{i=0}^n \alpha_i \langle \mathcal{L}\varphi_k, \psi_i \rangle, & k = 1, \dots, d. \\ \frac{\partial \mathfrak{L}}{\partial e_k} = 0 \rightarrow \gamma e_k + \alpha_k = 0, & k = 1, \dots, n. \\ \frac{\partial \mathfrak{L}}{\partial b} = 0 \rightarrow \sum_{i=0}^n \alpha_i \langle \mathcal{L}1, \psi_i \rangle = 0, \\ \frac{\partial \mathfrak{L}}{\partial \alpha_k} = 0 \rightarrow \sum_{i=1}^d w_i \langle \mathcal{L}\varphi_i - f, \psi_k \rangle + \langle \mathcal{L}0, \psi_k \rangle = e_k. & k = 1, \dots, n. \end{cases} \quad (9.7)$$

By eliminating w and e , the following linear system is obtained:

$$\left[\begin{array}{c|c} 0 & \tilde{\mathcal{L}}^T \\ \hline \tilde{\mathcal{L}} & \Omega + I/\gamma \end{array} \right] \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix}, \quad (9.8)$$

in which

$$\begin{aligned} \alpha &= [\alpha_1, \dots, \alpha_n]^T, \\ 1_n &= [1, \dots, 1]^T, \\ y &= [\langle f, \psi_1 \rangle, \langle f, \psi_2 \rangle, \dots, \langle f, \psi_n \rangle]^T, \\ \tilde{\mathcal{L}} &= \langle \mathcal{L}1, \psi_i \rangle. \end{aligned} \quad (9.9)$$

The kernel trick is also applied within the matrix Ω :

$$\begin{aligned} \Omega_{i,j} &= \langle \mathcal{L}\varphi, \psi_i \rangle^T \langle \mathcal{L}\varphi, \psi_j \rangle \\ &= \langle \mathcal{L}(\mathcal{L}K(x, t), \psi_i), \psi_j \rangle, \quad i, j = 1, 2, \dots, n, \end{aligned}$$

with any valid Mercer kernel $K(x, t)$. The approximate solution in the dual form takes the form

$$\tilde{u}(x) = \sum_{i=1}^n \alpha_i \tilde{K}(x, x_i) + b, \quad (9.10)$$

where

$$\tilde{K}(x, x_i) = \langle \langle \mathcal{L}\varphi, \psi_i \rangle, \varphi \rangle = \langle \mathcal{L}K(x, t), \psi_i \rangle.$$

9.3.2 Multi-dimensional case

One of the most used techniques to solve multi-dimensional integral equations, is to approximate the solution using a nested summation with a tensor of unknown coefficients and some basis functions. For instance, in order to solve a two-dimensional integral equation, we can use this function:

$$u(x, y) \simeq \tilde{u}(x, y) = \sum_{i=1}^d \sum_{j=1}^d w_{i,j} \varphi_i(x) \varphi_j(y) + b.$$

Note that the upper summation bound d and basis functions φ can vary in each dimension. Fortunately, there is no need to reconstruct the proposed model. In order to use LS-SVR for solving multi-dimensional equations, we can vectorize the unknown tensor w , basis functions φ_i , φ_j , and training points X . For example, in the case of 2D integral equations, we first vectorize the $d \times d$ matrix w :

$$\bar{w} = [w_{1,1}, w_{1,2}, \dots, w_{1,d}, w_{2,1}, w_{2,2}, \dots, w_{2,d}, \dots, w_{d,1}, w_{d,2}, \dots, w_{d,d}],$$

and then use new indexing function

$$w_{i,j} = \bar{w}_{i*d+j}.$$

For three-dimensional case the indexing function

$$w_{i,j,k} = \bar{w}_{i*d^2+j*d+k},$$

can be used. Also, this indexing should be applied to the basis function and training data tensor. After using this technique, the proposed dual form Eq. 9.8 can be utilized for a one-dimensional case. Solving the dual form returns the vector α which can be converted to a tensor using the inverse of the indexing function.

9.3.3 System of integral equations

In the system of integral equations, there are k equations and unknown functions:

$$\mathcal{N}_i(u_1, u_2, \dots, u_k) = f_i, \quad i = 1, 2, \dots, k. \quad (9.11)$$

For solving these types of equations, the approximated solution can be formulated as:

$$\tilde{u}_i(x) = w_i^T \varphi(x) + b_i, \quad i = 1, 2, \dots, k.$$

where, $\varphi(x)$ is feature map vector, and w_i and b_i are the unknown coefficients. In the next step, the unknown coefficients are set side by side in a vector form, thus, we have:

$$\bar{w} = [w_{1,1}, w_{1,2}, \dots, w_{1,d}, w_{2,1}, w_{2,2}, \dots, w_{2,d}, \dots, w_{k,1}, w_{k,2}, \dots, w_{k,d}].$$

Same as the previous formulation for solving high dimensional equations, the basis functions can vary for each approximate function, but for simplicity, the same functions are used. Since these functions are shared, they can be seen in a d -dimensional vector. Using this formulation, the optimization problem for solving Eq. 9.11 can be constructed as

$$\begin{aligned} \min_{w,e} \quad & \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e \\ \text{s.t.} \quad & \langle \mathcal{N}_i(\tilde{u}_1, \tilde{u}_2, \dots, \tilde{u}_k) - f_i, \psi_j \rangle = e_{i,j}, \quad j = 1, \dots, n, \end{aligned} \quad (9.12)$$

where $i = 1, 2, \dots, k$. Also the matrix $e_{i,j}$ should be vectorized same as unknown coefficients w . For any linear operator \mathcal{N} , denoted by \mathcal{L} , the dual form of the optimization problem Eq. 9.12 can be derived. Here we obtain the dual form for a system of two equations and two unknown functions. This process can be generalized for the arbitrary number of equations.

Suppose the following system of equations is given:

$$\begin{cases} \mathcal{L}_1(u_1, u_2) = f_1 \\ \mathcal{L}_2(u_1, u_2) = f_2 \end{cases}.$$

By approximating the solutions using,

$$\begin{aligned} \tilde{u}_1(x) &= w_1^T \varphi(x) + b_1, \\ \tilde{u}_2(x) &= w_2^T \varphi(x) + b_2, \end{aligned}$$

the optimization problem Eq. 9.12 takes the form

$$\begin{aligned} \min_{w,e} \quad & \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e \\ \text{s.t.} \quad & \langle \mathcal{L}_1(\tilde{u}_1, \tilde{u}_2) - f_1, \psi_j \rangle = e_j, \quad j = 1, \dots, n, \\ \text{s.t.} \quad & \langle \mathcal{L}_2(\tilde{u}_1, \tilde{u}_2) - f_2, \psi_j \rangle = e_j, \quad j = n+1, \dots, 2n, \end{aligned}$$

where

$$\begin{aligned} w &= [w_1, w_2] = [w_{1,1}, w_{1,2}, \dots, w_{1,d}, w_{2,1}, w_{2,2}, \dots, w_{2,d}], \\ e &= [e_1, e_2] = [e_{1,1}, e_{1,2}, \dots, e_{1,d}, e_{2,1}, e_{2,2}, \dots, e_{2,d}]. \end{aligned}$$

For the dual solution, the Lagrangian function is constructed as follows:

$$\begin{aligned} \mathfrak{L}(w, e, \alpha) &= \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e \\ &\quad - \sum_{j=1}^n \alpha_j \langle \mathcal{L}_1(\tilde{u}_1, \tilde{u}_2) - f_1, \psi_j \rangle - e_j \\ &\quad - \sum_{j=1}^n \alpha_{n+j} \langle \mathcal{L}_2(\tilde{u}_1, \tilde{u}_2) - f_2, \psi_j \rangle - e_{n+j}, \end{aligned}$$

then, the conditions for optimality of the Lagrangian function are given by

$$\frac{\partial \mathfrak{L}}{\partial w_k} = 0 \rightarrow w_k = \begin{cases} \sum_{j=1}^n \alpha_j \langle \mathcal{L}_1(\varphi_k, 0) - f_1, \psi_j \rangle - e_j + \\ \sum_{j=1}^n \alpha_{n+j} \langle \mathcal{L}_2(\varphi_k, 0) - f_2, \psi_j \rangle - e_{n+j}, & k = 1, 2, \dots, d. \end{cases}$$

(9.13)

$$\frac{\partial \mathfrak{L}}{\partial e_k} = 0 \rightarrow \gamma e_k + \alpha_k = 0, \quad k = 1, \dots, 2n.$$

$$\frac{\partial \mathfrak{L}}{\partial b} = 0 \rightarrow \begin{cases} \frac{\partial \mathfrak{L}}{\partial b_1} = 0 \rightarrow \sum_{i=1}^n \alpha_i \langle \mathcal{L}_1(1, 0), \psi_i \rangle + \sum_{i=1}^n \alpha_{n+i} \langle \mathcal{L}_2(1, 0), \psi_i \rangle, \\ \frac{\partial \mathfrak{L}}{\partial b_2} = 0 \rightarrow \sum_{i=1}^n \alpha_i \langle \mathcal{L}_1(0, 1), \psi_i \rangle + \sum_{i=1}^n \alpha_{n+i} \langle \mathcal{L}_2(0, 1), \psi_i \rangle, \end{cases}$$

$$\frac{\partial \mathfrak{L}}{\partial \alpha_k} = 0 \rightarrow \begin{cases} \sum_{j=1}^d w_j \langle \mathcal{L}_1(\varphi_j, 0) - f_1, \psi_k \rangle + \\ \sum_{j=1}^d w_{d+j} \langle \mathcal{L}_1(0, \varphi_j) - f_1, \psi_k \rangle = e_k & k = 1, 2, \dots, n. \\ \sum_{j=1}^d w_j \langle \mathcal{L}_2(\varphi_j, 0) - f_2, \psi_k \rangle + \\ \sum_{j=1}^d w_{d+j} \langle \mathcal{L}_2(0, \varphi_j) - f_2, \psi_k \rangle = e_k & k = n+1, n+2, \dots, 2n. \end{cases}$$

(9.14)

By defining

$$\begin{aligned} A_{i,j} &= \langle \mathcal{L}_1(\varphi_i, 0), \psi_j \rangle, \\ B_{i,j} &= \langle \mathcal{L}_2(\varphi_i, 0), \psi_j \rangle, \\ C_{i,j} &= \langle \mathcal{L}_1(0, \varphi_i), \psi_j \rangle, \\ D_{i,j} &= \langle \mathcal{L}_2(0, \varphi_i), \psi_j \rangle, \\ E_j &= \langle \mathcal{L}_1(1, 0), \psi_j \rangle, \\ F_j &= \langle \mathcal{L}_2(1, 0), \psi_j \rangle, \\ G_j &= \langle \mathcal{L}_1(0, 1), \psi_j \rangle, \\ H_j &= \langle \mathcal{L}_2(0, 1), \psi_j \rangle, \end{aligned}$$

and

$$Z = \begin{bmatrix} A & B \\ C & D \end{bmatrix},$$

$$V = \begin{bmatrix} E & F \\ G & H \end{bmatrix},$$

the relation Eq. 9.14 can be reformulated as

$$\begin{cases} Z\alpha = w \\ e = -\alpha/\gamma \\ b = V^T\alpha \\ Z^T w - e = y. \end{cases}$$

Eliminating w and e yields

$$\left[\begin{array}{c|c} \mathbf{0} & V^T \\ \hline V & \Omega + I/\gamma \end{array} \right] \left[\begin{array}{c} b \\ \alpha \end{array} \right] = \left[\begin{array}{c} 0 \\ y \end{array} \right], \quad (9.15)$$

where

$$\alpha = [\alpha_1, \dots, \alpha_{2n}]^T,$$

$$y = [\langle f_1, \psi_1 \rangle, \langle f_1, \psi_2 \rangle, \dots, \langle f_1, \psi_n \rangle, \langle f_2, \psi_1 \rangle, \langle f_2, \psi_2 \rangle, \dots, \langle f_2, \psi_n \rangle]^T,$$

and

$$\Omega = Z^T Z = \begin{bmatrix} A^T & C^T \\ B^T & D^T \end{bmatrix} \begin{bmatrix} A & B \\ C & D \end{bmatrix}. \quad (9.16)$$

The kernel trick also appears at each block of matrix Ω . The approximated solution in the dual form can be computed using:

$$\tilde{u}_1(x) = \sum_{i=1}^n \alpha_i \tilde{K}_1(x, x_i) + \sum_{i=1}^n \alpha_i \tilde{K}_2(x, x_i) + b_1,$$

$$\tilde{u}_2(x) = \sum_{i=1}^n \alpha_i \tilde{K}_3(x, x_i) + \sum_{i=1}^n \alpha_i \tilde{K}_4(x, x_i) + b_2,$$

where

$$\tilde{K}_1(x, x_i) = \langle \langle \mathcal{L}_1(\varphi, 0), \psi_i \rangle, \varphi \rangle = \langle \mathcal{L}_1(K(x, t), 0), \psi_i \rangle,$$

$$\tilde{K}_2(x, x_i) = \langle \langle \mathcal{L}_2(\varphi, 0), \psi_i \rangle, \varphi \rangle = \langle \mathcal{L}_2(K(x, t), 0), \psi_i \rangle,$$

$$\tilde{K}_3(x, x_i) = \langle \langle \mathcal{L}_1(0, \varphi), \psi_i \rangle, \varphi \rangle = \langle \mathcal{L}_1(0, K(x, t)), \psi_i \rangle,$$

$$\tilde{K}_4(x, x_i) = \langle \langle \mathcal{L}_2(0, \varphi), \psi_i \rangle, \varphi \rangle = \langle \mathcal{L}_2(0, K(x, t)), \psi_i \rangle.$$

9.3.4 CLS-SVR method

In this section, the collocation form of the LS-SVR model is proposed for solving integral equations. Similar to the weighted residual methods, by using the Dirac delta function as the test function in the test space, we can construct the collocation LS-SVR model, abbreviated as CLS-SVR. In this case, the primal form of optimization problem Eq. 9.5 can be expressed as

$$\begin{aligned} \min_{w,e} \quad & \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e \\ \text{s.t.} \quad & \mathcal{N}(\tilde{u})(x_k) - f(x_k) = e_k, \quad k = 1, \dots, n. \end{aligned} \quad (9.17)$$

where n is the number of training data and \mathcal{N} is a linear or non-linear functional operator. If the operator is linear, then the dual form of the optimization problem Eq. 9.17 can be computed using the following linear system of equations:

$$\left[\begin{array}{c|c} 0 & \tilde{\mathcal{L}}^T \\ \hline \tilde{\mathcal{L}} & \Omega + I/\gamma \end{array} \right] \begin{bmatrix} b \\ \alpha \end{bmatrix} = \begin{bmatrix} 0 \\ y \end{bmatrix}, \quad (9.18)$$

in which

$$\begin{aligned} \alpha &= [\alpha_1, \dots, \alpha_n]^T, \\ y &= [f(x_1), f(x_2), \dots, f(x_n)]^T, \\ \tilde{\mathcal{L}} &= [\mathcal{L}1(x_1), \mathcal{L}1(x_2), \dots, \mathcal{L}1(x_n)], \\ \Omega_{i,j} &= \mathcal{L}\varphi(x_i)^T \mathcal{L}\varphi(x_j) \\ &= \mathcal{L}\mathcal{L}K(x_i, x_j), \quad i, j = 1, 2, \dots, n. \end{aligned} \quad (9.19)$$

The approximated solution in the dual form takes the form:

$$\tilde{u}(x) = \sum_{i=1}^n \alpha_i \tilde{K}(x, x_i) + b,$$

where

$$\tilde{K}(x, x_i) = \mathcal{L}\varphi(x_i)^T \varphi(x) = \mathcal{L}K(x, x_i).$$

9.3.5 GLS-SVR method

The Galerkin approach is a famous method for solving a wide of problems. In this approach, the test functions ψ are chosen equal to the basis functions. If the basis functions are orthogonal together, this approach leads to a sparse system of algebraic equations. Some examples of this feature are provided in the next section. For now, let us define the model. In the primal space, the model can be constructed as:

$$\begin{aligned} \min_{w,e} \quad & \frac{1}{2} w^T w + \frac{\gamma}{2} e^T e \\ \text{s.t.} \quad & \int_{\Delta} [\mathcal{L}\tilde{u}(x) - f(x)] \varphi_k(x) dx = e_k, \quad k = 0, \dots, d. \end{aligned} \quad (9.20)$$

where d is the number of basis functions and \mathcal{N} is a linear or nonlinear functional operator. If the operator is linear, then the dual form of the optimization problem Eq. 9.20 can be computed using the following linear system:

$$\left[\begin{array}{c|c} 0 & \tilde{\mathcal{L}}^T \\ \hline \tilde{\mathcal{L}} & \Omega + I/\gamma \end{array} \right] \left[\begin{array}{c} b \\ \alpha \end{array} \right] = \left[\begin{array}{c} 0 \\ y \end{array} \right], \quad (9.21)$$

in which

$$\begin{aligned} \alpha &= [\alpha_1, \dots, \alpha_n]^T, \\ \tilde{\mathcal{L}} &= \left[\int_{\Delta} \mathcal{L}1(x) \varphi_1(x) dx, \int_{\Delta} \mathcal{L}1(x) \varphi_2(x) dx, \dots, \int_{\Delta} \mathcal{L}1(x) \varphi_d(x) dx, \right], \\ y &= \left[\int f(x) \varphi_1(x) dx, \int f(x) \varphi_2(x) dx, \dots, \int f(x) \varphi_d(x) dx \right]^T, \\ \Omega_{i,j} &= \left(\int_{\Delta} \mathcal{L}\varphi(x) \varphi_i(x) dx \right)^T \left(\int_{\Delta} \mathcal{L}\varphi(x) \varphi_j(x) dx \right) \\ &= \int_{\Delta} \int_{\Delta} \mathcal{L}\mathcal{L}K(s, t) \varphi_i(s) \varphi_j(t) ds dt, \quad i, j = 1, 2, \dots, d. \end{aligned} \quad (9.22)$$

The approximated solution in the dual form takes the form

$$\tilde{u}(x) = \sum_{i=1}^n \alpha_i \tilde{K}(x, x_i) + b,$$

where

$$\tilde{K}(x, x_i) = \left(\int_{\Delta} \mathcal{L}\varphi(s) \varphi_i(s) ds \right)^T \varphi(x) = \int_{\Delta} \mathcal{L}K(x, s) \varphi_i(s) ds.$$

9.4 Numerical simulations

In this section, some integral equations are considered as test problems and the efficiency of the proposed method is shown by approximating the solution of these test problems. Also, the shifted Legendre polynomials are used as the kernel of LS-SVR. Since the Legendre polynomial of degree 0 is a constant, the bias term b can be removed and the approximate solution is defined as $\sum_{i=0}^d w_i P_i(x)$. As a result, the system Eq. 9.8 reduces to $\Omega\alpha = y$, which can be efficiently solved using

the Cholesky decomposition or the conjugate gradient method. The training points for the following examples are the roots of the shifted Legendre polynomials, and the test data are equidistant points in the problem domain.

This method has been implemented in Maple 2019 software with 15 digits of accuracy. The results are obtained on an Intel Core i5 CPU with 8GB of RAM. In all of the presented numerical tables, the efficiency of the method is computed using the mean absolute error function:

$$L(u, \tilde{u}) = \frac{1}{n} \sum_{i=1}^n |u_i - \tilde{u}_i|,$$

where u_i and \tilde{u}_i are the exact and the predicted value at x_i , respectively.

Example 9.1 Suppose the following Volterra integral equation of the second kind with the exact solution $\ln(1+x)$ [36].

$$x - \frac{1}{2}x^2 - \ln(1+x) + x^2 \ln(1+x) = \int_0^x 2tu(t)dt.$$

In Table 9.1, the error of the approximate solutions for the CLS-SVR and GLS-SVR are given. Fig. 9.1 shows the approximate and the residual function of the approximate solution. Also, the non-zero elements of the matrix Ω of the GLS-SVR method are drawn. It is observed that most of the matrix elements are approximately zero.

Table 9.1: The convergence of the CLS-SVR and GLS-SVR methods for example 9.1 by different d values. The number of training points for each approximation is $d+1$. The CPU time is also reported in seconds

| d | CLS-SVR | | | GLS-SVR | | |
|-----|----------|----------|------|----------|----------|------|
| | Train | Test | Time | Train | Test | Time |
| 4 | 7.34E-05 | 7.65E-05 | 0.01 | 3.26E-05 | 5.58E-05 | 0.08 |
| 6 | 1.96E-06 | 2.32E-06 | 0.02 | 8.80E-07 | 1.68E-06 | 0.17 |
| 8 | 5.35E-08 | 7.50E-08 | 0.03 | 2.41E-08 | 4.61E-08 | 0.21 |
| 10 | 1.47E-09 | 2.14E-09 | 0.03 | 6.65E-10 | 1.48E-09 | 0.26 |
| 12 | 5.12E-11 | 8.92E-11 | 0.07 | 1.90E-11 | 4.26E-11 | 0.38 |

Example 9.2 Consider the following Fredholm integral equation of the first kind. As stated before, these equations are ill-posed, and their solution may not be unique. A solution for this equation is $\exp(x)$ [36]:

$$\frac{1}{4}e^x = \int_0^{\frac{1}{4}} e^{x-t} u(t)dt.$$

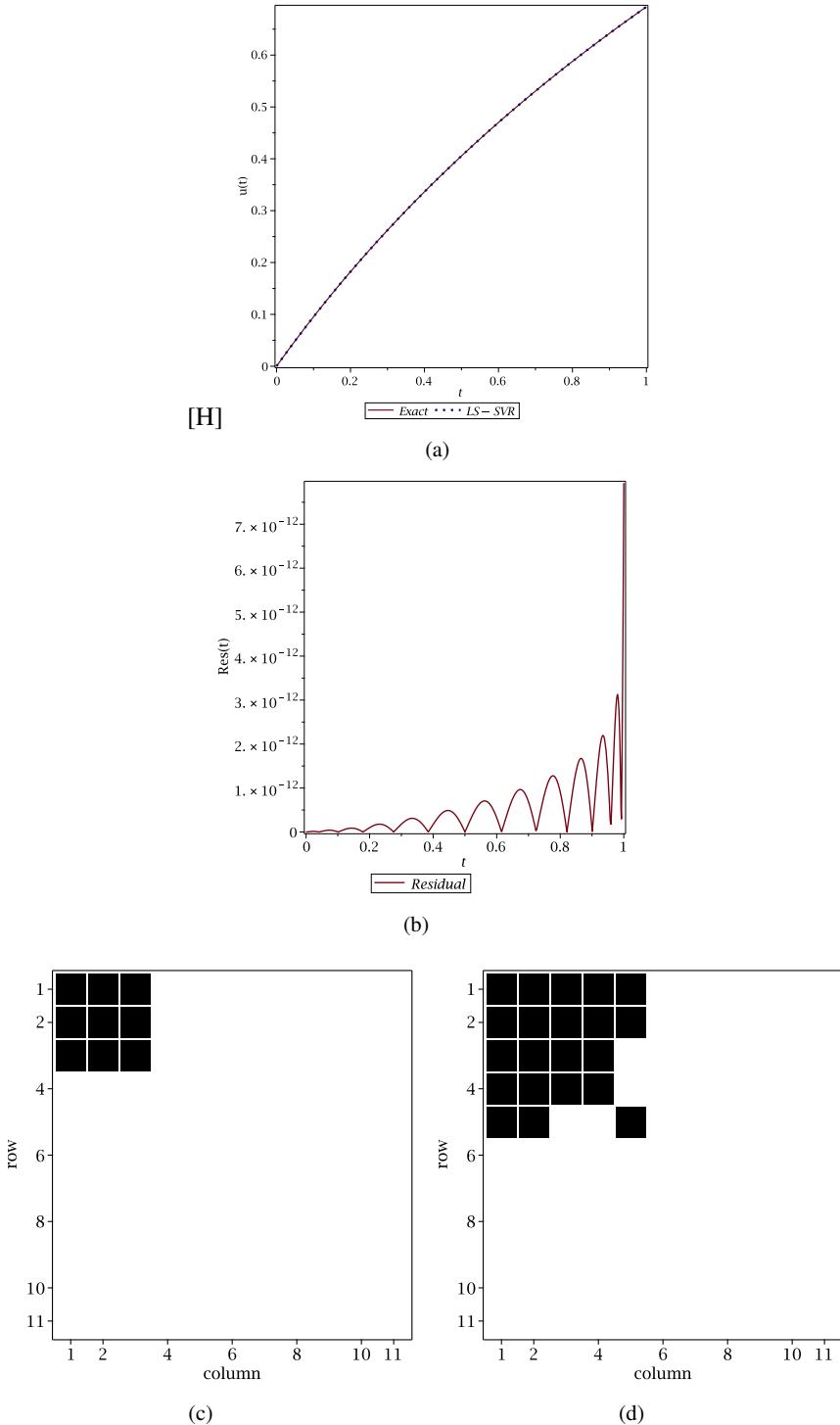


Fig. 9.1: The plots of example 9.1 (a) Exact vs. LS-SVR (b) Absolute of the residual function (c, d) Sparsity of matrix Ω in the GLS-SVR with fuzzy zero 10^{-3} and 10^{-4}

In Table 9.2, the obtained results of solving these equations with different values for γ with $d = 6$ and $n = 7$ are reported.

Table 9.2: The obtained solution norm and training error of the CLS-SVR method for example 9.2

| γ | $\ w\ _2$ | $\ e\ _2$ |
|----------|-----------|-----------|
| 1E-01 | 0.041653 | 0.673060 |
| 1E+00 | 0.312717 | 0.505308 |
| 1E+01 | 0.895428 | 0.144689 |
| 1E+02 | 1.100491 | 0.017782 |
| 1E+03 | 1.126284 | 0.001820 |
| 1E+04 | 1.128930 | 0.000182 |
| 1E+05 | 1.129195 | 0.000018 |

Example 9.3 Suppose the Volterra-Fredholm integral equation as following:

$$u(x) = 2e^x - 2x - 2 + \int_0^x (x-t)u(t)dt + \int_0^1 xu(t)dt.$$

The exact solution of this equation is $\exp(x)$ [71]. Table 9.3 shows the convergence of the proposed method for this equation. Fig. 9.2 is plotted the exact solution and the residual function of the approximate solution. Also, it is seen that the matrix Ω of the GLS-SVR method, has good sparsity.

Table 9.3: The convergence of the CLS-SVR and GLS-SVR method for example 9.3 by different d values. The number of training points for each approximation is $d + 1$. The CPU time is reported in seconds

| d | CLS-SVR | | | GLS-SVR | | |
|-----|----------|----------|------|----------|----------|------|
| | Train | Test | Time | Train | Test | Time |
| 4 | 1.53E-07 | 1.18E-04 | 0.02 | 4.22E-06 | 1.18E-04 | 0.13 |
| 6 | 1.16E-10 | 2.46E-07 | 0.04 | 6.04E-09 | 2.46E-07 | 0.18 |
| 8 | 5.90E-14 | 2.71E-10 | 0.04 | 5.13E-12 | 2.72E-10 | 0.28 |
| 10 | 4.12E-15 | 2.44E-13 | 0.07 | 2.21E-14 | 2.40E-13 | 0.38 |
| 12 | 6.84E-15 | 8.80E-15 | 0.12 | 2.12E-14 | 1.55E-14 | 0.60 |

Example 9.4 For a multi-dimensional case, consider the following 2D Fredholm integral equation [69]:

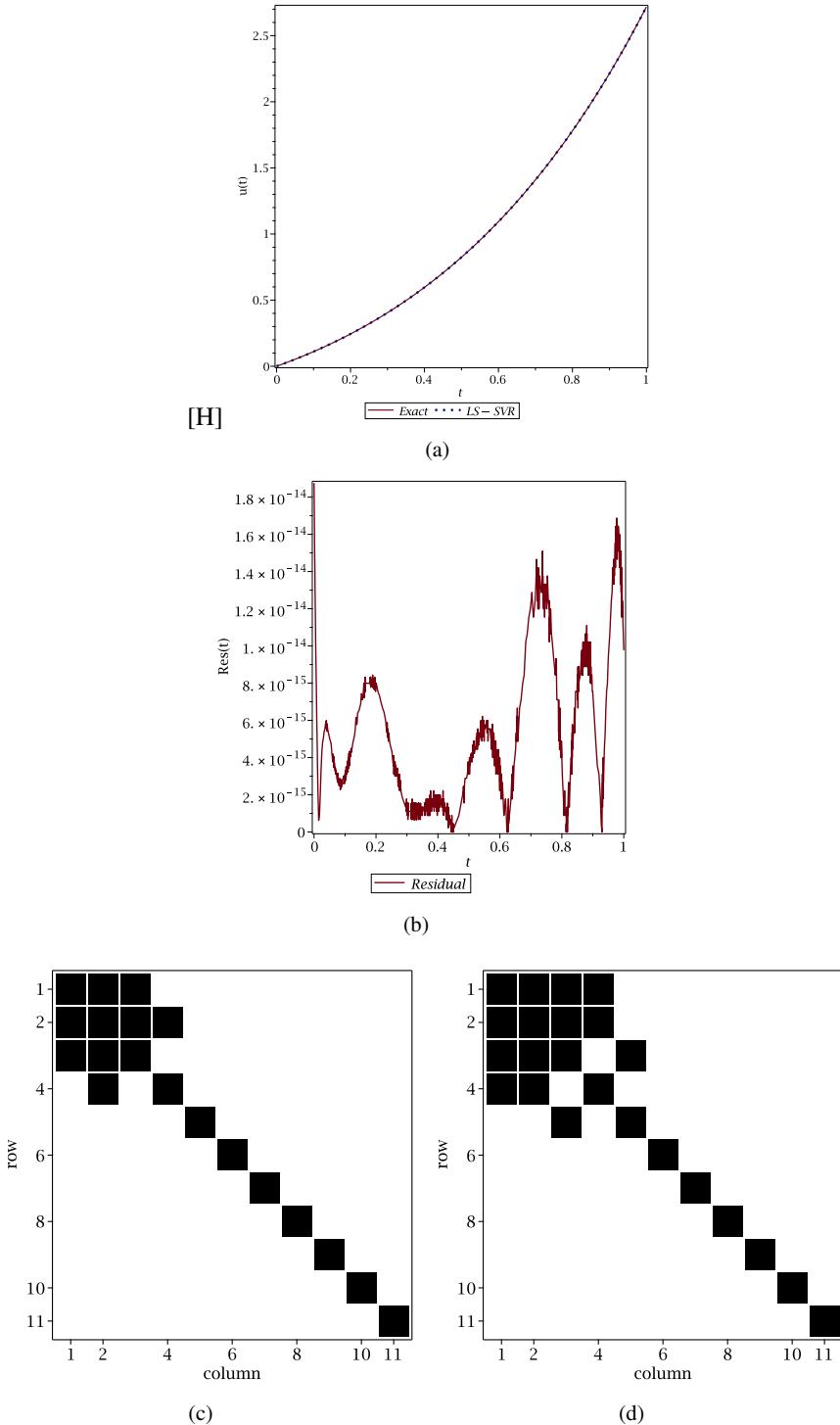


Fig. 9.2: The plots of example 9.3 (a) Exact vs. LSSVR (b) Absolute of the residual function (c, d) Sparsity of the matrix Ω in the GLS-SVR with fuzzy zero 10^{-3} and 10^{-4}

$$u(x, y) = x \cos(y) - \frac{1}{6}(\sin(1) + 3) \sin(1) + \int_0^1 \int_0^1 (s \sin(t) + 1) u(s, t) ds dt.$$

The exact solution of this equation is $u(x, y) = x \cos(y)$. The numerical results of the CLS-SVR and GLS-SVR methods are given in Table 9.4. Fig. 9.3 shows the plot of the exact solution and the residual function. Also, the sparsity of the methods in the two-dimensional case is attached. It can be seen that the sparsity dominates in the higher dimensional problems [66].

Table 9.4: The convergence of the CLS-SVR and GLS-SVR methods for example 9.4 by different d values. The CPU time is also reported in seconds

| d | n | CLS-SVR | | | GLS-SVR | | |
|-----|-----|----------|----------|------|----------|----------|------|
| | | Train | Test | Time | Train | Test | Time |
| 1 | 4 | 1.93E-03 | 1.93E-03 | 0.01 | 8.54E-04 | 8.12E-04 | 0.06 |
| 2 | 9 | 1.33E-05 | 1.33E-05 | 0.05 | 1.69E-05 | 5.86E-06 | 0.16 |
| 3 | 16 | 5.56E-08 | 5.56E-08 | 0.11 | 3.12E-07 | 2.34E-08 | 0.38 |
| 4 | 25 | 1.73E-10 | 1.73E-10 | 0.26 | 1.59E-08 | 8.16E-11 | 0.80 |
| 5 | 36 | 2.19E-11 | 2.19E-11 | 0.73 | 2.19E-11 | 2.19E-11 | 1.57 |

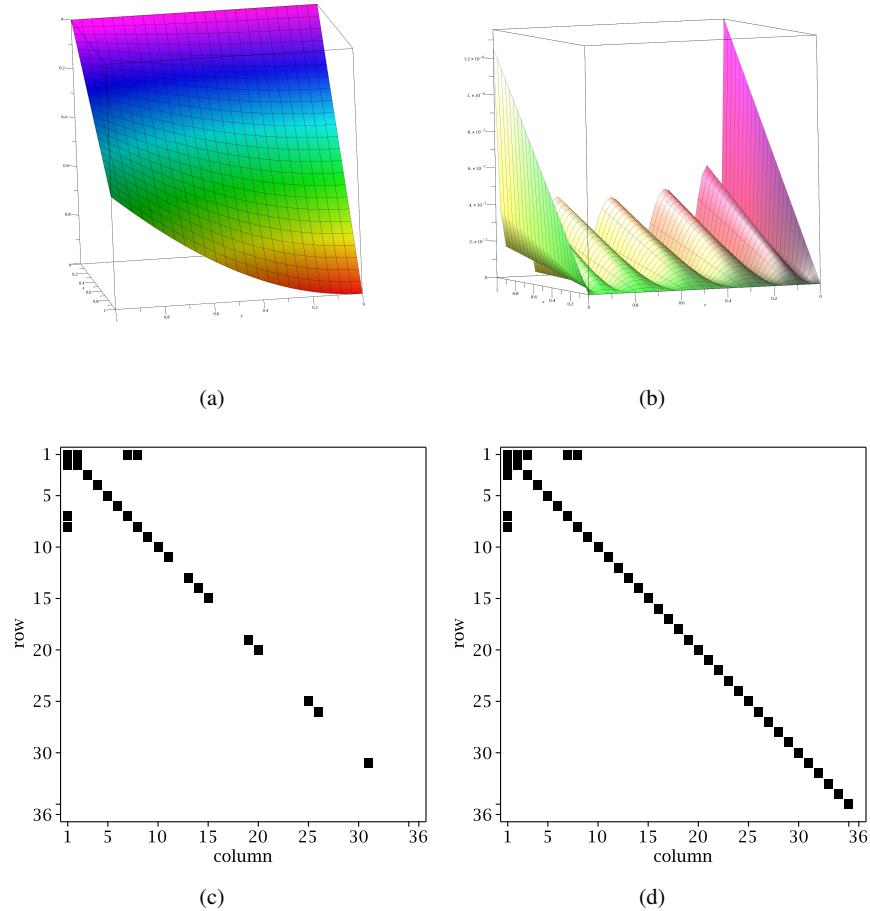


Fig. 9.3: The plots of example 9.4(a) Exact solution(b) Absolute of the residual function (c, d) Sparsity of matrix Ω in the GLS-SVR with fuzzy zero 10^{-3} and 10^{-4}

Example 9.5 Consider the following system of integral equation with the exact solution $u_1(x) = \sin(x)$, $u_2(x) = \cos(x)$ [36].

$$\begin{cases} u_1(x) = \sin x - 2 - 2x - \pi x + \int_0^\pi ((1+xt)u_1(t) + (1-xt)u_2(t))dt, \\ u_2(x) = \cos x - 2 - 2x + \pi x + \int_0^\pi ((1-xt)u_1(t) - (1+xt)u_2(t))dt. \end{cases}$$

The numerical simulation results of this example are given in Table 9.5. Also, Fig. 9.4 plots the exact solution and approximate solution of the methods. Since the matrix

Ω in this case is a symmetric block matrix, the resulting matrix of the GLS-SVR has an interesting structure.

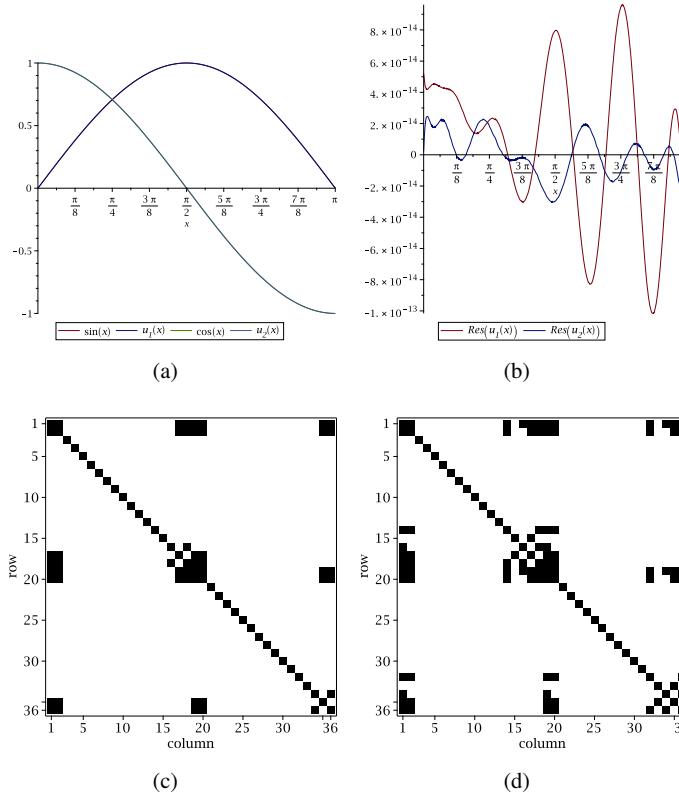


Fig. 9.4: The plots of example 9.5 (a) Exact vs. LS-SVR (b) Absolute of the residual functions (c, d) Sparsity of matrix Ω in the GLS-SVR with fuzzy zero 10^{-3} and 10^{-4}

Table 9.5: The convergence of mean squared error value for the CLS-SVR and GLS-SVR methods in example 9.5 by different d values. The number of training points for each approximation is $d + 1$. The CPU time is reported in seconds

| d | u_1 | | u_2 | | | Time |
|-----|----------|----------|----------|----------|------|------|
| | Train | Test | Train | Test | Time | |
| 4 | 9.02E-13 | 1.20E-06 | 2.28E-13 | 2.00E-05 | 0.34 | |
| 7 | 5.43E-28 | 6.04E-11 | 2.22E-27 | 1.67E-12 | 0.48 | |
| 10 | 3.07E-26 | 3.35E-19 | 3.29E-27 | 2.18E-17 | 0.66 | |
| 13 | 8.19E-27 | 1.36E-24 | 2.09E-27 | 3.86E-26 | 0.91 | |
| 16 | 2.38E-27 | 2.31E-27 | 1.87E-28 | 1.85E-28 | 1.38 | |

In Fig. 9.4(a) the the solutions and the approximations are not discriminable.

Example 9.6 For a non-linear example, consider the following Volterra-Fredholm integral equation of the second kind [70]

$$u(x) = \frac{1}{36}(35 \cos(x) - 1) + \frac{1}{12} \int_0^t \sin(t) u^2(t) dt + \frac{1}{36} \int_0^{\frac{\pi}{2}} (\cos^3(x) - \cos(x)) u(t) dt.$$

The exact solution of this equation is $u(x) = \cos(x)$ Since the equation is non-linear, the corresponding optimization problem leads to a non-linear programming problem. Also, The dual form yields a system of non-linear algebraic equations. In Table 9.6 and Fig. 9.5 the numerical results and the convergence of the method can be seen.

Table 9.6: The convergence of the CLS-SVR and GLS-SVR methods for example 9.6 by different d values. The number of training points for each approximation is $d + 1$. The CPU time is also reported in seconds

| d | CLS-SVR | | | GLS-SVR | | |
|-----|----------|----------|------|----------|----------|-------|
| | Train | Test | Time | Train | Test | Time |
| 2 | 9.23E-05 | 8.66E-03 | 0.08 | 8.55E-04 | 8.76E-03 | 0.72 |
| 4 | 4.24E-07 | 8.45E-05 | 0.12 | 4.28E-06 | 8.5E-05 | 1.54 |
| 6 | 1.12E-09 | 3.22E-07 | 0.15 | 1.16E-08 | 3.23E-07 | 3.84 |
| 8 | 1.99E-12 | 6.97E-10 | 0.23 | 7.82E-11 | 7.39E-10 | 12.04 |
| 10 | 3.07E-13 | 1.22E-12 | 0.33 | 1.31E-10 | 1.71E-10 | 38.37 |

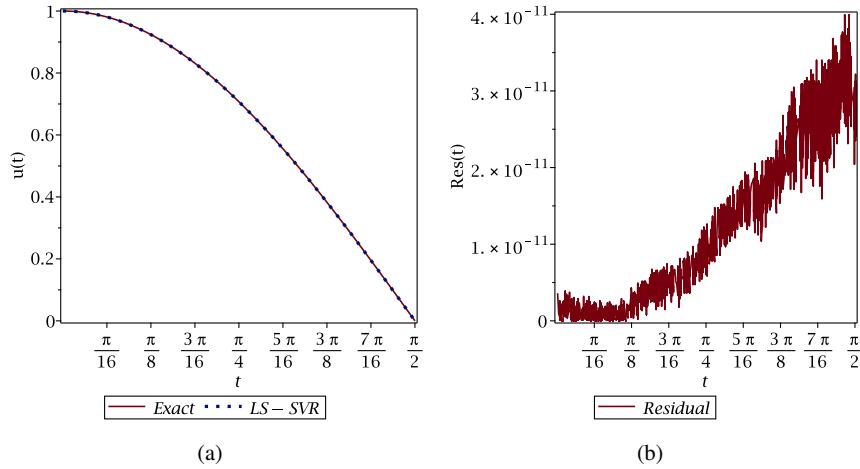


Fig. 9.5: The plots of example 9.5 (a) Exact vs. LS-SVR (b) Absolute of the residual function

9.5 Conclusion

In this chapter, a new computational method for solving different types of integral equations, including multi-dimensional cases and systems of integral equations is proposed. In linear equations, learning the solution reduces to solving a positive definite system of linear equations. This formulation was similar to the LS-SVR method for solving regression problems. By using the ideas behind spectral methods, we have presented CLS-SVR and GLS-SVR methods. Although the CLS-SVR method is more computationally efficient, the resulting matrix in the GLS-SVR method has a sparsity property. In the last section, some integral equations have been solved using the CLS-SVR and GLS-SVR methods. The numerical results show that these methods have high efficiency and exponential convergence rate for integral equations.

References

1. Eswaran, K.: On the solutions of a class of dual integral equations occurring in diffraction problems. *Proc. Math. Phys. Eng. Sci.* **429**, 399–427 (1990)
2. Barlette, V. E., Leite, M. M., Adhikari, S. K.: Integral equations of scattering in one dimension. *Am. J. Phys.* **69**, 1010–1013 (2001)
3. Kanaun, S., Martinez, R.: Numerical solution of the integral equations of elasto-plasticity for a homogeneous medium with several heterogeneous inclusions. *Comput. Mater. Sci.* **55**, 147–156 (2012)
4. Reichel, L.: A fast method for solving certain integral equations of the first kind with application to conformal mapping. *J. Comput. Appl. Math.* **14**, 125–142 (1986)

5. Manam, S. R.: Multiple integral equations arising in the theory of water waves. *Appl. Math. Lett.* **24**, 1369–1373 (2011)
6. Mohammad, M.: A numerical solution of Fredholm integral equations of the second kind based on tight framelets generated by the oblique extension principle. *Symmetry*, **11**, 854–869 (2019)
7. Wazwaz, A. M.: A reliable treatment for mixed Volterra–Fredholm integral equations. *Appl. Math. Comput.* **127**, 405–414 (2002)
8. Chen, R. T., Rubanova, Y., Bettencourt, J., Duvenaud, D. K.: Neural ordinary differential equations. *Advances in neural information processing systems*, **31** (2018)
9. Assari, P., Dehghan, M.: On the numerical solution of logarithmic boundary integral equations arising in laplace's equations based on the meshless local discrete collocation method. *Adv. Appl. Math. Mech* **11**, 807–837 (2019)
10. Bažant, Z. P., Jirásek, M.: Nonlocal integral formulations of plasticity and damage: survey of progress. *J. Eng. Mech.* **128**, 1119–1149 (2002)
11. Bremer, J.: A fast direct solver for the integral equations of scattering theory on planar curves with corners. *J. Comput. Phys.* **231**, 1879–1899 (2012)
12. Mikhlin, S. G.: *Multidimensional singular integrals and integral equations*. Elsevier, (2014)
13. Zaky, M. A., Ameen, I. G., Elkot, N. A., Doha, E. H.: A unified spectral collocation method for nonlinear systems of multi-dimensional integral equations with convergence analysis. *Appl Numer Math* **161**, 27–45 (2021)
14. Abdelkawy, M. A., Amin, A. Z., Bhrawy, A. H., Machado, J. A. T., Lopes, A. M.: Jacobi collocation approximation for solving multi-dimensional Volterra integral equations. *Int. J. Nonlinear Sci. Numer. Simul.* **18**, 411–425 (2017)
15. Bhrawy, A. H., Abdelkawy, M. A., Machado, J. T., Amin, A. Z. M.: Legendre–Gauss–Lobatto collocation method for solving multi-dimensional Fredholm integral equations. *Comput. Math. Appl* **4**, 1–13 (2016)
16. Kulish, V. V., Novozhilov, V. B.: Integral equation for the heat transfer with the moving boundary. *J Thermophys Heat Trans* **17**, 538–540 (2003)
17. Esmaeilbeigi, M., Mirzaee, F., Moazami, D.: A meshfree method for solving multidimensional linear Fredholm integral equations on the hypercube domains. *Appl. Math. Comput.* **298**, 236–246 (2017)
18. Lu, Y., Yin, Q., Li, H., Sun, H., Yang, Y., Hou, M.: Solving higher order nonlinear ordinary differential equations with least squares support vector machines. *J. Ind. Manag. Optim.* **16**, 1481–1502 (2020)
19. Mirzaee, F., Alipour, S.: An efficient cubic B-spline and bicubic B-spline collocation method for numerical solutions of multidimensional nonlinear stochastic quadratic integral equations. *Math. Methods Appl. Sci.* **43**, 384–397 (2020)
20. Parand, K., Delkhosh, M.: Solving the nonlinear Schliomilch's integral equation arising in ionospheric problems. *Afr. Mat.* **28**, 459–480 (2017)
21. Volterra, V.: Variations and fluctuations of the number of individuals in animal species living together. *ICES Mar. Sci. Symp.* **3**, 3–51 (1928)
22. Keller, A., Dahm, K.: Integral equations and machine learning. *Math Comput Simul* **161**, 2–12 (2019)
23. Dahm, K., Keller, A.: Learning light transport the reinforced way. In International Conference on Monte Carlo and Quasi-Monte Carlo Methods in Scientific Computing, 181–195 (2016)
24. Abbasbandy, S.: Numerical solutions of the integral equations: Homotopy perturbation method and Adomian's decomposition method. *Appl. Math. Comput.* **173**, 493–500 (2006)
25. Assari, P., Dehghan, M.: A meshless local discrete Galerkin (MLDG) scheme for numerically solving two-dimensional nonlinear Volterra integral equations. *Appl. Math. Comput.* **350**, 249–265 (2019)
26. Fatahi, H., Saberi-Nadjafi, J., Shivanian, E.: A new spectral meshless radial point interpolation (SMRPI) method for the two-dimensional Fredholm integral equations on general domains with error analysis. *J. Comput. Appl. Math* **294**, 196–209 (2016)
27. Golberg, M. A.: *Numerical solution of integral equations*. Springer, Berlin (2013)

28. Mandal, B. N., Chakrabarti, A.: Applied singular integral equations. CRC press, Florida (2016)
29. Marzban, H. R., Tabrizidooz, H. R., Razzaghi, M.: A composite collocation method for the nonlinear mixed Volterra–Fredholm–Hammerstein integral equations. *Commun Nonlinear Sci Numer Simul* **16**, 1186–1194 (2011)
30. Amiri, S., Hajipour, M., Baleanu, D.: On accurate solution of the Fredholm integral equations of the second kind. *Appl Numer Math* **150**, 478–490 (2020)
31. Maleknejad, K., Nosrati Sahlan, M.: The method of moments for solution of second kind Fredholm integral equations based on B-spline wavelets. *Int. J. Comput. Math.* **87**, 1602–1616 (2010)
32. Li, X. Y., Wu, B. Y.: Superconvergent kernel functions approaches for the second kind Fredholm integral equations. *Appl Numer Math* **167**, 202–210 (2021)
33. Rad, J. A., Parand, K.: Numerical pricing of American options under two stochastic factor models with jumps using a meshless local Petrov-Galerkin method. *Applied Numerical Mathematics* **115**, 252–274 (2017)
34. Rad, J. A., Parand, K.: Pricing American options under jump-diffusion models using local weak form meshless techniques. *International Journal of Computer Mathematics* **94**, 1694–1718 (2017)
35. Nematni, S., Lima, P. M., Ordokhani, Y.: Numerical solution of a class of two-dimensional nonlinear Volterra integral equations using Legendre polynomials. *J. Comput. Appl* **242**, 53–69 (2013)
36. Wazwaz, A. M.: Linear and nonlinear integral equations. Springer, Berlin (2011)
37. Wazwaz, A. M. First Course In Integral Equations, A. World Scientific Publishing Company (2015)
38. Rahman, M.: Integral equations and their applications. WIT press (2007)
39. Oldham, K., Spanier, J.: The fractional calculus theory and applications of differentiation and integration to arbitrary order. Elsevier, Amsterdam (1974)
40. Jafari, H., Ganji, R. M., Nkomo, N. S., Lv, Y. P.: A numerical study of fractional order population dynamics model. *Results Phys.* **27**, 104456 (2021)
41. Wang, G. Q., Cheng, S. S.: Nonnegative periodic solutions for an integral equation modeling infectious disease with latency periods. In *Intern. Math. Forum* **1**, 421–427 (2006)
42. Unterreiter, A.: Volterra integral equation models for semiconductor devices. *Math. Methods Appl. Sci.* **19**, 425–450 (1996)
43. Miller, K. S., Ross, B.: An introduction to the fractional calculus and fractional differential equations. Wiley, New York (1993)
44. Messina, E., Vecchio, A.: Stability and boundedness of numerical approximations to Volterra integral equations. *Appl Numer Math* **116**, 230–237 (2017)
45. Tang, T., Xu, X., Cheng, J.: On spectral methods for Volterra integral equations and the convergence analysis. *J. Comput. Math.* **26** 825–837 (2008)
46. Maleknejad, K., Hashemizadeh, E., Ezzati, R.: A new approach to the numerical solution of Volterra integral equations by using Bernstein’s approximation. *Commun. Nonlinear Sci. Numer. Simul.* **16**, 647–655 (2011)
47. Assari, P., Asadi-Mehregan, F., Dehghan, M.: On the numerical solution of Fredholm integral equations utilizing the local radial basis function method. *Int J Comput Math* **96**, 1416–1443 (2019)
48. Parand, K., Rad, J. A.: Numerical solution of nonlinear Volterra–Fredholm–Hammerstein integral equations via collocation method based on radial basis functions. *Appl. Math. Comput.* **218**, 5292–5309 (2012)
49. Maleknejad, K., Almasieh, H., Roodaki, M.: Triangular functions (TF) method for the solution of nonlinear Volterra–Fredholm integral equations. *Commun. Nonlinear Sci. Numer. Simul.* **15**, 3293–3298 (2010)
50. Yousefi, S., Razzaghi, M.: Legendre wavelets method for the nonlinear Volterra–Fredholm integral equations. *Math Comput Simul* **70**, 1–8 (2005)
51. Ghasemi, M., Kajani, M. T., Babolian, E.: Numerical solutions of the nonlinear Volterra–Fredholm integral equations by using homotopy perturbation method. *Appl. Math. Comput.* **188**, 446–449 (2007)

52. Babolian, E., Masouri, Z., Hatamzadeh-Varmazyar, S.: Numerical solution of nonlinear Volterra–Fredholm integro-differential equations via direct method using triangular functions. *Comput. Math. with Appl.* **58**, 239–247 (2009)
53. Babolian, E., Shaerlar, A. J.: Two dimensional block pulse functions and application to solve Volterra-Fredholm integral equations with Galerkin method. *Int. J. Contemp. Math. Sci* **6**, 763–770 (2011)
54. Maleknejad, K., Hadizadeh, M.: A new computational method for Volterra-Fredholm integral equations. *Comput. Math. with Appl.* **37**, 1–8 (1999)
55. Bahmanpour, M., Kajani, M. T., Maleki, M.: Solving Fredholm integral equations of the first kind using Müntz wavelets. *Appl Numer Math.* **143**, 159–171 (2019)
56. Dehghan, M.: Solution of a partial integro-differential equation arising from viscoelasticity. *Int. J. Comput. Math.* **83**, 123–129 (2006)
57. Dehghan, M., Saadatmandi, A.: Chebyshev finite difference method for Fredholm integro-differential equation. *Int. J. Comput. Math.* **85**, 123–130 (2008)
58. El-Shahed, M.: Application of He's homotopy perturbation method to Volterra's integro-differential equation. *Int. J. Nonlinear Sci. Numer. Simul.* **6**, 163–168 (2005)
59. Wang, S. Q., He, J. H.: Variational iteration method for solving integro-differential equations. *Phys. Lett. A* **367**, 188–191 (2007)
60. Parand, K., Abbasbandy, S., Kazem, S., Rad, J. A.: A novel application of radial basis functions for solving a model of first-order integro-ordinary differential equation. *Commun Nonlinear Sci Numer Simul* **16**, 4250–4258 (2011)
61. Parand, K., Hosseini, S. A., Rad, J. A.: An operation matrix method based on Bernstein polynomials for Riccati differential equation and Volterra population model. *Appl. Math. Model.* **40**, 993–1011 (2016)
62. Parand, K., Rad, J. A.: An approximation algorithm for the solution of the singularly perturbed Volterra integro-differential and Volterra integral equations. *Int. J. of Nonlinear Science* **12**, 430–441 (2011)
63. Parand, K., Rad, J. A., Nikarya, M.: A new numerical algorithm based on the first kind of modified Bessel function to solve population growth in a closed system. *Int. J. Comput. Math.* **91**, 1239–1254 (2014)
64. Parand, K., Yari, H., Taheri, R., Shekarpaz, S.: A comparison of Newton–Raphson method with Newton–Krylov generalized minimal residual (GMRes) method for solving one and two dimensional nonlinear Fredholm integral equations. *Sema* **76**, 615–624 (2019)
65. Brunner, H.: On the numerical solution of nonlinear Volterra–Fredholm integral equations by collocation methods. *SIAM J. Numer. Anal.* **27**, 987–1000 (1990)
66. Parand, K., Aghaei, A. A., Jani, M., Ghodsi, A.: A new approach to the numerical solution of Fredholm integral equations using least squares-support vector regression. *Math Comput Simul* **180**, 114–128 (2021)
67. Parand, K., Delafkar, Z., Pakniat, N., Pirkhedri, A., Haji, M. K.: Collocation method using sinc and Rational Legendre functions for solving Volterra's population model. *Commun Nonlinear Sci Numer Simul* **16**, 1811–1819 (2011)
68. Maleknejad, K., Shahrezaee, M.: Using Runge–Kutta method for numerical solution of the system of Volterra integral equation. *Appl. Math. Comput.* **149**, 399–410 (2004)
69. Derakhshan, M., Zarebnia, M.: On the numerical treatment and analysis of two-dimensional Fredholm integral equations using quasi-interpolant. *Comput. Appl. Math.* **39**, 1–20 (2020)
70. Amiri, S., Hajipour, M., Baleanu, D.: A spectral collocation method with piecewise trigonometric basis functions for nonlinear Volterra–Fredholm integral equations. *Appl. Math. Comput.* **370**, 124915 (2020)
71. Malaikah, H. M.: The adomian decomposition method for solving Volterra–Fredholm integral equation using maple. *Appl. Math.* **11**, 779–787 (2020)
72. Parand, K., Razzaghi, M., Sahleh, R., Jani, M.: Least squares support vector regression for solving Volterra integral equations. *Eng Comput* **38**, 38, 789–796 (2022)
73. Mirzaei, D., Dehghan, M.: A meshless based method for solution of integral equations. *Appl Numer Math* **60**, 245–262 (2010)

Chapter 10

Solving Distributed-Order Fractional Equations by LS-SVR

Amir Hosein Hadian Rasanan and Arsham Gholamzadeh Khoee and Mostafa Jani

Abstract Over the past years, several artificial intelligence methods have been developed for solving various types of differential equations; due to their potential to deal with different problems, so many paradigms of artificial intelligence algorithms such as evolutionary algorithms, neural networks, deep learning methods, and support vector machine algorithms, have been applied to solve them. In this chapter, an artificial intelligence method has been employed to approximate the solution of distributed order fractional differential equations, which is based on the combination of least squares support vector regression algorithm and a well-defined spectral method named collocation approach. The importance of solving distributed order fractional differential equations is being used to model some significant phenomena that existed in nature, such as the ones in viscoelasticity, diffusion, sub-diffusion, and wave. We used the modal Legendre functions, which have been introduced in Chapter 4 and have been applied several times in the previous chapters as the least squares support vector regression algorithm's kernel basis in the presented method. The uniqueness of the obtained solution is proved for the resulting linear systems. Finally, the efficiency and applicability of the proposed algorithm are determined by the results of applying it to different linear and nonlinear test problems.

Amir Hosein Hadian Rasanan

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Tehran, Iran, e-mail: amir.h.hadian@gmail.com

Arsham Gholamzadeh Khoee

Department of Computer Science, School of Mathematics, Statistics, and Computer Science, University of Tehran, Tehran, Iran e-mail: arsham.khoee@ut.ac.ir

Mostafa Jani

Department of Computer and Data Science, Faculty of Mathematical Sciences, Shahid Beheshti University, Tehran, Iran e-mail: mostafa.jani@gmail.com

10.1 Introduction

Fractional differential equations are frequently used for modeling real-world problems in science and engineering, such as fractional time evolution, polymer physics, rheology, and thermodynamics [1], cognitive science [2, 3], neuroscience [4]. Since the analytical methods can only solve some special simple cases of fractional differential equations, numerical methods are developed for solving these equations involving linear and nonlinear cases. On the other hand, since the fractional calculus foundation, several definitions have been presented by scientists such as Liouville [1], Caputo [5, 6], Riesz [7], and Atangana-Baleanu-Caputo [8]. So solving fractional differential equations is still a challenging problem.

In modeling some phenomena, the order of derivatives depends on time (i.e., derivative operators' order as a function of t). It is for the memory property of the variable-order fractional operators [9]. A similar concept to the variable order differentiation is the distributed order differentiation, which indicates a continuous form of weighted summation of various orders chosen in an interval. In many problems, differential equations can have different orders. For instance, the first-order differential equation can be written by the summation of various orders along with their proper weight functions as follows

$$\sum_{j=0}^n \omega_j D_t^{\alpha_j} u, \quad (10.1)$$

in which α_j s are descending and have equal distance to each other, ω_j s can be determined by using the data. It is worth mentioning, by calculating the limit of the above equation, convert to the following form, which is convergence:

$$\int_0^1 \omega(x) D^\alpha u dx. \quad (10.2)$$

Distributed order fractional differential equations (DOFDEs) have been considered since 2000 [52]. DOFDEs have many applications in different areas of physics and engineering. For example, DOFDEs can describe various phenomena in viscoelastic material [10, 11], statistical and solid mechanics [12, 13]. System identification is also a significant application of DOFDEs [14]. Moreover, in the study of the anomalous diffusion process, DOFDEs provide more compatible models with experimental data [15, 16]. Also, in signal processing and system controls, DOFDEs have been used frequently [17]. In this chapter, an algorithm based on a combination of the least squares support vector regression (LS-SVR) algorithm, and modal Legendre collocation method is presented for solving a DOFDE given by [28, 48]:

$$\int_a^b G_1(p, D^p u(t)) dp + G_2(t, u(t), D^{\alpha_i} u(t)) = F(t), \quad t > 0, \quad (10.3)$$

with the initial conditions:

$$u^{(k)}(0) = 0, \quad (10.4)$$

where $i \in \mathbb{N}$, $\alpha_i > 0$, $k = 0, 1, \dots, \lfloor \max\{b, \alpha_i\} \rfloor$ and G_1, G_2 can be linear or nonlinear functions. In the proposed algorithm, the unknown function is expanded by using modal Legendre polynomials; then, using the support vector regression algorithm and collocating the training points in the residual function, the problem is converted to a constrained optimization problem. By solving this optimization problem, the solution of DOFDEs is obtained. Since DOFDEs have a significant role in various fields of science, many researchers have been developed several numerical algorithms for solving them. In the following section, we review the existing methods used to solve DOFDEs and some artificial intelligence algorithms that are developed to solve different types of differential equations.

10.1.1 A brief review of other methods existing in the literature

Najafi et al. analyzed the stability of three classes of DOFDEs subjected to the nonnegative density function [18]. Atanacković et al. studied the existence, and the uniqueness of mild and classical solutions for the specific general form, which are arisen in distributed derivative models of viscoelasticity and identification theory [19]. He and his collaborators also studied some properties of the distributed-order fractional derivative in viscoelastic rod solutions [20]. Refahi et al. presented DOFDEs to generalized the inertia and characteristics of polynomial concepts concerning the nonnegative density function [21]. Aminikhah et al. employed a combined Laplace transform and new Homotopy perturbation method for solving a particular class of the distributed order fractional Riccati equation [22]. For a time distributed order multi-dimensional diffusion-wave equation which is contained a forcing term, Atanacković et al. reinterpreted a Cauchy problem [23].

Katsikadelis proposed a method based on the finite-difference method to solve both linear and nonlinear DOFDEs numerically [24]. Dielthem and Ford proposed a method for DOFDEs of the general form $\int_0^m \mathcal{A}(r, D_*^r u(t)) dr = f(t)$ where $m \in \mathbb{R}^+$ and D_*^r is the fractional derivative of Caputo type of order r and introduced its analysis [25]. Zaky and Machado first derived the generalized necessary conditions for optimal control with dynamics described by DOFDEs and then proposed a practical numerical scheme for solving these equations [26]. Hadian-Rasanan et al. provided an artificial neural network framework for approximating various types of Lane-Emden equations such as fractional-order, and Lane-Emden equations system [27]. Razzaghi and Mashayekhi presented a numerical method to solve the DOFDEs based on hybrid function approximation [28]. Mashhoof and Refahi proposed methods based on the fractional-order integration's operational matrix with an initial value point for solving the DOFDEs [29]. Li et al. presented a high order numerical scheme for solving diverse DOFDEs by applying the reproducing kernel [30]. Gao and Sun derived two implicit difference schemes for two-dimensional DOFDEs [31].

In the first chapter, SVM has been fully and comprehensively discussed. Mehrkanoon et al. introduced a novel method based on LS-SVMs to solve ODEs [32]. Mehrkanoon and Suykens also presented another new approach for solving

delay differential equations [33]. Ye et al. presented an orthogonal Chebyshev kernel for SVMs [34]. Leake et al. compared the application of the theory of connections by employing LS-SVMs [35]. Baymani et al. developed a new technique by utilizing ϵ -LS-SVMs to achieve the solution of the ODEs in an analytical form [36]. Chu et al. presented an improved method for the numerical solution of LS-SVMs. They indicated that by using a reduced system of linear equations, the problem could be solved. They believed that their proposed approach is about twice as effective as the previous algorithms [37]. Pan et al. proposed an orthogonal Legendre kernel function for SVMs using the properties of kernel functions and comparing it to the previous kernels [38]. Ozer et al. introduced a new set of functions with the help of generalized Chebyshev polynomials, and they also increase the generalization capability of their previous work [39].

Lagaris et al. proposed a method based on artificial neural networks that can solve some categories of ODEs and PDEs and later compared their method to those obtained using the Galerkin finite element method [40]. Meade and Fernandez illustrated theoretically how a feedforward neural network could be constructed to approximate arbitrary linear ODEs [41]. They also indicated the way of directly constructing a feedforward neural network to approximate the nonlinear ordinary differential equations without training [42]. Dissanayake and Phan-Thien presented a numerical method for solving PDEs, which is based on neural-network-based functions [43]. To solve ODEs and elliptic PDEs, Mai-Duy and Tran-Cong proposed mesh-free procedures that rely on multiquadric radial basis function networks [44]. Effati and Pakdaman proposed a new algorithm by utilizing feedforward neural networks for solving fuzzy differential equations [45]. To solve the linear second kind integral equations of Volterra and Fredholm types, Golbabai and Seifollahi presented a novel method based on radial basis function networks that applied a neural network as the approximate solution of the integral equations [46]. Jianyu et al. illustrated a neural network to solve PDEs using the radial basis functions as the activation function of the hidden nodes [47].

The rest of the chapter is organized as follows. Some preliminaries are presented in Section 10.2, including the fractional derivatives and the numerical integration. We present the proposed method for simulating distributed-order fractional dynamics, and we depict LS-SVR details in Section 10.3. Numerical results are given in Section 10.4 to show the validity and efficiency of the proposed method. In Section 10.5, we draw concluding remarks.

10.2 Preliminaries

As we point earlier, modal Legendre functions are used in the proposed algorithm. We skip clarifying the properties and the procedure of making modal Legendre functions in this chapter as they have been discussed thoroughly in Chapter 4. Albeit, the fractional derivative focusing on the Caputo definition of the fractional derivative and the numerical integration are described in this section.

10.2.1 Fractional derivative

By considering $f(x)$ as a function, the Cauchy formula for n-th order can be obtained, and by generalizing the Cauchy formula for non-integer orders, we can achieve the Riemann-Liouville definition of fractional integral, and due to this, the well-known Gamma function is used as the factorial function for non-integer numbers. Therefore, by denoting the Riemann-Liouville definition of the fractional order of integral as ${}_a^{RL} I_x^\beta f(x)$, it can be defined as follows [50]:

$${}_a^{RL} I_x^\beta f(x) = \frac{1}{\Gamma(\beta)} \int_a^x (x-t)^{\beta-1} f(t) dt. \quad (10.5)$$

in which β is a real number which indicates the order of integral. Moreover, there is another definition for fractional derivative, namely, Caputo definition, and by denoting it as ${}_a^C D_x^\beta f(x)$, it can be defined as follows [50]:

$${}_a^C D_x^\beta f(x) = {}_a^{RL} I_x^{(k-\beta)} f^{(k)}(x) = \begin{cases} \frac{1}{\Gamma(k-\beta)} \int_a^x \frac{f^{(k)}(t) dt}{(x-t)^{\beta+1-k}} & \text{if } \beta \notin \mathbb{N} \\ \frac{d^\beta}{dx^\beta} & \text{if } \beta \in \mathbb{N} \end{cases}, \quad (10.6)$$

It is worth to mention that Eqs. (10.7) and (10.8) are the most useful properties of the Caputo derivative [50]

$${}_0^C D_x^\alpha x^\gamma = \begin{cases} \frac{\Gamma(\gamma+1)}{\Gamma(\gamma-\alpha+1)} x^{\gamma-\alpha}, & 0 \leq \alpha \leq \gamma \\ 0, & \alpha > \gamma \end{cases}, \quad (10.7)$$

and since the Caputo derivative is a linear operator [50], we have

$${}_a^C D_x^\alpha (\lambda f(x) + \mu g(x)) = \lambda {}_a^C D_x^\alpha f(x) + \mu {}_a^C D_x^\alpha g(x) \quad \lambda, \mu \in \mathbb{R}. \quad (10.8)$$

10.2.2 Numerical integration

In numerical analysis, quadrature rule is an approximation of the definite integral of a function, and it usually stated as a weighted sum of function values at specified points within the considered domain of integration. An n-point Gaussian quadrature rule, named after Carl Friedrich Gauss, the German mathematician, is a quadrature rule constructed to yield an exact result for polynomials of degree $2n - 1$ or less, by a suitable choice of the nodes x_i and weights $\omega_i = \int_a^b h_i(x) \omega(x) dx$ for $i = 1, 2, \dots, n$. The most common domain of integration for such a rule is taken as $[-1, 1]$, but we assume the interval $[a, b]$ for the integral boundary so the rule is stated as

$$\int_a^b f(x) dx \approx \sum_{i=1}^n \omega_i f(x_i) \quad (10.9)$$

Now we consider the following theorem

Theorem 10.1 [51] Suppose x_0, x_1, \dots, x_n as the roots of the orthogonal polynomial p_{N+1} of order $N + 1$. Then there exist a unique set of quadrature weights $\omega_0, \omega_1, \dots, \omega_N$ defined as $\omega_j = \int_a^b h_j(x)\omega(x) dx$

$$\int_a^b \omega(x)p(x) dx \approx \sum_{i=1}^n \omega_i p(x_i) \quad \forall p \in P_{2N+1} \quad (10.10)$$

In which the weights are positive and calculate as following

$$\omega_i = \frac{k_{n+1}}{k_n} \frac{\|f_n\|_\omega^2}{f_n(x_i)f'_{n+1}(x_i)}, \quad 0 \leq i \leq n \quad (10.11)$$

which k_i s are the leading coefficients of f_i

In the next section, the mentioned method is proposed by considering the Least squares support vector regression definition.

10.3 LS-SVR method for solving distributed order fractional differential equations

In this section, first, the LS-SVR is introduced, then some of its characteristics are discussed, then the considered method based on LS-SVR, which helps solve the DOFDEs, is included. Having the concepts of LS-SVM, we present our solution based on the LS-SVM regression in this section. Due to this, consider the following equations

$$\int_a^b G_1(p, D^p u(t)) dp + G_2(t, u(t), D^{\alpha_i} u(t)) = F(t), \quad t \in [0, \eta], \quad (10.12)$$

with the following initial conditions:

$$u^{(k)}(0) = 0, \quad (10.13)$$

where $k = 0, 1, \dots, \lfloor \max\{b, \alpha_i\} \rfloor$. Now, we discuss the convergence, consistency of method analysis in a linear situation:

$$\int_a^b g(p)D^p u(t) dp + A(u(t)) = F(t), \quad (10.14)$$

since A is a linear operator, we can rewrite the Eq. 10.14 as

$$L(u(t)) = F(t), \quad (10.15)$$

now we consider the LS-SVR for solving the Eq. 10.14:

$$u(x) \approx u_N(x) = \omega^T \phi(x), \quad (10.16)$$

which $\omega = [\omega_0, \dots, \omega_N]^T$ are the weight coefficients and $\phi = [\phi_0, \dots, \phi_N]^T$ are the basis functions. To determine the unknown coefficients, we consider the following optimization problem

$$\min_{\omega, \varepsilon} \frac{1}{2} \omega^T \omega + \frac{\gamma}{2} \varepsilon^T \varepsilon, \quad (10.17)$$

such that for every $i = 0, \dots, N$

$$\int_a^b g(p) D^p(u(t_i)) dp + Au(t_i) - F(t_i) = \varepsilon_i \quad (10.18)$$

hence we have

$$Lu(t_i) - F(t_i) = \varepsilon_i \quad (10.19)$$

consider the Lagrangian $L(u) = \sum_{j=0}^N \omega_j l_j$ we have

$$\mathcal{L} = \frac{1}{2} \omega^T \omega + \frac{\gamma}{2} \varepsilon^T \varepsilon - \sum_{i=0}^N \lambda_i (L(u(t_i)) - F(t_i) - \varepsilon_i), \quad (10.20)$$

and we also have

$$\mathcal{L} = \frac{1}{2} \sum_{j=0}^N \omega_j^2 + \frac{\gamma}{2} \sum_{i=0}^N \varepsilon_i^2 - \sum_{i=0}^N \lambda_i \left(\sum_{j=0}^N \omega_j L(\phi_j(t_i)) - F_i - \varepsilon_i \right), \quad (10.21)$$

we can calculate the extremums of Eq. 10.21 as follows

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial \omega_k} = 0 & \omega_k = \sum_{i=0}^N \lambda_i \underbrace{(L(\phi_k(t_i)))}_{S_{ki}} = 0 \quad k = 0, 1, \dots, N \\ \frac{\partial \mathcal{L}}{\partial \varepsilon_k} = 0 & 2\gamma \varepsilon_k + \lambda_k = 0 \quad k = 0, 1, \dots, N \\ \frac{\partial \mathcal{L}}{\partial \lambda_k} = 0 & \sum_{j=0}^N \omega_j (L(\phi_j(t_k))) = F_k - \varepsilon_k \quad k = 0, 1, \dots, N \end{cases}, \quad (10.22)$$

and it can be summarized in a vector form as follows

$$\begin{cases} \omega - S\lambda = 0 & (\text{I}) \\ 2\gamma \varepsilon = -\lambda ; \varepsilon = -\frac{1}{2\gamma} \lambda & (\text{II}) \\ S^T \omega - f - \varepsilon = 0 & (\text{III}) \end{cases}. \quad (10.23)$$

Matrix S is as below

$$S_{ij} = L(\phi_i(t_j)) = \int_a^b g(p) D^P \phi_i(t_j) dp + A\phi_i(t_j), \quad (10.24)$$

and by considering this equation, $M_{ij} := D^P \phi_i(t_j)$ can be defined. We use the Gaussian numerical integration to calculate $\int_a^b g(p) M_{ij}^{(p)} dp$.

There exist two ways to determine numerical integration

$$\int_a^b f(x) dx \approx \sum_{i=1}^N \omega_i f(x_i) \quad (10.25)$$

Ordinarily, we can apply the Newton-Cotes method by considering each x_i as a fixed point and finding ω_i s. Alternatively, we can optimize the x_i s and ω_i s. The x_i s are the Legendre polynomials roots relocated to interval $[a, b]$. We notice calculating $Au(t_i)$ as A is a differential operator (which can be fractional) is explained. With the help of Eq. 10.23-(II) and substituting Eq. 10.23-(I) in Eq. 10.23-(III) we have

$$S^T S \lambda + \frac{1}{2\gamma} \lambda = f, \quad (10.26)$$

By defining $A := S^T S + \frac{1}{2\gamma} I$ the following equation is achieved

$$A\alpha = f. \quad (10.27)$$

Remark 10.1 The Matrix A is positive definite if and only if for all vectors $x \in \mathbb{R}^{n+1}$ we have $x^T Ax \geq 0$. So for every vector x , we can address

$$x^T (S^T S + \frac{1}{2\gamma} I) x = x^T S^T S x + \frac{1}{\gamma} x^T x, \quad (10.28)$$

as $\gamma > 0$ the matrix A is positive definite.

Theorem 10.2 *In linear system Eq. 10.23 there is a unique solution.*

Proof By considering the Remark 10.1, the theorem is proved. \square

Since the system Eq. 10.27 is positive definite and the sparseness of A with solving Eq. 10.27, we can reach α , and with the help of criteria (I) of Eq. 10.23, we can calculate ω .

In the nonlinear form, applying the Quasi-Linearization Method (QLM) can be converted to a sequence of linear equations and solve them independently.

In the next section, some numerical examples are provided for indicating the efficiency and accuracy of the presented method, and the convergence of the proposed approach is represented.

10.4 Numerical results and discussion

In this section, various numerical examples are presented to express the mentioned method's accuracy and efficiency, here three linear and three nonlinear examples and the comparison our results with other related works. Moreover, the interval $\Omega = [0, T]$ is considered as the intended domain. Additionally, the utilized norm for comparing the exact solution and the approximated one, which is denoted by $\|e\|_2$ is defined as follows:

$$\|e\|_2 = \left(\int_0^T (u(t) - u_{app}(t))^2 dt \right)^{\frac{1}{2}}, \quad (10.29)$$

in which $u(t)$ is the exact solution and the $u_{app}(t)$ is the approximated solution. All numerical experiments are computed in Maple software on a 3.5 GHz Intel Core i5 CPU machine with 8 GB of RAM.

10.4.1 Test problem 1

As the first example, consider the following equation [48]

$$\int_{0.2}^{1.5} \Gamma(3-p) D^p u(t) dp = 2 \frac{t^{1.8} - t^{0.5}}{\ln(t)}, \quad (10.30)$$

with the following initial condition

$$u(0) = u'(0) = 0. \quad (10.31)$$

in which $u(t) = t^2$ is the exact solution. Then by applying the proposed method, the following equation will be concluded: Fig. 10.1 indicates the errors, for example, 10.4.1 for three different aspects, which are the number of Gaussian points, Gamma, and the number of basis functions.

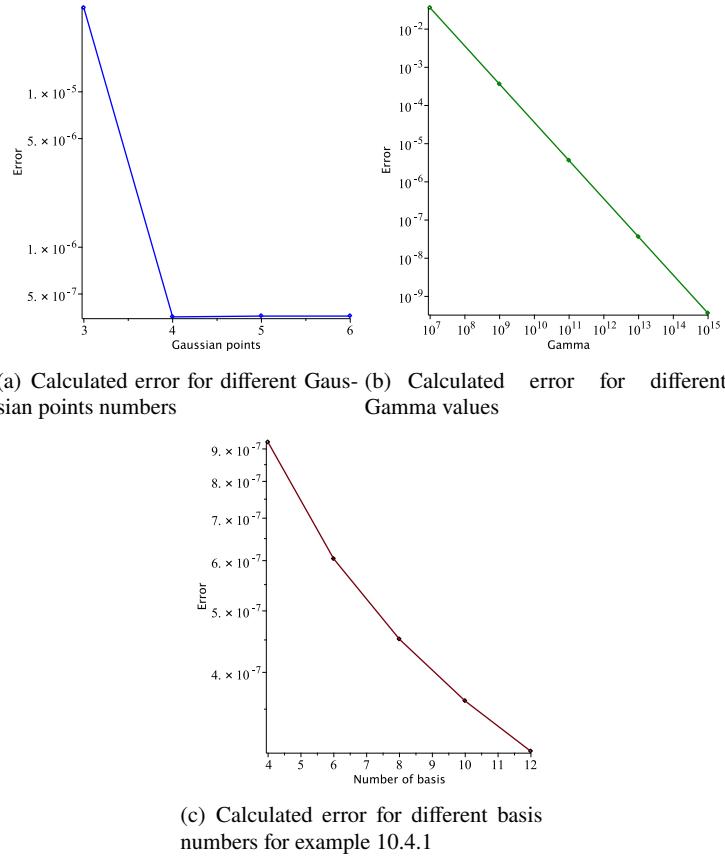


Fig. 10.1: Calculated error for three different aspects for example 10.4.1

By looking at the Fig. 1(a), it can be concluded that by increasing the Gaussian points, the error is converging to zero; by a glance at the Fig. 1(b), it is obvious the error is decreasing exponentially, and Fig. 1(c), it seems that increasing the number of bases, caused error reducing, constantly.

10.4.2 Test problem 2

Suppose the following nonlinear equation [49]

$$\int_0^1 \Gamma(5-p) D^p u(t) dp = \sin(u(t)) + 24 \frac{t^4 - t^3}{\ln(t)} - \sin(t^4), \quad (10.32)$$

with the initial condition as below

$$u(0) = 0. \quad (10.33)$$

which $u(t) = t^4$ is the exact solution. Now, look at the following figure, which demonstrates the error behavior in the different number of Gaussian points, amount of Gamma, and the number of basis.

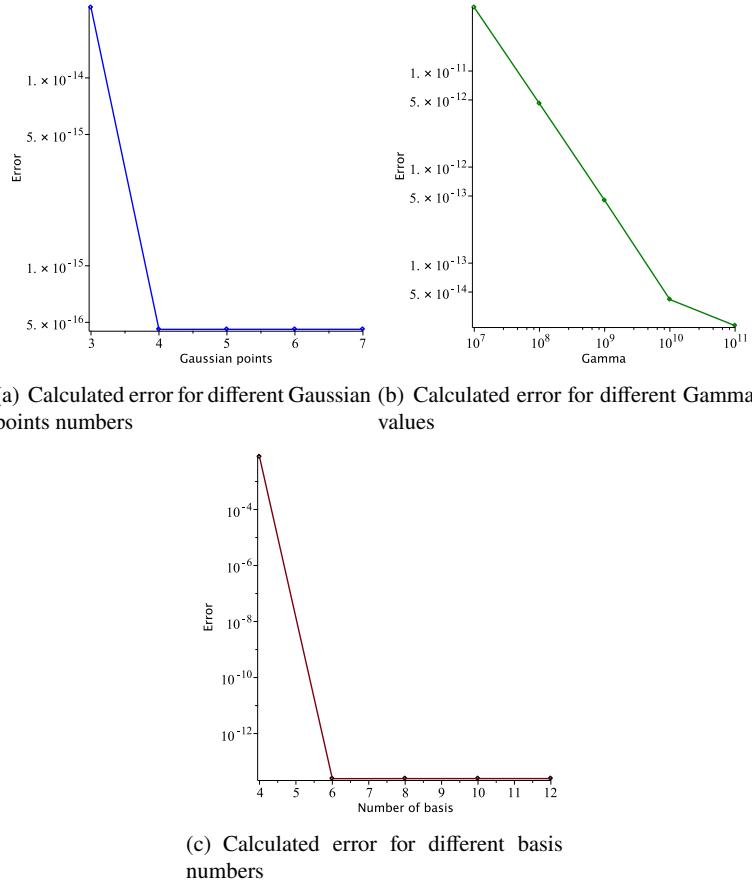


Fig. 10.2: Calculated error for three different aspects for example 10.4.2

By considering the Fig. 10.2, Fig. 2(a) is shown that by increasing the Gaussian points, the error converges to zero more. Fig. 2(b) shows that the error reduced quite exponentially, and by looking at the Fig. 2(c) indicates that decreasing error is due to increasing the bases number.

This example has also been solved by Xu in [49]. Table 10.1 is a comparison between the method as mentioned earlier and the one that Xu proposed. It can be concluded that for different Gaussian numbers, the proposed method is achieving better results.

Table 10.1: The table of the example 10.4.2 in comparison with Xu method [49]

| | M Method of [49] with $N = 4$ | Presented method with $N = 4$ |
|---|-------------------------------|-------------------------------|
| 2 | 2.4768E-008 | 6.9035E-010 |
| 3 | 9.0026E-011 | 2.3935E-014 |
| 4 | 2.3632E-013 | 8.4114E-018 |
| 5 | 2.7741E-015 | 3.0926E-023 |

10.4.3 Test problem 3

Suppose the following example [49]

$$\int_0^1 \Gamma(7-p) D^p u(t) dp = -u^3(t) - u(t) + 720 \frac{t^5(t-1)}{\ln(t)} - t^6, \quad (10.34)$$

by considering the following initial condition

$$u(0) = 0. \quad (10.35)$$

$u(t) = t^6$ is the exact solution of this example. Now consider Fig. 10.3 which is displaying error in different aspect.

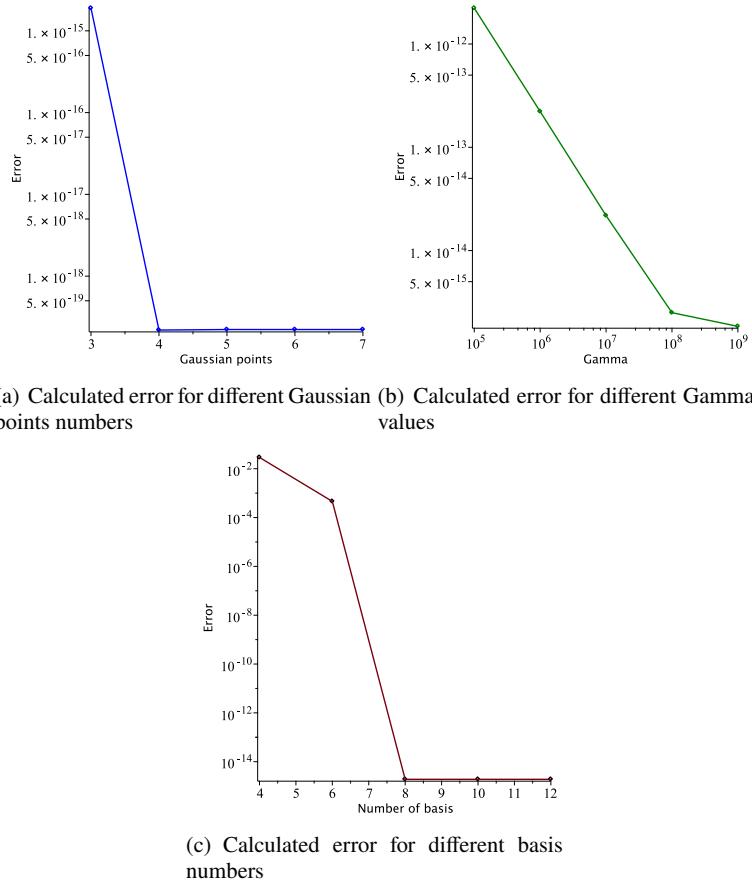


Fig. 10.3: Calculated error for three different aspects for example 10.4.3

By looking at Fig. 3(a), it is clear that as the Gaussian points are increasing, the error is decreasing. Same in Fig. 3(b) and 3(c) when Gamma and number of bases are decreasing, respectively, the error decrease too.

In Table 10.2, the results of the proposed technique and the method presented in [49] are compared. By a glance at the table, it is clear that the suggested method is working way better than Xu's method.

Table 10.2: The table of the comparison of our results for example 10.4.3 and Xu method [49]

| M | Method in [49] with $N = 7$ | Proposed method with $N = 7$ | Method of [49] with $N = 9$ | Presented method with $N = 9$ |
|---|-----------------------------|------------------------------|-----------------------------|-------------------------------|
| 2 | 4.3849E-009 | 1.1538E-010 | 1.3008E-008 | 1.1550E-010 |
| 3 | 1.5375E-011 | 1.8923E-015 | 3.8919E-011 | 1.8932E-015 |
| 4 | 3.7841E-014 | 3.0791E-019 | 8.0040E-014 | 2.4229E-019 |
| 5 | 3.6915E-016 | 3.1181E-019 | 1.2812E-015 | 2.4578E-019 |

10.4.4 Test problem 4

Suppose an example as follows [49]:

$$\int_0^1 e^p D^p u(t) dp = -u^3(t) - u(t) + \Gamma(7.1) \frac{t^{2.1}(e-1)}{\ln(t)} + t^{9.3} + t^{3.1}, \quad (10.36)$$

with considering the following initial condition

$$u(0) = 0. \quad (10.37)$$

The exact solution is $u(t) = t^{3.1}$. Now consider the following figure, which presents that the error is converging to zero.

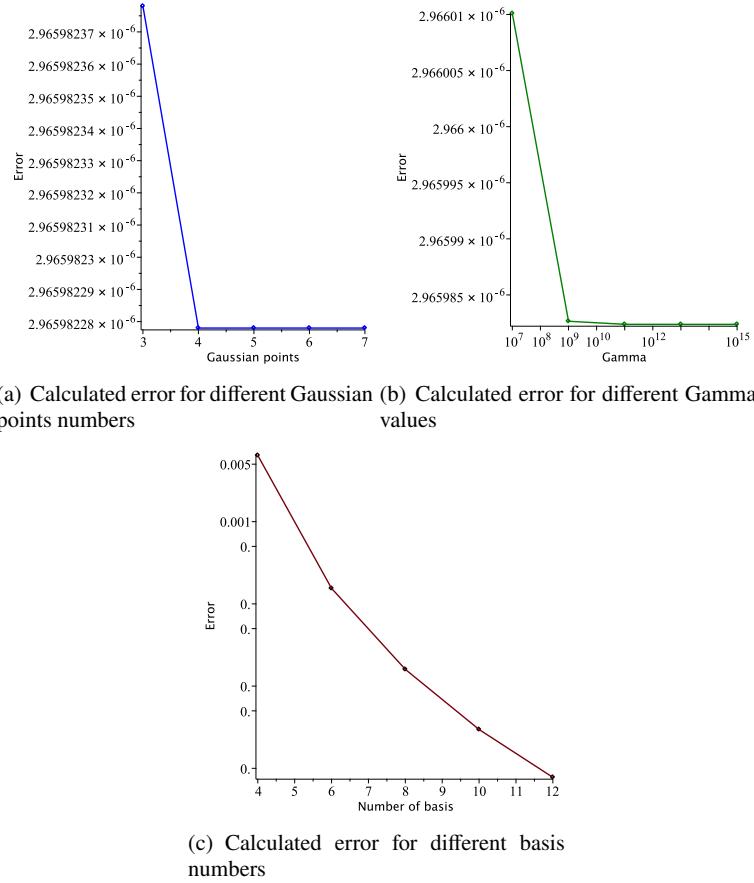


Fig. 10.4: Calculated error for three different aspects for example 10.4.4

Fig. 10.4, Fig. 4(a), 4(b) and 4(c) confirms that if the Gaussian points, Gamma and number of basis increases, respectively, then the error will become convergence to zero. Table 10.3 shows the results obtained in [49] and the result obtained by using the presented method.

Table 10.3: The table of the comparison of our results for example 10.4.4 and Xu method [49]

| M | Method in [49] with $N = 6$ | Proposed method with $N = 6$ | Method of [49] with $N = 14$ | Presented method with $N = 14$ |
|---|-----------------------------|------------------------------|------------------------------|--------------------------------|
| 2 | 4.4119E-005 | 1.5437E-005 | 2.4317E-006 | 2.5431E-007 |
| 3 | 4.3042E-005 | 1.5435E-005 | 2.4984E-007 | 2.5221E-007 |
| 4 | 4.3034E-005 | 1.5435E-005 | 2.4195E-007 | 2.5221E-007 |
| 5 | 4.3034E-005 | 1.5435E-005 | 2.4190E-007 | 2.5221E-007 |

10.4.5 Test problem 5

For the final example consider the following equation [49]

$$\frac{1}{120} \int_0^2 \Gamma(6-p) D^p u(t) dp = \Gamma(7.1) \frac{t^5 - t^3}{\ln(t)}, \quad (10.38)$$

with the following initial condition

$$u(0) = 0. \quad (10.39)$$

Where $u(t) = t^5$ is the exact solution. Now suppose Fig. 10.5, which is showing the behavior of error by considering different aspects.

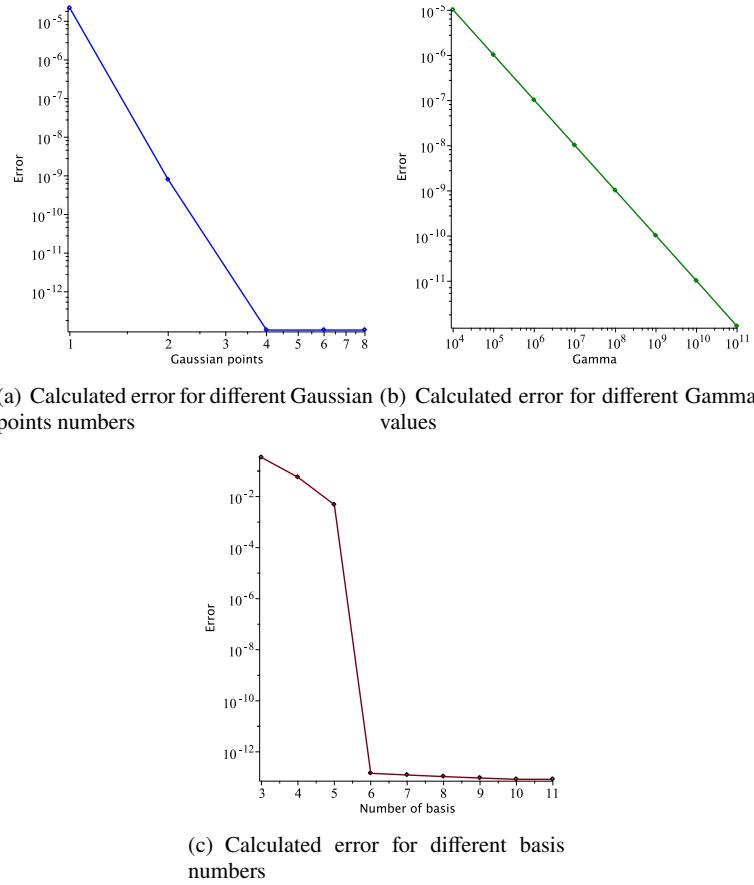


Fig. 10.5: Calculated error for three different aspects for example 10.4.5

By looking at Fig. 5(a) and 5(c), it can be concluded that the error which proposed in Eq. 10.29 is convergence to zero and in Fig. 5(b) in it converging to zero exponentially and that means the approximate results are converging to the exact solution.

Table 10.4 indicating the comparison between our proposed method and the method proposed in [49]. By studying this table, it can be concluded that besides $N = 5$, our method is achieving results more accurately.

Table 10.4: The table of the comparison of our results for example 10.4.5 and Xu method [49]

| N | Method in [49] with $M = 4$ | Proposed method with $M = 4$ |
|---|-----------------------------|------------------------------|
| 5 | 4.2915E-010 | 4.8113E-002 |
| 6 | 3.3273E-010 | 1.6388E-013 |
| 7 | 1.9068E-010 | 1.4267E-013 |
| 8 | 9.4633E-011 | 1.2609E-013 |
| 9 | 6.7523E-011 | 1.1283E-013 |

10.5 Conclusion

In this chapter, the LS-SVR algorithm was utilized for solving the Distributed order fractional differential equations, which is based on utilizing modal Legendre as the basis function. The mentioned algorithm has better accuracy in comparison with other proposed method for DOFDEs. On the other hand, the uniqueness of the solution is provided. One of the useful parameters in solving this kind of equations and algorithm's accuracy is the parameter Gamma and how the accuracy changes indicated in numerical examples. Moreover, it has to be considered that the Gamma value can not be increased too much because it might become more than machine accuracy. Even though all of the computations are done with MAPLE, which is a symbolic application, this point is considered. For indicating the applicability of the proposed algorithm, five examples are proposed in which the obtained accuracy of these examples are compared with other methods.

References

1. Hilfer, R.: Applications of fractional calculus in physics. World scientific, (2000)
2. Cao, K. C., Zeng, C., Chen, Y., Yue, D.: Fractional Decision Making Model for Crowds of Pedestrians in Two-Alternative Choice Evacuation. IFAC-PapersOnLine **50**, 11764–11769 (2017)
3. Hadian-Rasanan, A. H., Rad, J. A., Sewell. D. K.: Are there jumps in evidence accumulation, and what, if anything, do they reflect psychologically? An analysis of Lévy-Flights models of decision-making. PsyArXiv (2021), doi: 10.31234/osf.io/vy2mh
4. Datsko, B., Gafiychuk, V., Podlubny, I.: Solitary travelling auto-waves in fractional reaction-diffusion systems. Commun. Nonlinear Sci. Numer. Simul. **23**, 378–387 (2015)
5. Caputo, M.: Linear models of dissipation whose Q is almost frequency independent—II. Geophys. J. Int. **13**, 529–539 (1967)
6. Rad, J. A., Kazem, S., Shaban, M., Parand, K., Yildirim, A.: Numerical solution of fractional differential equations with a Tau method based on Legendre and Bernstein polynomials. Mathematical Methods in the Applied Sciences **37**, 329–342 (2014)

7. Podlubny, I.: Fractional differential equations: an introduction to fractional derivatives, fractional differential equations, to methods of their solution and some of their applications. Elsevier, (1998)
8. Atangana, A., Gómez-Aguilar, J. F.: A new derivative with normal distribution kernel: Theory, methods and applications. *Phys. A: Stat. Mech. Appl.* **476**, 1–14 (2017)
9. Heydari, M. H., Atangana, A., Avazzadeh, Z., Mahmoudi, M. R.: An operational matrix method for nonlinear variable-order time fractional reaction-diffusion equation involving Mittag-Leffler kernel. *Eur. Phys. J. Plus* **135**, 1–19 (2020)
10. Bagley, R. L., Torvik, P. J.: Fractional calculus in the transient analysis of viscoelastically damped structures. *AIAA J.* **23**, 918–925 (1985)
11. Umarov, S., Gorenflo, R.: Cauchy and nonlocal multi-point problems for distributed order pseudo-differential equations: Part one. *J. Anal. Appl.* **245**, 449–466 (2005)
12. Carpinteri, A., Mainardi, F.: Fractals and fractional calculus in continuum mechanics. Springer, (2014)
13. Rossikhin, Y. A., Shitikova, M. V.: Applications of fractional calculus to dynamic problems of linear and nonlinear hereditary mechanics of solids, *Appl Mech Rev* **50**, 15–67 (1997)
14. Hartley, T. T.: Fractional system identification: an approach using continuous order-distributions. NASA Glenn Research Center, (1999)
15. Caputo, M.: Diffusion with space memory modelled with distributed order space fractional differential equations. *Ann. Geophys.* **46**, 223–234 (2003)
16. Sokolov, I. M., Chechkin, A. V., Klafter, J.: Distributed-order fractional kinetics. arXiv preprint cond-mat/0401146, (2004)
17. Parodi, M., Gómez, J. C.: Legendre polynomials based feature extraction for online signature verification. Consistency analysis of feature combinations. *Pattern Recognit.* **47**, 128–140 (2014)
18. Najafi, H. S., Sheikhani, A. R., Ansari, A.: Stability analysis of distributed order fractional differential equations. In *Abstract and Applied Analysis* **2011**, 175323 (2011)
19. Atanacković, T. M., Oparnica, L., Pilipović, S.: On a nonlinear distributed order fractional differential equation. *J. Math. Anal.* **328**, 590–608 (2007)
20. Atanackovic, T. M., Budincevic, M., Pilipovic, S.: On a fractional distributed-order oscillator. *Journal of Physics A: Mathematical and General*, **38**, 6703 (2005)
21. Refahi, A., Ansari, A., Najafi, H. S., Merhdoust, F.: Analytic study on linear systems of distributed order fractional differential equations. *Matematiche* **67**, 3–13 (2012)
22. Aminikhah, H., Sheikhani, A. H. R., Rezazadeh, H.: Approximate analytical solutions of distributed order fractional Riccati differential equation. *Ain Shams Eng. J.* **9**, 581–588 (2018)
23. Atanackovic, T. M., Pilipovic, S., Zorica, D.: Time distributed-order diffusion-wave equation. II. Applications of Laplace and Fourier transformations. *Proc. R. Soc. A: Math. Phys. Eng. Sci.* **465**, 1893–1917 (2009)
24. Katsikadelis, J. T.: Numerical solution of distributed order fractional differential equations. *J. Comput. Phys.* **259**, 11–22 (2014)
25. Diethelm, K., Ford, N. J.: Numerical analysis for distributed-order differential equations. *J. Comput. Appl. Math.* **225**, 96–104 (2009)
26. Zaky, M. A., Machado, J. T.: On the formulation and numerical simulation of distributed-order fractional optimal control problems. *Commun. Nonlinear Sci. Numer. Simul.* **52**, 177–189 (2017)
27. Hadian-Rasanan, A. H., Rahmati, D., Gorgin, S., Parand, K.: A single layer fractional orthogonal neural network for solving various types of Lane–Emden equation. *New Astron.* **75**, 101307 (2020)
28. Mashayekhi, S., Razzaghi, M.: Numerical solution of distributed order fractional differential equations by hybrid functions. *Journal of Computational Physics* **315**, 169–181 (2016)
29. Mashhoof, M., Sheikhani, A. R.: Simulating the solution of the distributed order fractional differential equations by block-pulse wavelets. *UPB Sci. Bull., Ser. A: Appl. Math. Phys.* **79**, 193–206 (2017)
30. Li, X., Li, H., Wu, B.: A new numerical method for variable order fractional functional differential equations. *Appl. Math. Lett.* **68**, 80–86 (2017)

31. Gao, G. H., Sun, Z. Z.: Two alternating direction implicit difference schemes for two-dimensional distributed-order fractional diffusion equations. *J. Sci. Comput.* **66**, 1281–1312 (2016)
32. Mehrkanoon, S., Falck, T., Suykens, J. A.: Approximate solutions to ordinary differential equations using least squares support vector machines. *IEEE Trans. Neural Netw. Learn. Syst.* **23**, 1356–1367 (2012)
33. Mehrkanoon, S., Suykens, J. A.: LS-SVM based solution for delay differential equations. In *Journal of Physics: Conference Series* **410**, 012041 (2013)
34. Ye, N., Sun, R., Liu, Y., Cao, L.: Support vector machine with orthogonal Chebyshev kernel. In *18th International Conference on Pattern Recognition (ICPR'06)* **2**, 752–755 (2006)
35. Leake, C., Johnston, H., Smith, L., Mortari, D.: Analytically embedding differential equation constraints into least squares support vector machines using the theory of functional connections. *Mach. learn. knowl. extr.* **1**, 1058–1083 (2019)
36. Baymani, M., Teymoori, O., Razavi, S. G.: Method for solving differential equations. *Am J Comput Sci Inf Eng* **3**, 1–6 (2016)
37. Chu, W., Ong, C. J., Keerthi, S. S.: An improved conjugate gradient scheme to the solution of least squares SVM. *IEEE Trans Neural Netw* **16**, 498–501 (2005)
38. Pan, Z. B., Chen, H., You, X. H.: Support vector machine with orthogonal Legendre kernel. In *2012 International Conference on Wavelet Analysis and Pattern Recognition*, 125–130 (2012)
39. Ozer, S., Chen, C. H., Cirpan, H. A.: A set of new Chebyshev kernel functions for support vector machine pattern classification. *Pattern Recognit.* **44**, 1435–1447 (2011)
40. Lagaris, I. E., Likas, A., Fotiadis, D. I.: Artificial neural networks for solving ordinary and partial differential equations. *IEEE Trans Neural Netw Learn Syst* **9**, 987–1000 (1998)
41. Meade Jr, A. J., Fernandez, A. A.: The numerical solution of linear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling* **19**, 1–25 (1994)
42. Meade Jr, A. J., Fernandez, A. A.: Solution of nonlinear ordinary differential equations by feedforward neural networks. *Mathematical and Computer Modelling* **20**, 19–44 (1994)
43. Dissanayake, M. W. M. G., Phan-Thien, N.: Neural-network-based approximations for solving partial differential equations. *Commun. Numer. Methods Eng.* **10**, 195–201 (1994)
44. Mai-Duy, N., Tran-Cong, T.: Numerical solution of differential equations using multiquadric radial basis function networks. *Neural Netw* **14**, 185–199 (2001)
45. Effati, S., Pakdaman, M.: Artificial neural network approach for solving fuzzy differential equations. *Inf. Sci.* **180**, 1434–1457 (2010)
46. Golbabai, A., Seifollahi, S.: Numerical solution of the second kind integral equations using radial basis function networks. *Appl. Math. Comput.* **174**, 877–883 (2006)
47. Jianyu, L., Siwei, L., Yingjian, Q., Yaping, H.: Numerical solution of elliptic partial differential equation using radial basis function neural networks. *Neural Netw* **16**, 729–734 (2003)
48. Yuttanan, B., Razzaghi, M.: Legendre wavelets approach for numerical solutions of distributed order fractional differential equations. *Appl. Math. Model.* **70**, 350–364 (2019)
49. Xu, Y., Zhang, Y., Zhao, J.: Error analysis of the Legendre-Gauss collocation methods for the nonlinear distributed-order fractional differential equation. *Appl. Numer. Math.* **142**, 122–138 (2019)
50. Hadian Rasanan, A. H., Bajalan, N., Parand, K., Rad, J. A.: Simulation of nonlinear fractional dynamics arising in the modeling of cognitive decision making using a new fractional neural network. *Math. Methods Appl. Sci.* **43**, 1437–1466 (2020)
51. Mastroianni, G., Milovanovic, G.: *Interpolation processes: Basic theory and applications*. Springer Science & Business Media, Berlin (2008)
52. Ding, W., Patnaik, S., Sidhardh, S., Semperlotti, F.: Applications of distributed-order fractional operators: a review. *Entropy* **23**, 110 (2021)

Part IV

Orthogonal kernels in action

Chapter 11

GPU Acceleration of LS-SVM, Based on Fractional Orthogonal Functions

Armin Ahmadzadeh, Mohsen Asghari, Dara Rahmati, Saeid Gorgin, and Behzad Salami

Abstract SVM classifiers are employed widely in classification problems. However, the computation complexity prevents it from becoming applicable without acceleration approaches. General-Purpose Computing using Graphics Processing Units (GPGPU) is one of the most used techniques for accelerating array-based operations. In this chapter, a method for accelerating the SVM with the kernel of fractional orthogonal functions applied on GPU devices is introduced. The experimental result for the first kind of Chebyshev function, used as an SVM kernel in this book, resulting in a 2.2X speedup compared to GPU acceleration with CPU execution. In the fit function and training part of the code, a 58X speedup on the Google Colab GPU devices is obtained. This chapter proposes more details of GPU architecture and how it can be used as a co-processor along with the CPU to accelerate the SVM classifiers.

Mohsen Assghari

School of Computer Science, Institute for Research in Fundamental Sciences, Farmanieh Campus, Tehran, Iran e-mail: m.asgari@ipm.ir

Armin Ahmadzadeh

School of Computer Science, Institute for Research in Fundamental Sciences, Farmanieh Campus, Tehran, Iran e-mail: a.ahmadzadeh@ipm.ir

Dara Rahmati

Computer Science and Engineering (CSE) Department, Shahid Beheshti University, Tehran, Iran e-mail: d Rahmati@sbu.ac.ir

Saeid Gorgin

Department of Electrical Engineering and Information Technology, Iranian Research Organization for Science and Technology (IROST), Tehran, Iran e-mail: gorgin@irost.ir

Behzad Salami

Barcelona Supercomputing Center (BSC) e-mail: behzad.salami@bsc.es

11.1 Parallel processing

Nowadays, computers and their benefits have revolutionized human life. Due to the vast application usage, the need for accuracy, and multi-feature application demand has made them more complex. These complexities are due to data accessibility, and processing complexity. Chip vendors always try to produce memories with less latency and broader bandwidth to overcome data accessibility. Besides, the parallel processing approach overcomes the processing complexity [5]. Parallel processing is an approach to divide large and complex tasks into several small tasks. This approach allows executing the processing parts simultaneously. GPUs are special-purpose processors mainly designed for graphical workloads. Using GPUs for general-purpose computing enlightened the way for the emergence of a new era in high-performance computing [1, 2], including accelerators for cryptosystems [2, 3, 4], multimedia compression standards [6], scientific computing [8, 9], machine learning and clustering accelerators [7], simulation of molecular dynamics [10], and quantum computing simulation[8]. The designers are gradually making improvements in GPU architecture to accelerate it, resulting in real-time processing in many applications. Many smaller computational units handle simple parallel tasks within a GPU that traditionally CPUs were supposed to hold. This usage of GPUs instead of CPUs is called General-Purpose Computing on Graphics Processing Unit (GPGPU). One admiring improvement of the GPGPUs is the release of CUDA (Compute Unified Device Architecture), NVIDIA's parallel computing platform and programming model. CUDA was introduced in 2007, allowing many researchers and scientists to deploy their compute-intensive tasks on GPUs. CUDA provides libraries and APIs that could be used inside multiple programming languages such as C/C++ and Python, general-purpose applications such as multimedia compression, cryptosystems, machine learning, etc. Moreover, they are generally faster than CPUs, performing more instructions in a given time. Therefore, together with CPUs, GPUs provide heterogeneous and scalable computing, acting as co-processors while reducing the CPU's workload.

The rest of this chapter is organized as follows. In section two, the NVIDIA GPU architecture and how it helps us as an accelerator is presented. PyCUDA is described also in section two. In section three, the proposed programming model is analyzed before acceleration. Section four defines the hardware and software requirements for implementing our GPU acceleration platform. The method of accelerating the Chebyshev kernel is described in section five in detail. In section six, we propose critical optimizations that should be considered for more speedup. Then, we recommend a Quadratic Problem Solver (QPS) based on GPU in section seven for more speedup in our platform. This chapter is concluded in section eight.

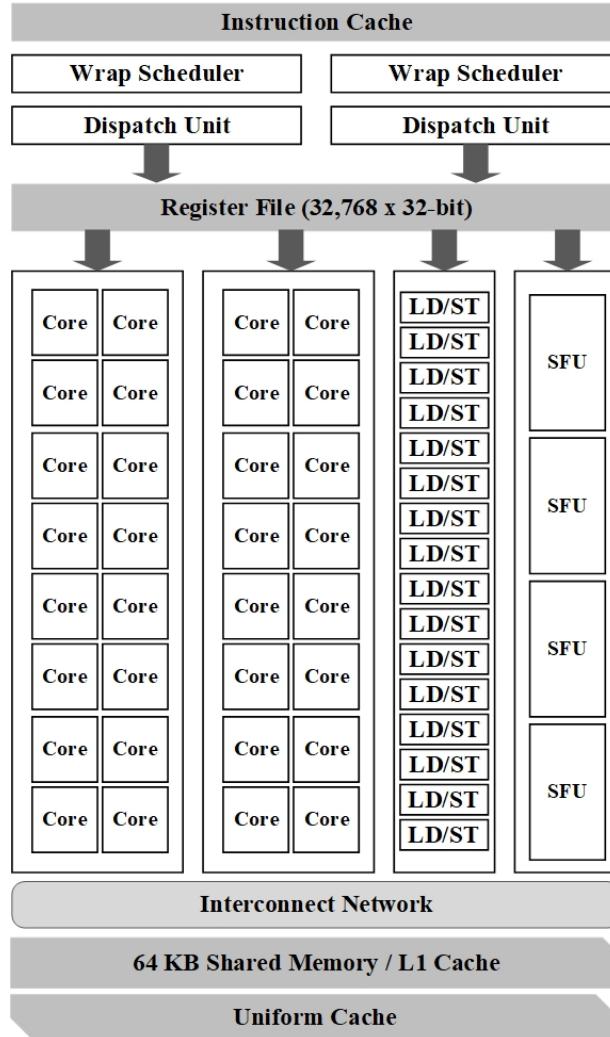


Fig. 11.1: GPU Architecture (CUDA Cores, Shared Memory, Special Function Units, WARP Scheduler, and Register Files) [12]

11.2 GPU Architecture

The GPUs are specifically well suited for three-dimensional graphical processing. They are massively parallel processing units that have their usage in general purpose applications. Because of the power of parallelism, and the flexible programming model of modern GPUs, GPGPU computation has become an attractive platform for both research and industrial applications. As some examples, developers can use

GPUs for physics-based simulation, linear algebra operations, Fast Fourier Transform (FFT) and reconstruction, flow visualization, database operations, etc, to achieve high performance, while having accurate results.

A GPU consists of many streaming multi-processors (SM) modules, and each SM contains several processors that are referred to as CUDA cores in NVIDIA products (Fig. 11.1) [12]. These multi-processors are utilized by scheduling thousands of threads in thread blocks and dispatching the blocks onto SMs. The latency of data load/store operation is high in GPUs. However, unlike CPUs that have low latency in data loading/storing and lower processing throughput, they are high throughput processors. Threads of a block are arranged as WARPs to be scheduled for execution. Every 32 consecutive threads make a WARP, and all threads of a WARP are executed simultaneously in the SM as a Single Instruction Multiple Thread (SIMT) operation. In a SIMT operation, all threads of a WARP execute the same instruction, and any thread makes the output of the operation on its private data. By using WARP scheduling and switching between stalling WARPs and ready-to-execute WARPs, GPUs can hide the memory latency. In GPUs, the codes are executed in the form of Kernels. Kernels are blocks of codes that are separated from the main code and are called from the CPU (known as a host) and executed on GPU (known as device) (Fig. 11.2) [12].

The kernel code should fully exploit the computational task's data parallelism in order to benefit fruitfully from the GPU's massively parallel architecture. A group of threads that can be launched together is called a thread block. There are different CUDA memory types in a GPU. The most important memory types include global memory, shared memory, and register memory [13]. Global memory is the slowest of the three types, but is accessible to all of the threads and has the largest capacity. Shared memory is accessible to all of the threads that share the same thread block, and register memory is accessible only to a single thread, but it is the fastest of the three memory types.

The host can read (write) from (to) GPU's global memory shown in Fig. 11.3. These accesses are made possible by the CUDA API. The global memory and constant memory are the only memories in which their content can be accessed and manipulated by the host [12]. The CUDA programming model uses fine-grained data parallelism and thread-level parallelism, nested within coarse-grained data parallelism and task parallelism. The task can be partitioned into sub-tasks executed independently in parallel by blocks of threads [12]. The GPU runs a program function called kernels using thousands of threads. By considering the GPU limitations and parallel concepts, a CUDA application can achieve higher performance by maximizing the utilization of WARP and processor resources and the memory hierarchy resources.

There are two main limitations for GPGPU computations. Indeed, to increase the number of cores and SMs and achieve their energy efficiency, the GPU cores are designed very simply and with low clock rates. They only provide performance gains when a program is carefully parallelized into a large number of threads. Hence, trying to make a general code working on GPUs is a difficult task and often results in inefficient running. Instead, it is more efficient to execute only those pieces of

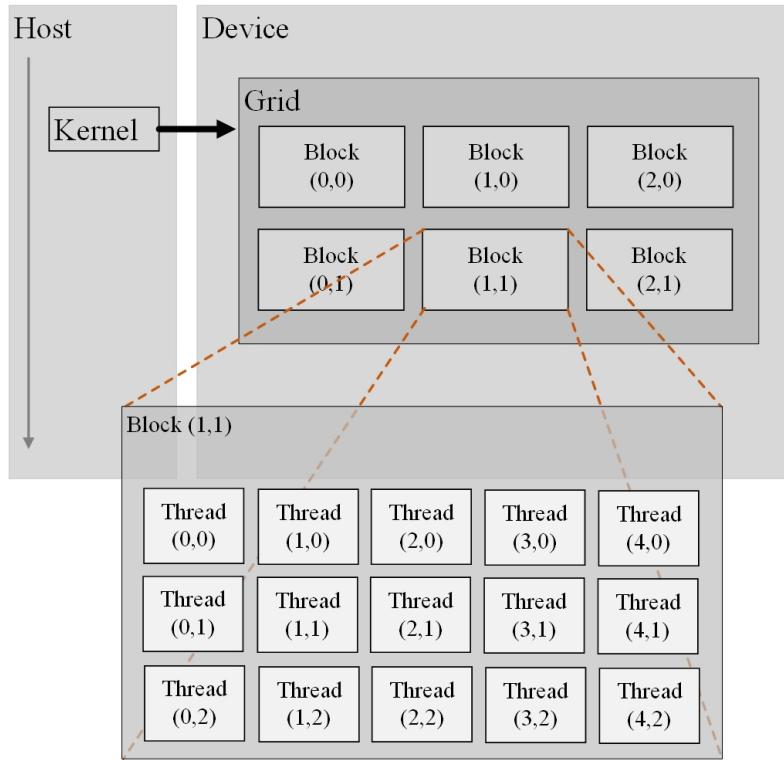


Fig. 11.2: GPU kernel and thread hierarchy (blocks of threads and grid of blocks) [12]

programs suitable to GPU-style parallelism, such as linear algebra code, and leaving the other sections of the program code for the CPU. Therefore, writing suitable programs that utilize the GPU cores is intricate even for the algorithms well-suited to be programmed in parallel. There has been a generation of libraries that provide GPU implementations of standard linear algebra kernels (BLAS), which help separate code to pieces sub-tasks used in these libraries and achieve higher performance [12]. This described the first limitation. The second limitation of GPUs for GPGPU computation is that GPUs use a separated memory from the host memory. In other words, the GPU has a special memory and access hierarchy and uses a different address space from the host memory.

The host (the CPU) and the GPU cannot share data easily. This limitation of communication between host and GPU is especially problematic when both host and GPU are working on shared data simultaneously. In this case, the data must be returned, and the host must wait for the GPU and vice versa. The worse section of this scenario is transferring data between the host and the GPU as it is slow, especially compared to the speed of host memory or the GPU dedicated memory.

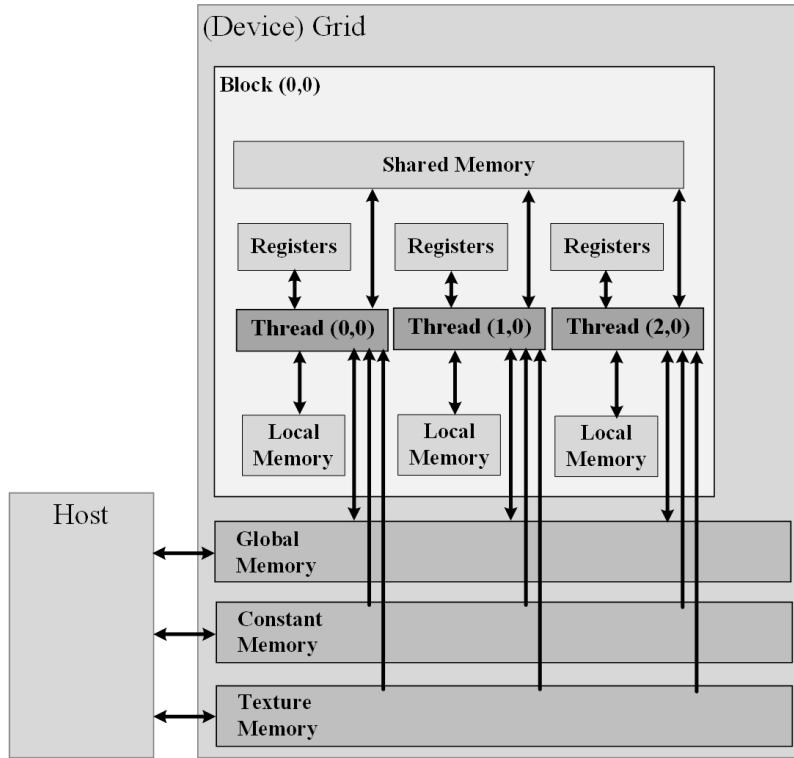


Fig. 11.3: GPU memory hierarchy (Global, Constant, Local, Shared, Registers) [12]

The maximum speed of data transferred is limited to the PCI express bus speed, as the GPU is connected to the host using the PCI express bus. Consequently, data transmission often has the highest cost in GPU computation. Several methods have tried to avoid data transfers when costs exceed the gain of GPU computation [14]. Therefore, GPU programming has many limitations that need to be considered to achieve higher performance. For example, the amount of shared memory capacity and the sequence of process executions and branch divergence, or other bottlenecks, which are needed to be fixed by considering the GPU architecture. The NVIDIA processors different architectures such as Tesla, Maxwell, Pascal, Volta, and Turing are introduced here for better exploration.

- Tesla architecture emerged by introducing the GeForce 8800 product line, which has unified the vertex and the pixel processor units. This architecture is based on scalable array processors by facilitating an efficient parallel processor. In 2007, C2050, another GPU with this architecture, found its way to the market [15]. The performance of the Tesla C2050 reaches 515 GFLOPS in double-precision arithmetic. It benefits from a 3 GB GDDR5 memory with 144 GB/s bandwidth. This GPU has 448 CUDA-enabled cores at a frequency of 1.15 GHz.

- Another NVIDIA GPU architecture codename is Kepler. This architecture was introduced in April 2012 [16, 17]. The VLSI technology feature size is 28 nm, which is better than previous architectures. This architecture was the first NVIDIA GPU that is designed considering energy efficiency. Most of the NVIDIA GPUs are based on Kepler architecture, such as K20 and K20x, which are Tesla computing devices with double-precision. The performance of the Tesla K20x is 1312 GFLOPS in double-precision computations and 3935 GFLOPS in single precision. This GPU has 6 GB GDDR5 memory with 250 GB/s bandwidth. The K20x family has 2688 CUDA cores that work at a frequency of 732 MHz. This architecture was also followed by the new NVIDIA GPUs, called Maxwell (GTX 980) and also Geforce 800M series.
- The Maxwell architecture was introduced in 2014, and the GeForce 900 series is a member of this architecture. This architecture was also manufactured in TSMC 28 nm as the same process as the previous model (Kepler) [18]. Maxwell architecture has a new generation of streaming multi-processors (SM); this architecture decreased the power consumption of the SMs. The Maxwell line of products came with a 2 MB L2 cache. The GTX980 GPU has the Maxwell internal architecture, which delivers five TFLOPS of single-precision performance [16]. In this architecture, one CUDA core includes an integer and also a floating-point logic that can work simultaneously. Each SM has a WARP scheduler, dispatch unit, instruction cache, and register file. Besides the 128 cores in each SM, there are eight load/store units (LD/ST) and eight special function units (SFU) [17]. The GTX 980 has four WARP schedulers in each SM, enabling four concurrent WARPs to be issued and executed. LD/ST units calculate source and destination addresses in parallel, letting eight threads load and store data at each address inside the on-chip cache or DRAM [19]. SFUs execute transcendental mathematical functions such as Cosine, Sine, and Square root. At each clock, one thread can execute a single instruction in SFU, so each WARP has to be executed four times if needed.
- Another NVIDIA GPU architecture codename is Pascal was introduced in April 2016 as the successor of the Maxwell architecture. As the members of this architecture, we can refer to GPU devices such as Tesla P100 and GTX 1080, manufactured in TSMC's 16 nm FinFET process technology. The GTX 1080 Ti GPU has the Pascal internal architecture, which delivers 10 TFLOPS of single-precision performance. This product has a 3584 CUDA core, which works at 1.5 GHz frequency, and the memory size is 11 GB with 484 GB/s bandwidth. GTX 1080 Ti has 28 SMs, and each SM has 128 cores, 256 KB of register file capacity, 96 KB shared memory, and 48 KB of total L1 cache with CUDA capability 6.1 [20]. This computation capability has features such as dynamic parallelism and atomic addition operating on 64-bit float values in GPU global memory and shared memory [20].
- The next-generation NVIDIA GPU architecture is Volta which succeeds the Pascal GPUs. This architecture was introduced in 2017, which is the first chip with

Tensor cores specially designed for deep learning to achieve more performance over the regular CUDA cores. NVIDIA GPUs with this architecture, such as Tesla V100, are manufactured in TSMC 12 nm FinFET process. Tesla V100 performance is 7.8 TFLOPS in double-precision and 15.7 TFLOPS in single-precision floating-point computations. This GPU card is designed with a CUDA capability of 7.0, and each SMs have 64 CUDA cores; total cores are 5120 units, and the memory size is 16 GB with 900 GB/s bandwidth, which employs the HBM2 memory features [19]. The maximum power consumption is 250 Watt designed very efficiently for power consumption and performance per watt [21].

- Another NVIDIA GPU architecture, which was introduced in 2018, is the Turing architecture, and the famous RTX 2080 Ti product is based on it [22]. The Turing architecture enjoys a capability of real-time ray tracing with dedicated ray-tracing processors and dedicated artificial intelligence processors (Tensor Cores). The performance of 4352 CUDA cores at 1.35 GHz frequency is 11.7 TFLOPS in single-precision computation, which uses GDDR6/HBM2 memory controller [22]. Its global memory size is 11 GB, and the bandwidth of this product is 616 GB/s, and the maximum amount of shared memory per thread block is 64 KB. However, the maximum power consumption is 250 Watt, manufactured with TSMC 12 nm FinFET process, and this GPU supports the compute capability 7.5 [2, 22, 26, 27].

11.2.1 CUDA Programming with Python

CUDA provides APIs with libraries for C/C++ programming languages. However, Python also has its own libraries for accessing CUDA APIs and GPGPU capabilities. PyCUDA [23] performs Pythonic access to the CUDA API for parallel computation in which claims:

1. PyCUDA has an automatic object clean-up after the object lifetime.
2. With some abstractions, it is more convenient than programming with NVIDIA's C-based runtime.
3. PyCUDA has all the CUDA's driver API.
4. It supports automatic error checking.
5. PyCUDA's base layer is written in C++; therefore, it is fast.

11.3 Analysing Codes and Functions

In order to make an implemented CPU-based application faster, it is necessary to analyze it. It is essential to find the parts of the code that may be made parallel. An approach is to find the loops in the SVM code. Especially in this chapter, we focus

on SVM with Chebyshev's first kind kernel function. The proposed work is divided into two parts, the training (fit) function, and the test (project) function.

11.3.1 Analyzing the Training Function

At the beginning of the fit function, the two-nested for-loop calculates the K matrix. The matrix K is 2D with the train size elements for each dimension. Inside the nested loops, a Chebyshev function is calculated in each iteration. The recursive form of Chebyshev calculation is not appropriate. In recursive functions, handling the stack and always returning the results of each function call is problematic. GPU-based implementations of the repeated memory access tasks by recursive functions are not efficient. However, the T_n polynomial has an explicit form, which is called the Chebyshev polynomial. For more details of the Chebyshev function and its explicit form, refer to chapter 3. The matrix P is the result of an inner product of matrix K with the vectorized outer product on the target array. The next complex part of the fit function is a quadratic problem (QP) solver. For the case of the GPU acceleration, CUDA has its own QP solver APIs.

11.3.2 Analyzing the Test Function

In the project function (test or inference), the complex part is the iterations inside the nested loop, which in turn contains a call to the Chebyshev function. As mentioned in sub-section 11.3.1, we avoid recursive functions due to the lack of efficient stack handling in GPUs, and thus using them is not helpful. Therefore, for the test function, we do the same we did in the training function for calling the Chebyshev function.

11.4 Hardware and Software Requirements

So far, we have elaborated on the proper GPU architecture to be selected for an SVM application. In our application (accelerating the LS-SVM programs), a single-precision GPU device is computationally sufficient, while it complies with performance-cost limitations. This is in contrast to the training phase in which a double-precision GPU is preferred. This work has been tested on an NVIDIA Tesla T4 GPU device based on Turing architecture, a power-efficient GPU device with 16GB memory and 2560 CUDA cores and 8.1 TFLOPS performance; however, any other CUDA-supported GPU devices may be used for this purpose. As a first step to replicate this platform, the CUDA Toolkit and appropriate driver for your device must be installed. Then, Python3 must be installed in the operating system. Python may be installed independently or with the Anaconda platform and its libraries. As

an alternative, we recommend using Google Colab [24] to get all the hardware requirements and software drivers available with only registration and few clicks on a web page.

```

!nvidia-smi
Fri May 7 09:09:01 2021
+-----+
| NVIDIA-SMI 465.19.01    Driver Version: 460.32.03    CUDA Version: 11.2 |
+-----+
| GPU Name      Persistence-M| Bus-Id     Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap| Memory-Usage | GPU-Util  Compute M. |
| |                               |             |            MIG M. |
+-----+
| 0  Tesla T4           Off  | 00000000:00:04.0 Off |          0 |
| N/A   76C   P0    34W /  70W |    112MiB / 15109MiB |     0%      Default |
|                               |                  |            N/A |
+-----+
+-----+
| Processes:
| GPU  GI  CI      PID  Type  Process name        GPU Memory |
| ID   ID              |                 Usage
|-----+

```

Fig. 11.4: The output of nvidia-smi command

Install PyCUDA with the following command

```
pip3 install pycuda
```

To check the CUDA toolkit and its functionality, execute this command

```
nvidia-smi
```

The above command results in Fig. 11.4 in which the GPU card information with its running processes. To test PyCUDA, you may execute the program listed in Program 1 using Python3 are displayed.

Program 1: First PyCUDA program to test the library and driver

```

1: import pycuda.driver as cuda
2: import pycuda.autoinit
3: from pycuda.compiler import SourceModule
4:
5: if __name__ == "__main__":
6:
7:     mod = SourceModule("""
8:         #include <stdio.h>
9:         __global__ void myfirst_kernel(){
10:             printf("Tread[%d, %d]: Hello PyCUDA!!!\\n",
11:                   threadIdx.x, threadIdx.y);
12:         }
13:         """
14:     )
15:     function = mod.get_function("myfirst_kernel")

```

In this program, the PyCUDA library and its compiler are loaded. Then a source module is created, which contains C-based CUDA kernel code. At line 15, the function is called with a 4X4 matrix of threads inside a single grid.

11.5 Accelerating the Chebyshev Kernel

For accelerating the Chebyshev Kernel, its explicit form (refer to chapter three) is more applicable. As mentioned before, handling recursive functions due to the lack of stack support in GPUs is complex. Therefore, the T_n ¹ statements from the Chebyshev_Tn function should be replaced with its explicit form as listed in Program 2.

Program 2: The explicit form of Chebyshev_Tn function.

```

1: import pycuda.driver as cuda
2: from pycuda.compiler import SourceModule
3: import numpy
4: import pdb
5:

```

¹ The statement of one iteration in the recursive form of a first kind *Chebyshev* (refer to chapter three).

```

6: def Chebyshev(x,y,n=3,f='r'):
7:     m=len(x)
8:     chebyshev_result = 0
9:     p_cheb = 1
10:
11:    for j in range(0,m):
12:        for i in range(0,n+1):
13:            a= np.cos(i * np.arccos(x[j]))
14:            b= np.cos(i * np.arccos(y[j]))
15:            chebyshev_result += (a * b)
16:            weight = np.sqrt(1 - (x[j] * y[j]) + 0.0002)
17:            p_cheb *= chebyshev_result / weight
18:    return p_cheb

```

In the next step, lets take a look at the statement that calls the function `Chebyshev`. The third line in Program 3, means calling `Chebyshev(X[i], X[j], n=3, f='r')`. This function is called n_sample^2 times, and as n_sample equals 60, then it will be called 3600 times.

Program 3: The statement which calls the function `Chebyshev`.

```

1: for i in range(n_samples):
2:     for j in range(n_samples):
3:         K[i, j] = self.kernel(X[i], X[j])

```

Accordingly, at first, the `Chebyshev` function is changed to CUDA kernel as a single thread GPU function, then it is changed to a multi-thread GPU function to remove the nested for-loop instruction shown in Program 3. The C-based CUDA kernel is shown in Program 4. In this code, the instruction `np.cos(i * np.arccos(x[j]))` has changed to `cos(i * acos(x[j]))` because of the translation of Python to C programming language. A single thread will execute the `Chebyshev` function; therefore, if you put it inside the nested for-loop shown in Program 3, this single thread will be called 3600 times. According to GPUs' architecture mentioned previously, the GPU's global memory is located on the device. Hence, the `x` and `y` variables should be copied to the device memory from the host side (the main memory from the CPU side). Therefore, for small and sequential applications, this process will take a longer time in comparison with the CPU execution without GPU.

Program 4: C-based CUDA kernel for `Chebyshev_GPU` function

```

1: __global__ void Chebyshev(float *x,
                           float *y,
                           float *p_cheb){
2:     int i, j;
3:     float a,b, chebyshev_result, weight;
4:     chebyshev_result =0;
5:     int idx = blockDim.x * blockIdx.x + threadIdx.x *4;
6:     p_cheb[idx] = 1;
7:     for( j=0; j<4; j++){
8:         for( i=0; i<4; i++){
9:             a = cos(i * acos(x[j+idx]));
10:            b = cos(i * acos(y[j+idx]));
11:            chebyshev_result += (a * b);
12:        }
13:        weight = sqrt(1 - (x[j+idx] * y[j+idx]) + 0.0002);
14:        p_cheb[idx] *= chebyshev_result / weight;
15:    }
16: }
```

This program should transfer a massive block of data in single access to reduce the number of accesses and minimize the memory access latency. Moreover, when a GPU is active, all its functional units consume energy; hence, utilizing more threads is more efficient. In Program 5, the listed program breaks the inner for-loop (second line of Program 3), then it calls the Chebyshev function only 60 times. According to Program 5, to break the mentioned for-loop, the `threadIdx.x` is used. The matrix data has been considered as a linear array. Each row of the matrix `y` is followed after the first row of it in a single line. Therefore, the starting index of each row is calculated with `threadIdx.x * 4`.

Before calling the CUDA Kernel function, we have to load all the needed variables and array elements on the GPU device's global memory. The `mem_alloc` instruction allocates the required memory on the device. After that, the `memcpy_htod` copies the data from the host to the device. As shown in lines 8 and 12, the `x_gpu` and `y_gpu` are the allocated memory on the device. The `x` variable is a single row of the `X` matrix, containing 60 rows with four features. We send all the `X` matrix as the `y` input. In line 35, the block size for the threads is defined. In this code, the block contains 60 (`n_samples`) threads in the `x` dimension, and its `y` dimension is set to 1, which shows a single dimension block is used. In order to break the other for-loop (first line from Program 3), a two-dimensional block with 60×60 threads could be used. In this case, together with reducing memory access from the host site for getting `p_cheb` outputs or writing `x` and `y` variables on the device, the amount of data elements would be reduced as you do not need to copy the `y` variable to the

y_gpu.

Program 5: C-based CUDA kernel for the multiple threads

```

1:  for i in range(n_samples): #n_samples
2:      p_cheb = np.random.randn(1,n_samples)
3:      p_cheb = p_cheb.astype(np.float32)
4:      p_cheb_gpu = cuda.mem_alloc(p_cheb.nbytes)
5:      cuda.memcpy_htod(p_cheb_gpu, p_cheb)
6:      t1 = X[i].astype(np.float32)
7:      x_gpu = cuda.mem_alloc(t1.nbytes)
8:      cuda.memcpy_htod(x_gpu, t1)
9:
10:     t2 = X.astype(np.float32)
11:     y_gpu = cuda.mem_alloc(t2.nbytes)
12:     cuda.memcpy_htod(y_gpu, t2)
13:
14:     mod = SourceModule("""
15:         #include <stdio.h>
16:         __global__ void Chebyshev(float *x,
17:                                 float *y,
18:                                 float *p_cheb){
19:             int i, j;
20:             float a,b, chebyshev_result, weight;
21:             chebyshev_result = 0;
22:             int idx = blockIdx.x  blockDim.x + threadIdx.x *4;
23:             p_cheb[idx] = 1;
24:             for( j=0; j<4; j++){
25:                 for( i=0; i<4; i++){
26:                     a = cos(i * acos(x[idx+j]));
27:                     b = cos(i * acos(y[idx+j]));
28:                     chebyshev_result += (a * b);
29:                 }
30:                 weight = sqrt(1-(x[idx+j]*y[idx+j])+0.0002);
31:                 p_cheb[idx] *= chebyshev_result / weight;
32:             }
33:         """
34:         func = mod.get_function("Chebyshev")
35:         func(x_gpu,y_gpu,p_cheb_gpu, block=(n_samples,1,1))
36:         p_cheb = np.empty_like(p_cheb)
37:         cuda.memcpy_dtoh(p_cheb, p_cheb_gpu)
K[i] = p_cheb;

```

At the end of the code (Program 5) in line 37, the CPU reads the whole data from the `p_cheb_gpu` location with the command `memcpy_dtoh`, which means copy the memory from the device to the host. Therefore, each GPU execution brings a row containing 60 elements for the K matrix.

11.6 More Optimizations

In this section, we focus on Program 5. Although it breaks 3600 calls of the Chebyshev function into only 60 calls, it is not efficient yet. This is because the execution time on the CPU is less than it on GPU. Therefore, for acceleration, it is worth considering these hints:

1. The object `SourceModule` (line 15 of Program 5) is not a compiled source code. Hence, having this statement inside a for-loop is inefficient because of the extra time for compilation in each iteration. We may move lines 15 to 33 of Program 5 outside the for-loop before line 1.
2. Having memory allocation in each iteration of a for-loop is time-consuming. Therefore, we should allocate the needed memory at first, and we use it in every iteration.
3. It is more efficient to increase the utilization of CUDA cores. This will be reached by hiring more threads in our implementation. This issue will increase parallelism and also reduces memory communications.

After the optimizations, 2.2X speedup is obtained on Tesla T4 GPU over the CPU in a Colab machine containing two VCPUs of intel Xeon processor working at 2.20GHz and with 13GB memory. You may download the codes from the footnote link². In the chosen SVM classification code with the first kind of Chebyshev kernel, there are two parts, fit function and predict function. Significantly, the speedup for the fit function is 58X over the CPU. The nested loops inside the fit function result in more array-based computations. Therefore, the nature of our algorithm directly affects the rate of acceleration.

11.7 Accelerating the QP Solver

Quadratic Programming (QP) is the process of solving a particular mathematical problem. As mentioned in previous chapters, it is needed to solve a quadratic equation and find its minimal answer to create the best separating line. Therefore, we calculate the K matrix and prepare all the constraint matrixes to make a quadratic

² <https://github.com/sampp098/SVM-Kernel-GPU-acceleration->

problem. Previously the CVXOPT library was used to solve the mentioned quadratic equation. One of the complex parts of the code is the QP solver function. However, there is a valuable library inside the CUDA API for the quadratic equations. Moreover, it is possible to use the PyTorch and QPTH [25] libraries inside our Python implementation. As a prerequisite, the following needed libraries should be installed.

Installing torchvision and qpth packages

```
pip3 install torchvision
pip3 install qpth
```

After the installation, we have to import the solver:

Import the solver in your code

```
from qpth.qp import QPFunction
```

The following shows the problem formulation. The standard form of the QP is used as follows:

$$\begin{cases} \min \frac{1}{2}(x^T)Px + q^T.x \\ \text{subject to : } Gx \leq h, \text{ and } Ax = b \end{cases}$$

However, the QPTH define a quadratic program layer as:

$$\begin{cases} \min \frac{1}{2}(z^T)Qz + p^T.z \\ \text{subject to : } Gz \leq h, \text{ and } Az = b \end{cases}$$

The differences are only at the name of variables. Therefore, we replace the statement:

```
solution = cvxopt.solvers.qp(P, q, G, h, A, b)
```

to:

```
solution = QPFunction(verbose=False)(P, q, G, h, A, b)
```

It is also necessary to change all the CVXOPT matrices to NumPy form, for example:

```
P = np.outer(y, y) * K
```

This library will automatically send the process to execute on the GPU device. The PyTorch package also can be used for matrix multiplication and also other matrix operations on the GPU device.

11.8 Conclusion

In this chapter, the internal architecture of some NVIDIA GPUs was explained briefly. The GPUs are the many-core processors that have their memories. They are traditionally used only for graphical processing, for example, rendering videos or enhancing the graphics in computer games. Due to the structure of these devices, recently, their usage has significantly changed toward general-purpose applications, i.e., GPGPU programming. Machine Learning applications, with massive datasets and deep neural networks, are the most used applications nowadays running on GPU devices. The same as a specific application, for the SVM Kernel trick on classification problems, GPU devices can perform better when coupled with the CPUs.

Moreover, some methods for the GPGPU on the previous LS-SVM implementations (especially the first kind Chebyshev kernel) are proposed. Instead of CUDA, in our Python implementation PyCUDA package is used which resulting in an acceptable performance. An important issue is a gap between the main memory and the device memory on GPU devices. Therefore, it should be noted more access to the device memory will result in lower performance on the execution. Also, it should be noted that the GPGPU shows its superiority when there is a large dataset. However, the structure of a GPGPU program plays a significant role in getting speedups. In the first part of this chapter, GPU devices' architecture is explained, which is needed to be known to optimize the GPU kernels. There exist many libraries that use GPUs as their processors in the background, far from the user side. If the developer does not have enough knowledge of parallel programming, these libraries are the best choices to be employed.

We have used the mentioned acceleration methods, and hints based on GPGPU and the SVM application with the first kind Chebyshev function as its kernel is edited based on these methods. The experiments show 2.2X speedup (for both fit and test function), is gained over the CPU on Colab's Tesla T4 GPU. The partial optimization, only on fit function, resulted in a better speedup of about 58X due to the structure of this function. The main leads for getting performance are reducing memory access together with increasing the CUDA core utilization. In the optimization process, unwanted extra instructions inside the loops, like memory allocations and compilation processes are omitted.

References

1. Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., Purcell, T. J.: A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* **26**, 80–113 (2007)
2. Ahmadzadeh, A., Hajihassani, O., Gorgin, S.: A high-performance and energy-efficient exhaustive key search approach via GPU on DES-like cryptosystems. *J Supercomput* **74**, 160–182 (2018)
3. Gavahi, M., Mirzaei, R., Nazarbeygi, A., Ahmadzadeh, A., Gorgin, S.: High performance GPU implementation of k-NN based on Mahalanobis distance. In 2015 International Symposium

- on Computer Science and Software Engineering (CSSE), 1–6 (2015)
4. Luo, C., Fei, Y., Luo, P., Mukherjee, S., Kaeli, D.: Side-channel power analysis of a GPU AES implementation. In 2015 33rd IEEE International Conference on Computer Design (ICCD), 281–288 (2015)
 5. Asghari, M., Hadian Rasanan, A.H., Gorgin, S., Rahmati, D., Parand, K.: FPGA-orthopoly: a hardware implementation of orthogonal polynomials. *Eng Comput* (2022) doi: 10.1007/s00366-022-01612-x
 6. Xiao, B., Wang, H., Wu, J., Kwong, S., Kuo, C. C. J.: A multi-grained parallel solution for HEVC encoding on heterogeneous platforms. *IEEE Trans Multimedia* **21**, 2997–3009 (2019)
 7. Rahmani, S., Ahmadzadeh, A., Hajihassani, O., Mirhosseini, S., Gorgin, S.: An efficient multi-core and many-core implementation of k-means clustering. In ACM-IEEE International Conference on Formal Methods and Models for System Design (MEMOCODE), 128–131 (2016)
 8. Moayeri, M. M., Hadian Rasanan, A. H., Latifi, S., Parand, K., Rad, J. A.: An efficient space-splitting method for simulating brain neurons by neuronal synchronization to control epileptic activity. *Eng Comput*, 1–28 (2020)
 9. Parand, K., Aghaei, A. A., Jani, M., Ghodsi, A.: Parallel LS-SVM for the numerical simulation of fractional Volterra's population model. *Alex. Eng. J.* **60**, 5637–5647 (2021)
 10. Allec, S. I., Sun, Y., Sun, J., Chang, C. E. A., Wong, B. M.: Heterogeneous CPU+GPU-enabled simulations for DFTB molecular dynamics of large chemical and biological systems. *J. Chem. Theory Comput* **15**, 2807–2815 (2019)
 11. Doi, J., Takahashi, H., Raymond, R., Imamichi, T., Horii, H.: Quantum computing simulator on a heterogenous hpc system. In Proceedings of the 16th ACM International Conference on Computing Frontiers, 85–93 (2019)
 12. Cheng, J., Grossman, M., McKercher, T.: Professional CUDA c programming. John Wiley & Sons, Amsterdam (2014)
 13. Dalrymple R. A.: GPU/CPU Programming for Engineers Course, Class 13, (2014)
 14. AlSaber, N., Kulkarni, M.: Semcache: Semantics-aware caching for efficient gpu offloading. In Proceedings of the 27th international ACM conference on International conference on supercomputing, 421–432 (2013)
 15. Pienaar, J. A., Raghunathan, A., Chakradhar, S.: MDR: performance model driven runtime for heterogeneous parallel platforms. In Proceedings of the international conference on Supercomputing 225–234 (2011)
 16. Corporation, N.: CUDA Zone, <https://developer.nvidia.com/cuda-zone>, (2019)
 17. Wang, C., Jia, Z., Chen, K.: Tuning performance on Kepler GPUs: An introduction to Kepler assembler and its usage in CNN optimization. In GPU Technology Conference Presentation, (2015)
 18. NVIDIA, T.: NVIDIA GeForce GTX 750 Ti: Featuring First-Generation Maxwell GPU Technology, Designed for Extreme Performance per Watt, (2014)
 19. Mei, X., Chu, X.: Dissecting GPU memory hierarchy through microbenchmarking. *IEEE Trans Parallel Distrib Syst* **28**, 72–86 (2016)
 20. NVIDIA, T.: P100. The most advanced data center accelerator ever built. Featuring Pascal GP100, the world's fastest GPU (2016)
 21. NVIDIA, T.: V100 GPU architecture. The world's most advanced data center GPU. Version WP-08608-001_v1, (2017)
 22. NVIDIA, T.: NVIDIA Turing GPU architecture: Graphics reinvented, (2018)
 23. PyCUDA 2021, documentation, <http://documen.tician.de/pycuda/>, (2021)
 24. Welcome To Colaboratory, <https://colab.research.google.com>, (2021)
 25. A fast and differentiable QP solver for PyTorch, <https://locuslab.github.io/qpth/>, (2021)
 26. Kalaiselvi, T., Sriramakrishnan, P., Somasundaram, K.: Survey of using GPU CUDA programming model in medical image analysis. *Inform. Med. Unlocked* **9**, 133–144 (2017)
 27. Choquette, J., Gandhi, W., Giroux, O., Stam, N., Krashinsky, R.: Nvidia a100 tensor core gpu: Performance and innovation. *IEEE Micro* **41**, 29–35 (2021)

Chapter 12

Classification Using Orthogonal Kernel Functions: Tutorial on ORSVM Package

Amir Hosein Hadian Rasanan and Sherwin Nedaei Janbesaraei and Amirreza Azmoon and Mohammad Akhavan and Jamal Amani Rad

Abstract Classical and fractional orthogonal functions and their properties as the kernel functions for the SVM algorithm are discussed throughout this book. In Chapters 3, 4, 5, and 6, the four classical families of the orthogonal polynomials (Chebyshev, Legendre, Gegenbauer, and Jacobi) were considered and the fractional form of these polynomials were presented as kernel function and their performance has been shown. However, using these kernels needs much effort to implement. To make it easy for anyone who needs to try and use these kernels a Python package is provided here. In this chapter, the ORSVM package is introduced as an SVM classification package with orthogonal kernel functions.

Amir Hosein Hadian Rasanan

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, G.C. Tehran, Iran, e-mail: amir.h.hadian@gmail.com

Sherwin Nedaei Janbesaraei

School of Computer Science, Institute for Research in Fundamental Sciences e-mail: sherwin.nedaei@gmail.com

Amirreza Azmoon

Department of Computer Science, The Institute for Advance Studies in Basic Sciences (IASBS), Zanjan, Iran e-mail: a.azmoon@iasbs.ac.ir

Mohammad Akhavan

School of Computer Science, Institute for Research in Fundamental Sciences, Tehran, Iran e-mail: mohammad.akhavan@ipm.ir

Jamal Amani Rad

Department of Cognitive Modeling, Institute for Cognitive and Brain Sciences, Shahid Beheshti University, Evin, Tehran 19839, Iran, e-mail: j.amanirad@gmail.com

12.1 Introduction

Python programming language is one of the most popular programming languages among developers and researchers. However, there are a bunch of reasons why Python is popular, a large share of this popularity is because there exist hundreds of open-source Python packages which one can include in her main code and extent her code's functionality, without any need for more coding. This makes it possible for one to develop a reusable code. This feature of programming languages cultivates the development process by making it easier and faster. Suppose you have written a code for a customized SVM algorithm of your own, consisting of some classes with relevant functions. Instead of writing the same classes and functions, every time and everywhere you need to access them, you can save those classes and functions in a separate file let's name it, *custom-svm.py*, then you only need to add a single line of code *import custom-svm* , somewhere in your code to gain access to all the functions of your custom SVM algorithm. It is been a wide accepted paradigm of using, reusable codes specifically in the Python programming language. There exist multiple Python packages available for SVM classification, such as the *SVM* which *Scikit learn* package provides. Although it is easy to use it lacks kernel diversity, more specifically orthogonal kernels and definitely the novel fractional orthogonal kernels. Kernels in SVM play a critical role, without a doubt there does not exist a kernel that suits every problem of the real world, in precise one kernel function is not suitable for all datasets because each kernel function produces the higher dimension in SVM with the same data scattering pattern. Suppose the linear kernel function, $K(x, y) = x \cdot y$, the decision boundary of such kernel is a straight line in 2D space or a flat plane in 3D space. No argument that a flat plane can not classify all datasets. Hence, there is a perpetual struggle to find the most suitable kernel function for each dataset, and a new kernel function is always welcomed if can improve the success metrics of classification tasks. To make it easier for anyone to use orthogonal kernels functions and also the fractional form, we decided to create the Python package **ORSVM** (consists of multiple modules). Through this chapter, this package will be introduced. Moreover, the basics of the **ORSVM**, how to install and how to use it, are discussed all through examples.

12.1.1 ORSVM

ORSVM is a free and open-source Python package that provides an SVM classifier with some novel orthogonal kernel functions. This library provides a complete chain of using the SVM classifier from normalization to calculation of SVM equation and the final evaluation. However, note that there are some necessary steps before normalization, that should be handled for every dataset, such as duplicate checking, Null values or outlier handling, or even dimensionality reduction or whatever enhancements that may apply to a dataset. These steps are out of the scope of the SVM algorithm, thereupon, **ORSVM** package. Instead, the normalization step

which is a must before sending data points into orthogonal kernels is handled directly in ORSVM, by calling the relevant function. As it has been discussed already, the fractional form of all kernels is achievable during the normalization process too. **ORSVM** package includes multiple classes and functions. All of them will be introduced in this chapter.

ORSVM package is heavily dependent on *Numpy* and *cvxopt* packages. *Arrays*, *matrices* and *linear algebraic* functions are used repeatedly from *numpy* and the heart of SVM algorithm which is solving the convex equation of SVM and finding the *Support Vectors* is achievable by means of a convex quadratic solver from *cvxopt* library which is in turn a free Python package for convex optimization¹.

ORSVM is structured on two modules. A module consists of Kernel classes and the other one consists of relevant classes and functions supporting initialization, normalization, fitting process, capturing the fitted model and the classification report. **ORSVM** is structured as follows:

- **orsvm** module

This is the main module consisting of the fitting procedure, prediction and report. It includes Model and SVM classes and the transformation (normalization) function. We opted for the *transformation* name instead of *normalization* because transforming to fractional space is achievable through this function too.

- **Model** Class

Model class literally creates the model. Initialization starts under this class. Calling the *model.model_fit* function initiates an object from the SVM class. After calling the *Transformation* function, the train set gets ready to input into the SVM object. Then, an object is ready to start the fitting process of the SVM object from SVM class.

1. **ModelFit** function

Initiates an object from the SVM class and transforms/normalizes the train set, and calls the fit function of the SVM object. Finally captures the fitted model and parameters.

2. **ModelPredict** function

Transforms/Normalizes the train set. Calls the predict function of the SVM object with proper parameters. And finally calls *accuracy_score*, *confusion_matrix* and *classification_report* of *Scikit learn* (*sklearn.metrics*) with the previously captured result.

- **SVM** class

Here the svm equation is formed, required matrices of *cvxopt*, are created under the fit function of the SVM class. Invokes *cvxopt* and solves the SVM equation and to determine the support vectors and calculate the hyper-plane's equation. The prediction procedure is implemented under the prediction function of the SVM class.

¹ A suitable guide on this package is available at <http://cvxopt.org> about installation and how to use it.

1. **fit** function creates proper matrices by directly calling kernels. Such matrices are the *Gram matrix*, and also other matrices of the SVM equation required by *cvxopt*. As the result, *cvxopt* returns *Lagrange Multipliers*. Applying some criteria of user interest, support vectors opt from them, and as a result, the *SVM.fit* function returns the weights and bias of the hyper-plane's equation and also the array of support vectors.
 2. **predict** function maps data points from the test set with the decision boundary(hyper-plane equation) and determines to which class each data points belong.
- **Transformation** function
This is the function that normalizes the input dataset or in the case of fractional form, transforms the input data set into fractional space, in other words, normalizes the input in fractional space.
- **Kernels** module includes one class per kernel. So currently there exists four classes for each orthogonal kernel. The following classes are available in the *kernels* module:
 - **Chebyshev** class contains the relevant functions to calculate the orthogonal polynomials and fractional form of the Chebyshev family of the first kind,
 - **Legendre** class holds the relevant functions to calculate the orthogonal polynomial sand fractional form of the Legendre family.
 - **Gegenbauer** class consists of the relevant functions to calculate the orthogonal polynomials and fractional form of the Gegenabuer family.
 - **Jacobi** class includes the relevant functions to calculate the orthogonal polynomials and fractional form of the Jacobi family.

ORSVM is written in Python programming language which requires Python version 3 or higher.

12.2 How to install

To use the package you can install it using pip as blow:

```
pip install orsvm.py
```

or you can easily clone the package from our GitHub and use the setup.py file to install the package;

```
python setup.py install
```

12.3 Model Class

This is the interface to use the **ORSVM** package. Using **ORSVM** starts with creating an object of the *Model* class. This class includes a *ModelFit* function which itself

creates an **ORSVM** model as an instance of the *SVM* class. Then normalizes the input data in the case of the normal form of the kernel, or transforms the input data in the case of fractional form. Choosing between normal or fractional is determined by the value of the argument T if $T = 1$ the kernel is in normal form, and in the case of $0 < T < 1$ then it is the fractional form. *model_fit* function receives the train and test set separately, moreover, each of them should be divided into two matrices of x and y . The matrix x is the whole data set where the relevant column to output a.k.a class or label, is omitted and the matrix y is that specific column of class or label. Division of datasets can be achieved through multiple methods. A widely used one is *StratifiedShuffleSplit* from *sklearn.model_selection*.

Creating an object of Model class requires some input parameters, cause itself needs to create an SVM object with these passed parameters. Following example code creates an *orsvm* classifier from the *Jacobi* kernel function of order 3.

```
obj = orsvm.Model(kernel="jacobi", order=3, T=0.5,
                  k_param1=-0.8, k_param2=0.2,
                  sv_determiner='a', form='r', C=100)
```

Here $T = 0.5$ means that it is in the fractional form of order 0.5. "k_param1" is equivalent to ψ , and "k_param2" is equivalent to ω in case "Jacobi" kernel is chosen. If it was *Gegenbauer*, then only "k_param1" was applicable to the kernel. Because *Chebyshev* and *Legendre* both do not have any hyper-parameter, there is no need to pass any value for "k_param1" and "k_param2", even if one passes values for these parameters, those will be ignored. Parameter *sv_determiner* is the user's choice on how support vectors have to be selected among Lagrange Multipliers (data points which were computed through convex optimization solver and opted as candidates to draw the SVM's hyperplane). This is one of the important parameters that can affect final classification performance metrics. It is considered 3 options for choosing support vectors. Two options are widely used, first a number of type int, that represents the number of support vectors that should be selected from Lagrange Multipliers, if the number is greater than that number of Lagrange Multipliers then all of them, will be selected as support vectors. The second one is a number in scientific notation (often a negative power of 10) as a minimum threshold, and the third method which is considered in **ORSVM** is the flag 'a' that represents *Average*. Most of the time, we have no clear conjecture about the values of Lagrange Multipliers, how small they are, or how many of them will be available. Therefore, we do not know what the threshold should be, or how many support vectors should be chosen. Sometimes it leads to **zero** support vectors, then there will be an **Error**. Our solution to this situation, which always happens for a new dataset, is the *average* method. In this case, **ORSVM** finds the average of Lagrange Multipliers and sets the threshold. Then, never an error has occurred. However, this method does not guarantee the best generalization accuracy. But gives a factual estimation about support vectors. After all, choosing the best number of support vectors, itself is an important task to do in SVM. The parameter *form* is only applicable to *Chebyshev* kernel, because two implantations of *Chebyhsev Kernel* is available, an explicit equation and a recursive one. So 'r' refers to recursive and 'e' refers to 'explicit'. Parameter 'noise'

is only applicable to the *Jacobi* kernel, and the Jacobi's weight function indeed. In fact, 'noise' purpose is to avoid the errors that happen at the boundaries of weight function as it has already been explained. Finally, parameter 'C' is the regularization parameter of the SVM algorithm, which controls to what degree the misclassified data points are important. Setting 'C' to the best value leads to a better generalization of the classifier. Table 12.1 summarizes the parameters of the *Model* class.

Table 12.1: The Model class of ORSVM, and parameters list

| | Parameter | Default | Value |
|---|------------------|----------------|--|
| 1 | kernel | Chebyshev | Jacobi, Gegnbauer, Chebyshev, Legendre |
| 2 | order | 3 | 3, 4, 5, ... |
| 3 | T | 1 | 0<T<1 |
| 4 | param1 | None | <i>param1</i> > -1 for the Jacobi kernel, <i>param1</i> > -0.5 for the Gegenbauer kernel |
| 5 | param2 | None | <i>param2</i> > -1 for the Jacobi kernel |
| 6 | svd | 'a' | int, scientific notation, 'a' |
| 7 | form | 'r' | 'r' , 'e' |
| 8 | noise | '0.01' | often: 0.01, 0.001, ... |
| 8 | C | None | often: 10, 100, 1000, ... |

Right at this point, **ORSVM** just has initiated the model and has not done any computation yet, to fit the model we have to call the *ModelFit* function. Clearly, fitting requires an input dataset. As already discussed *ModelFit* receives train and test datasets which are divided into *x* (data without label) and *y* (label). The following code snippet represents how one can divide the dataset. Function *LoadDataSet*, reads and loads data set into a *pandas DataFrame*, then converts maps the classes to binary classification. As the "Clnum" is the label column, we have to select and convert that column solely into one Numpy array. Then remove the label column from Pandas data frame to reach the data without the label. It converts data into *numpy array*, however, it is not necessary. Calling the 'LoadDataSet' function gives the *x* and *y*. In the next line using *StratifiedShuffleSplit* we can create an object of stratified shuffle split with required parameters and then using the *split* function of the created object we can get the train and test set divided into *X* and *y*.

Now that train and test set are ready, we can call *ModelFit* with proper parameters and as the result function prints status messages and returns weights and bias of the SVM's hyperplane equation and also an array for support vectors and the kernel instance. Therefore, we have the fitted model and corresponding parameters we can use it to predict. By calling the *ModelPredict* function the final step of classification with **ORSVM** will be achieved. *ModelPredict* requires text sets and also the bias and the kernel instance to calculate the accuracy. It should be noted that the test set will be transformed into proper space through the *ModelPredict* function. As the

result, the accuracy score will be returned. Moreover, *ModelPredict* will print the confusion matrix, classification report, and the accuracy score.

```

import pandas
from sklearn.model_selection import StratifiedShuffleSplit

def LoadDataSet():
    # load dataset
    df = pandas.read_csv('/home/data/spiral.csv',
                         names=['Chem1', 'Chem2', 'Clnum'],
                         index_col=False)

    #convert to binary classification
    df.loc[df.Clnum!=1, ['Clnum']] = -1
    df.loc[df.Clnum==1, ['Clnum']] = 1
    y_np=df['Clnum'].to_numpy()
    df.drop('Clnum', axis=1, inplace=True)
    df_np=df.to_numpy()

    return df_np,y_np

X,y = data_set()
sss = StratifiedShuffleSplit(n_splits=5,
                             random_state=30,
                             test_size=0.9)

for train_index, test_index in sss.split(X, y):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

Weights, S_Vectors, Bias, K_Instance = obj.ModelFit(X_train,
                                                    y_train)

Accuracy_score = obj.ModelPredict(X_test,
                                  y_test,
                                  Bias,
                                  K_Instance)

```

12.4 SVM Class

The SVM class is the heart of this package where the fitting process happens, the SVM equation is evaluated to find the support vectors and finally, the equation of hyper-plane is constituted. Before getting delve into svm equation and finding the support vectors, we have to do the kernel trick. So we need a "Gram matrix", the

matrix of all possible inner products of x_train under the selected kernel. Therefore, considering $X_train_{m \times n}$ the "SVM.fit" will create a square matrix of shape $m \times m$:

$$K = \begin{bmatrix} k(x_1, x_2) & \cdots & k(x_1, x_m) \\ \vdots & & \vdots \\ \cdots & k(x_i, x_j) & \cdots \\ \vdots & & \vdots \\ k(x_m, x_1) & \cdots & k(x_m, x_m) \end{bmatrix},$$

This is also known as *kernel matrix* as it represents the kernel trick of SVM. "K" can be any of *Legendre*, *Gegenbauer*, *Chebyshev*, or *Jacobi*.

Here, after the convex optimization problem, or better to say minimization of the dual form of SVM equation, will be solved using *qp* function from *cvxopt* library:

```
cvxopt.solvers.qp(P, q, G, h, A, b)
```

This function solves a quadratic program:

$$\begin{aligned} & \text{minimize } \frac{1}{2}x^T Px + q^T x, \\ & \text{subject to } \quad \quad \quad Gx \leq h, \\ & \quad \quad \quad Ax = b. \end{aligned} \tag{12.1}$$

According to documents provided by developers of this library, the argument of *qp* functions are²:

- **P** is an $n \times n$ dense or sparse Wigner D-matrix with the lower triangular part of **P** stored in the lower triangle. It should be positive semi-definite.
- **q** is an $n \times 1$ dense Wigner D-matrix.
- **G** is an $m \times n$ dense or sparse Wigner D-matrix.
- **h** is an $m \times 1$ dense Wigner D-matrix.
- **A** is a $p \times n$ dense or sparse Wigner D-matrix. In particular, **A** is the y matrix (containing y values of the training set) with shape of row=1 and column = (number of rows of the x matrix)
- **b** is a $p \times 1$ dense Wigner D-matrix or None.

The output of *qp* contains the *Lagrange Multipliers*, which support vectors lie between them and we need the policy to choose some of them as support vectors. Here the trade-off is between over-fit and under-fit, choosing too much of them as support vectors cause the model to over-fit whereas selecting a few, leads to under-fitting the problem.

SVM class includes a fit function that constructs the Gram matrix using the specified kernel function. Thereafter, solving the SVM's equation returns the support vectors. Moreover, the bias is calculated in *Svm.fit* function. Also, *Svm.predict* is another function of class SVM which predicts labels of a given array. The user

² For comprehensive documentation and examples of cvxopt, please visit: <https://cvxopt.org>.

never needs to call functions under the SVM class directly, all usage of ORSVM is achievable through Model Class and its functions.

12.4.1 Chebyshev Class

Another kernel introduced in this package is the **Chebyshev** kernel, which is implemented in the *vectorial* approach which has already been introduced as Generalized Chebyshev Kernel. Similar to the Legendre kernel, the Chebyshev kernel is available when initiating the Model class object by passing the 'chebyshev' parameter as the kernel name. Another parameter applicable to the Chebyshev kernel is *form*. Two different types of Chebyshev kernel are implemented in the Chebyshev class. One uses an explicit equation and another is a recursive function. Selecting a type to use is possible through *form* parameter. The parameters of the Chebyshev class are as follows:

- *order* := Order of polynomial
- *form* := "e" for *Explicit form* and "r" for *Recursive form*

12.4.2 Legendre Class

Legendre kernel class is generally given as an object to the kernel parameter of *Models* class in the initialization step of an object and also its *Legendre-kernel* function is explicitly used in constructing the gram matrix K . Legendre class benefits from a recursive implementation of the Legendre kernel function. To use this kernel, the user only needs to initiate the Model object with 'Legendre' as kernel parameter of the Model class. The Legendre class needs the following parameters:

- *order* := Order of the Legendre kernel function.

12.4.3 Gegenbauer Class

The **Gegenbauer** kernel is also available in the *ORSVM* package. For the implementation of the Gegenbauer kernel function the fractional kernel function which is introduced in chapter 5 is used. Therefore in addition to the product of input vectors, the values are obtained from the two other equations. The Gegenbauer class has the following parameters:

- *order* := Order of the Gegenbauer kernel function.
- *lambda* := The λ parameter of the Gegenbauer kernel function.

12.4.4 Jacobi Class

The **Jacobi** kernel is the last orthogonal kernel currently available in the *ORSVM* package. *Jacobi kernel* is available to choose when the ORSVM package is imported. Simply, the kernel name should be ‘jacobi’ during the initiation of an object from the *Model* class. The Jacobi class needs the following parameters:

- *order* := Order of the Jacobi kernel function.
- *psi* := The ψ parameter of Jacobi kernel function.
- *omega* := The ω parameter of Jacobi kernel function.
- *noise* := A small noise to weight function to avoid errors at boundaries.

12.5 Transformation function

The transformation function is actually the composition of the well-known *Min-max feature scaling*

$$x' = \frac{x - x_{min}}{x_{max} - x_{min}},$$

and the mapping equation already introduced for the fractional form of all kernels.

$$x' = 2x^\alpha - 1,$$

so we have:

$$x' = 2\left(\frac{x - x_{min}}{x_{max} - x_{min}}\right)^\alpha - 1,$$

which transforms the input x into the kernel space related to the fractional-order of function (α). Therefore, the transformation function requires the x (input data) and alpha ('T' as a parameter of the Model class, 'T' represents the transformation step.) which is 1 by default and causes to normalize the input data as the Min-max feature scaling function does. The user never needs to call transformation function directly, but in the case on needs so:

```
import orsvm
orsvm.transformation(x, T=1)
```

12.6 How to use

Using the ORSVM library is easy and straightforward. The simplicity of use has been an important motivation in developing the ORSVM package. The user only should provide the dataset as matrices and select a kernel and by setting the T , the user can choose between the normal or fractional form of the kernel functions. Other

required parameters related to the chosen kernel should be provided in the object initiation step. Those parameters are completely discussed already. In this section, we only demonstrate a sample code of classification of a dataset using *ORSVM*.

As an example, the three monks problem dataset is considered which has been introduced before in Chapter 3. Three Monks' problem has the train set and test set separated. In case one needs to do this separation please refer to the code snippet in the Section 12.3. In order to use ORSVM, we first have to divide the train and test into *x_train* and *y_train*. This can be done by importing the dataset into a pandas data frame first, and then we will map the class values in monks dataset to -1 and 1 which are suitable for the SVM algorithm instead of 0 and 1.

```
import numpy as np
import pandas as pd
import orsvm

# load train-set
df = pd.read_csv('/home/datasets/1_monks.train',
                 names=['Class', 'col1', 'col2', 'col3',
                        'col4', 'col5', 'col6'],
                 index_col=False)

df.loc[df.Class==0, ['Class']] = -1 # map "0" to "-1"
y_train=df['Class'].to_numpy() # convert y_train to numpy array
df.drop('Class', axis=1, inplace=True) # drop the class label
X_train=df.to_numpy() # convert x_train to numpy array

# load test-set
df = pd.read_csv('/home/datasets/1_monks.test',
                 names=['Class', 'col1', 'col2', 'col3',
                        'col4', 'col5', 'col6'],
                 index_col=False)
df.loc[df.Class==0, ['Class']] = -1
y_test=df['Class'].to_numpy()
df.drop('Class', axis=1, inplace=True)
X_test=df.to_numpy()
```

Now that we have the *train-set* and *test-set* ready, we need an instance of Model class, to call the **ModelFit** function using proper arguments. For example, here we choose the Chebyshev kernel with $T = 0.5$ and we let the SVM's *regularization* parameter keep the default value, that is "None" and also the recursive implementation is preferred, in the second line by calling the *ModelFit* function, *ORSVM* fits the model and return the fitted parameters. We can capture these parameters for later use, for example for prediction.

```
# Create an object from Model class of ORSVM
obj = orsvm.Model(kernel="Chebyshev", order=3, T=0.5, form='r')
```

```
# fit the model and Capture parameters
Weights, S_Vectors, Bias, K_Instance = obj.ModelFit(X_train,
                                                    y_train)
```

These are only for logging purposes. Then in case one needs the prediction, may call the *ModelPredict* function that requires the test-set divided into x and y, and also the bias and the kernel instance from the previous step.

```
accuracy_score = obj.ModelPredict(X_test,
                                   y_test,
                                   Bias,
                                   K_Instance)
```

ModelPredict returns the accuracy score and we can capture it. Moreover this function prints much more information on classification. Here is the output:

```
***** 06/09/2021 16:43:14 *****
** ORSVM kernel: Chebyshev
** Order: 3
** Fractional mode, transition: 0.5
** Average SV determiner!
** SV threshold: 10^-4
```

Fig. 12.1: Jacobi Polynomials of order 5, fixed ψ and different ω

Using the log information may help for better debugging or gain a better understanding of how fitting gets done for a dataset and by setting the *log* parameter of the *fit* function as *True*.

After mentioning the basics of SVM in part one, introducing some fractional orthogonal kernel functions in part two, and reviewing some applications of SVM algorithms, the aim of this chapter was to present a python package which enables us to apply the introduced fractional orthogonal kernel functions in real world situations. The architecture of this package and a brief tutorial usage of it, are presented here. But for more information and a detailed updated tutorial on this package you can visit the online page of this package which is available at orsvm.readthedocs.io.

Part V

Appendices

Appendix A

Python Programming Prerequisite

Mohammad Akhavan

Abstract The objective of this appendix is to prepare the readers for gaining a relative knowledge of the Python programming language. In this appendix, the focus is on an introduction to Python which includes the foundation of the language, process of installing and defining variables, loops, conditions, and functions. In closing, three libraries are introduced; Pandas regarding data analysis, Numpy for working with arrays, and Matplotlib which is used for drawing graphs and charts. In fact, this appendix is a prerequisite for the concepts and applications of the ORSVM package introduced in this book.

A.1 Introduction

Python is an object-oriented, multi-paradigm, high-level programming language created by Guido van Rossum and was released in 1991. Its high-level built-in data structures and dynamic binding encourage developers to use it for application development as well as use it as a scripting language. According to the TIOBE Programming Community Index¹, Python is one of the top three popular programming languages of 2020 and was the winner of 2018 best programming language.

Python 2.0 was released on 16 October 2000 with minor fixes, optimizations, better error handling messages, cycle-detecting garbage collector, and support for Unicode. Python 3.0 was released on 3 December 2008 that is not completely backward-compatible. Python 2.7's end-of-life date was set in 2015 but postponed to 2020. The main reason for the postponement was concern that a large body of existing code could not easily be forward-ported to Python 3.

Mohammad Akhavan
School of Computer Science, Institute for Research in Fundamental Sciences, Tehran, Iran e-mail:
mohammad.akhavan@ipm.ir

¹ <https://www.tiobe.com/tiobe-index/>

Some of the very popular features of this programming language for users is its interpreter and the extensive standard library in source or binary form, which gives users the capability of using it without charge for all major platforms and this feature allows us to run the same compiled code on multiple platforms instead of recompilation again. As another feature of Python, the big community and a lot of open-source frameworks, tools, and libraries that decrease the cost and increase the speed of development can be noticed.

A.2 Basics of Python

In this section, we try to introduce the fundamentals of Python language, how to install Python and how to use this language, and also how to define variables, loops, conditions, and functions. Pandas, Numpy, Matplotlib is three libraries that are used for all the data scientists' tasks and we will discuss their preparations.

A.2.1 How to use Python?

There are many ways you can install Python and work with it. Here, some ways that are more appropriate are discussed. The basic way of installing is to go to the Python website² and find the suitable version of Python for your work based on your operation system. After downloading and installing Python, also we need a code editor for writing out code such as Notepad++³, Atom⁴, Sublime⁵, or any code editor. These code editors are open source and we can easily download and use them for free. There is also some commercial program like PyCharm, Spyder, Pydev, etc. The applicability of a program can be seen in debugging of big Python programs. There is another way to install Python using Anaconda⁶, a Python distribution that its objective is scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.). By installing Anaconda, all important and famous packages are effortlessly installed. Additionally, Anaconda installs some programs such as Jupyter Notebook⁷ which is a well-known open-source platform to develop code. One of the advantages of Jupyter Notebook over others is that in this program we can run Python code cell by cell which can be very helpful. For installing Anaconda in Windows or macOS you can download the graphical installer, but for Linux .sh file should be downloaded and run in the

² <https://www.python.org/downloads/>

³ <https://notepad-plus-plus.org/>

⁴ <https://atom.io/>

⁵ <https://www.sublimetext.com/>

⁶ <https://www.anaconda.com>

⁷ <https://jupyter.org/>

terminal. This is the last step to install Python and its packages. Now the basics of the language can be explained.

A.2.2 Python Basics

The basics of Python are very similar to other programming languages like C, C++, C#, Java, so if one is familiar with these programming languages can easily learn Python.

A.2.2.1 Basic Syntax

Python syntax can be executed by writing directly in the Command-Line. For example, write the following command.

```
print("Hello, World!")
```

Moreover, we can create a Python file; then, write the code in it, save the file using the .py extension, and run it in the Command-Line. For instance

```
> Python test.py
```

Python uses white-space indentation for delimiting blocks instead of curly brackets or keywords. The number of spaces is up to the programmer, but it has to be at least one.

```
if 2 > 1:  
    print("Two is greater than one!")  
if 2 > 1:  
    print("Two is greater than one!")  
  
> Two is greater than one!
```

A.2.2.2 Comments

Comments start with a "#" for the purpose of in-code documentation, and Python will render the rest of the line as a comment. Comments also can be placed at the end of a line. In order to prevent Python from executing commands, we can make them a comment.

```
# This is a comment.  
print("Hello, World!")  
print("Hello, World!") # This is a comment.  
# print("Hello, World!")
```

A.2.2.3 Variable Types

A Python variable unlike other programming languages does not need an explicit declaration to reserves memory space. The declaration happens automatically when a variable is created by assigning a value to it. The equal sign (=) is used to assign values to variables. The type of a variable can be changed by setting a new value to it.

```
x = 5 # x is integer
y = "Adam" # y is string
y = 1 # y is integer
```

String variables can be declared by using single or double-quotes.

```
y = "Adam" # y is string
y = 'Adam' # y is string
```

Rules for Python variables are:

- A variable name must start with a letter or underscore character and can not start with a number.
- A variable name can only contain alpha-numeric characters and underscores.
- Variable names are case-sensitive (yy, Yy, and YY are three different variables)

Python allows multiple variables to be assigned in one line and the same value is assigned to multiple variables in one line:

```
x, y, z = "Green", "Blue", "Red"
x = y = z = 2
```

In Python, a *print* statement can be used to combine both text and a variable(s) or mathematical operation with the + character but a string and a numeric variable are not integrable:

```
x = "World!"
print("Hello " + x)
x = 2
y = 3
print(x + y)
x = 5
y = "Python"
print(x + y) # raise error

> Hello World!
> 5
> TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Variables that are created outside of a function or class are known as global variables and can be used everywhere, both inside and outside of functions.

```
x = "World!"
def myfunc():
    print("Hello " + x)
myfunc()

> Hello World!
```

If a variable was defined globally and is redefined in a function with the same name, its local value can only be useable in that function and the global value remains as it was:

```
x = "World!"
def myfunc():
    x = "Python."
    print("Hello " + x)
myfunc()

> Hello Python.
```

To create or change a global variable locally in a function, a *global* keyword must use before the variable:

```
x = "2.7"
def myfunc():
    global x
    x = "3.7"
myfunc()
print("Python version is" + x)

> Python version is 3.7
```

A.2.2.4 Numbers and Casting

There are three types of numbers in Python:

- int
- float
- complex

which can be represented as below:

```
a = 10 # int
b = 3.3 # float
c = 2j # complex
```

The *type()* is a command used for identifying the type of any object in Python. Casting is an attribute to change the type of a variable to another. This can be done by using that type and giving the variable as input:

- *int()* manufacture an integer number from a float (by rounding down it) or string (it should be an integer).
- *float()* manufacture a float number from an integer or string (it can be float or int)
- *str()* manufacture a string from a wide variety of data types.

```
a = int(2)      # a will be 2
b = int(2.8)    # b will be 2
c = int("4")    # c will be 4
x = float(8)     # x will be 8.0
y = float(4.8)   # y will be 4.8
z = float("3")   # z will be 3.0
w = float("4.2") # w will be 4.2
i = str("aa1")   # i will be 'aa1'
j = str(45)      # j will be '45'
k = str(3.1)     # k will be '3.1'
```

A.2.2.5 Strings

Strings in Python can be represented by either quotation marks, or double quotation marks. Even string can be assigned to a variable and it can be a multi-line string by using three double quotes or three single quotes:

```
a = "Hello"
print(a)
> Hello

a = """Strings in Python can be represent
surrounded by either single quotation marks,
or double quotation marks."""
print(a)

> Strings in Python can be represent
surrounded by either single quotation marks,
or double quotation marks.

a = '''Strings in Python can be represent
surrounded by either single quotation marks,
or double quotation marks'''
print(a)

> Strings in Python can be represent
surrounded by either single quotation marks,
or double quotation marks.
```

In Python, there is no dedicated data type as the character. In fact, a single character is simply a string with a length equal to 1. Strings in Python like many other programming languages are arrays of bytes representing Unicode characters.

```
a = "Hello, World!"  
print(a[3])  
  
> 1
```

Slicing is an attribute in Python that returns a range of values in an array or list. In this case, slicing returns a range of characters. In order to specify the starting and the ending indices of range, separate them by a colon. Negative indexes can be used to start the slice from the end of a list or array.

```
a = "Hello, World!"  
print(a[3:7])  
print(a[-6:-1])  
  
> lo, W  
> World
```

To spot the length of a string, use the *len()* function.

```
a = "Hello, World!"  
print(len(a))  
  
> 13
```

There are so many built-in methods that are applicable on strings, just to name a few:

- *strip()* removes any white-space from the beginning or the end:

```
a = "Hello, World!"  
print(a.strip())  
  
> Hello, World!
```

- *lower()* returns the string in lower case:

```
a = "Hello, World!"  
print(a.lower())  
  
> hello, world!
```

- *upper()* returns the string in upper case:

```
a = "Hello, World!"  
print(a.upper())  
  
> HELLO, WORLD!
```

- *replace()* replaces a string with another one:

```
a = "Hello, World!"  
print(a.replace("W", "K"))  
  
> Hello, Korld!
```

- The `split()` method splits the string into substrings if it finds instances of the separator:

```
a = "Hello, World!"
print(a.split("o"))

> ['Hell', ' ', 'W', 'rld!']
```

Use the keywords `in` for checking if there is a certain phrase or a character in a string:

```
txt = "Hello, World! My name is Python."
if "Python" in txt:
    print("The Python word exist in txt.")

> The Python word exist in txt.
```

To concatenate, or combine two strings, `+` operator can be used.

```
x = "Hello"
y = "World"
z = x + " " + y
print(z)

> Hello World
```

Numbers can not be combined with strings. But we can apply casting by `str()` or use the `format()` method that takes the passed arguments and place them in the string where the placeholders `{}` are. It is worth mentioning that the `format()` method can take an unlimited number of arguments:

```
version = 3.7
age = 29
txt1 = "My name is Python, and my version is {} and my age is {}"
txt2 = "My name is Python, and my version is " + str(version) \
+ " and my age is " + str(age)
print(txt1.format(version, version))
print(txt2)

> My name is Python, and my version is 3.7 and my age is 29
> My name is Python, and my version is 3.7 and my age is 29
```

An escape character is a backslash \ for writing characters that are illegal in a string. For example a double quote inside a string that is surrounded by double quotes.

```
txt1 = "This is an "error" example."
txt2 = "This is an \"error\" example."
print(txt2)

> SyntaxError: invalid syntax
> This is an "error" example.
```

- `\'` single quote
- `\\` backslash
- `\\n` new line
- `\\t` tab
- `\\r` carriage Return
- `\\b` backspace
- `\\f` form feed
- `\\ooo` octal value
- `\\xhh` Hex value

A.2.2.6 Lists

Lists are one of all four collection data types. When we decide to save some data in one variable, a list container can be used. In Python lists, elements are written by square brackets and an item is accessible by referring to its index number.

```
mylist = ["first", "second"]
print(mylist)
print(mylist[1])
> ['first', 'second']
> second
```

As mentioned in the Strings section, a string is a list and all the negative indexing and range of indexes can be used for lists, and also remember that the first item in Python language has an index 0. By considering one side of colon devoid, the range will go on to the end of the beginning of the list.

```
mylist = ["first", "second", "Third", "fourth", "fifth"]
print(mylist[-1])
print(mylist[1:3])
print(mylist[2:])
> fifth
> ['second', 'Third']
> ['Third', 'fourth', 'fifth']
```

To change an item value in a list, we can refer to its index number. Iterating through the list items is possible by for loop. To determine that an item is present in a list we can use *in* keyword and with *len()* function we can find how many items are in a list.

```
mylist = ["first", "second"]
mylist[0] = "Third"
for x in mylist:
    print(x)
if "first" in mylist:
    print("No, first is not in mylist")
print(len(mylist))
```

```
> Third
> second
> No, first is not in mylist
> 2
```

There are various methods for lists and some of them are discussed in this appendix.

- *append()* adds an item to the end of the list.
- *insert()* inserts an item at a given position. The first argument is the index and the second is value.
- *remove()* removes the first item from the list.
- *pop()* removes the item at the selected position in the list, and returns it. If no index is specified *pop()* removes and returns the last item in the list.
- *del* removes the variable and makes it free for new value assignment.
- *clear()* removes all items from the list.

For making a copy of a list the syntax of `list1 = list2` can not be used. By this command, any changes in one of the two lists are applied to the other one, because `list2` is only a reference to `list1`. Instead, *copy()* or *list()* methods can be utilized to make a copy of a list.

```
mylist1 = ["first", "second"]
mylist2 = mylist1.copy()
mylist2 = list(mylist1)
```

For joining or concatenating two lists together we can easily use ‘+’ between two lists or using *extend()* method.

```
mylist1 = ["first", "second"]
mylist2 = ["third", "fourth"]
mylist3 = mylist1 + mylist2
print(mylist3)
mylist1.extend(mylist2)
print(mylist1)

> ['first', 'second', 'third', 'fourth']
> ['first', 'second', 'third', 'fourth']
```

A.2.2.7 Dictionary

Dictionary is another useful data type collection whose main difference with other collections is that the dictionary is unordered and unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by keys. Type of the keys can be any immutable types as strings and numbers. In addition, one can create an empty dictionary with a pair of braces. A dictionary with its keys and values is defined as follows:

```

mydict = {
    "Age": 23,
    "Gender": "Male",
    "Education": "Student"
}
print(mydict)

> {'Age': 23, 'Gender': 'Male', 'Education': 'Student'}

```

In order to access the items of a dictionary, we can refer to its key name inside square brackets or call the `get()` method. Moreover, the corresponding value to a key can be changed by assigning a new value to it. The `keys()` and `values()` methods can be used for getting all keys and values in the dictionary.

```

print(mydict["Age"])
print(mydict.get("Age"))
mydict["Age"] = 24
print(mydict["Age"])
print(mydict.keys())
print(mydict.values())

> 23
> 23
> 24
> dict_keys(['Age', 'Gender', 'Education'])
> dict_values([23, 'Male', 'Student'])

```

Iterating through dictionaries is a crucial technique to access the items in dictionary. The return value of the loop can be keys of the dictionary or by the key and corresponding value using the `items()`.

```

for x in mydict:
    print(x) # printing Keys
for x in mydict:
    print(mydict[x]) # printing Values
for x, y in mydict.items():
    print(x, y) # printing Keys and Values

> Age
> Gender
> Education
> 23
> Male
> Student
> Age 23
> Gender Male
> Education Student

```

Checking the existence of a key in a dictionary can be done by the *in* keyword same as lists. Moreover, to determine how many items are in a dictionary we can use the *len()* method. Adding an item to a dictionary can be easily done by using the value and its key.

```
if "Name" in mydict:  
    print("No, there is no Name in mydict.")  
print(len(mydict))  
mydict["weight"] = 72  
print(mydict)  
  
> No, there is no Name in mydict.  
> 3  
> {'Age': 23, 'Gender': 'Male', 'Education': 'Student' \  
, 'weight': 72}
```

A.2.2.8 If ... Else

Logical conditions from mathematics can be used in "if statements" and loops. An "if statement" is written by using the *if* keyword. The scope of "if statement" is defined with white-space while other programming languages use curly brackets. Also, we can use an *if* statement inside another *if* statement called nested if/else by observing the indentation.

- Equals: $a == b$
- Not Equals: $a != b$
- Less than: $a < b$
- Less than or equal to: $a <= b$
- Greater than: $a > b$
- Greater than or equal to: $a >= b$

```
x = 10  
y = 15  
if y > x:  
    print("y is greater than x")  
  
> y is greater than x
```

elif is a Python keyword that can be used for "if the previous conditions were not true, then try this condition". And the *else* keyword used for catches anything that couldn't be caught by previous conditions.

```
x = 10  
y = 9  
if x > y:  
    print("x is greater than y")  
elif x == y:
```

```

    print("x and y are equal")
else:
    print("y is greater than x")

> y is greater than x

```

There are some logical keywords that can be used for combining conditional statements like *and*, *or*.

```

a = 10
b = 5
c = 20
if a > b and c > a:
    print("Both conditions are True")
if a > b or a > c:
    print("At least one of the conditions is True")

> y is greater than x

```

You can have an *if* statement inside another *if* statement.

```

a = 10
b = 5
c = 20
if a > b:
    if a > c:
        print("a is grater than both b and c.")

> a is grater than both b and c.

```

A.2.2.9 While Loops

While loop is one of two types of iterations in Python that can execute specific statements as long as a condition is true.

```

i = 1
while i < 3:
    print(i)
    i += 1

> 1
> 2

```

Just like *if* statement, we can use *else* code block for when the while loop condition is no longer true.

```

i = 1
while i < 3:
    print(i)

```

```

    i += 1
else:
    print("i is grater or equal to 3.")

> 1
> 2
> i is grater or equal to 3.

```

A.2.2.10 For Loops

The *for* statement in Python differs from what can be used in other programming languages like C and Java. Unlike other programming languages that the *for* statement always iterates over an arithmetic progression of numbers or lets the user define both the iteration step and halting condition, in Python, we can iterate over the items of any sequence (like string).

```

nums = ["One", "Two", "Three"]
for n in nums:
    print(n)
for x in "One.":
    print(x)

> One
> Two
> Three
> 0
> n
> e
> .

```

There are some statements and a function that can be very useful in *for* statement which will be discussed in the next section.

A.2.2.11 Range, Break and Continue

If it is needed to iterate over a sequence of numbers, there is a built-in function *range()* which generates the arithmetic progression of numbers. Consider the *range()* which starts from 0 and ends at a specified number, except 1. To iterate over the indices of a list or any sequence, we can use the combination of the *range()* and *len()* functions that are applicable. The *range()* function by default moves in the sequence by one step, so we can specify the increment value by adding a third parameter to the function.

```

for x in range(3):
    print(x)

```

```
> 0
> 1
> 2

for x in range(2, 4):
    print(x)

> 2
> 3

nums = ["One", "Two", "Three"]
for i in range(len(nums)):
    print(i, nums[i])
for x in range(2, 10, 3):
    print(x)

> 0 One
> 1 Two
> 2 Three

for x in range(2, 10, 3):
    print(x)

> 2
> 5
> 8
```

Same as *while* loop, *else* statement can be used after *for* loop. In Python, *break* and *continue* are used like other programming languages to terminate the loop or skip the current iteration, respectively.

```
for x in range(10):
    print(x)
    if x > 2:
        break

> 0
> 1
> 2

for x in range(3):
    if x == 1:
        continue
    print(x)

> 0
> 2
```

A.2.2.12 Try, Expect

Like any programming language, Python has its own error handling statements named *try* and *except* to test a block of code for errors and handle the error, respectively.

```
try:
    print(x)
except:
    print("x is not define!")

> x is not define!
```

A developer should handle the error completely in order that the user can see where the error is accrued. For this purpose, we should use a *raise Exception* statement to raise the error.

```
try:
    print(x)
except:
    raise Exception("x is not define!")

> Traceback (most recent call last):
  File "<stdin>", line 4, in <module>
Exception: x is not define!
```

A.2.2.13 Functions

A function is a block of code, may take inputs, doing some specific statements or computations which may produce output. The purpose of a function is reusing a block of code to perform a single, related activity to enhance the modularity of the code. A function is defined by *def* keyword and is called by its name.

```
def func():
    print("Inside the function")

func()
> Inside the function
```

information is passed into the function by adding arguments after the function name, inside the parentheses. We can add as many arguments as is needed.

```
def func(arg):
    print("Passed argument is", arg)

func("name")
> Passed argument is name
```

```
def func(arg1, arg2):
    print("first argument is", arg1, "and the second is", arg2)

func("name", "family")
> first argument is name and the second is family
```

An argument defined in a function should be passed to the function when the function is called. Default parameter values are used to avoid an error in calling a function. So even when the user does not pass value to the function, the code works properly by default values.

```
def func(arg="Bob"):
    print("My name is", arg)

func("john")
func()

> My name is john
> My name is Bob
```

For returning a value from a function we can use the *return* statement

```
def func(x):
    return 5 * x

print(my_function(3))
> 15
```

It should be mentioned a list is passed to a function by reference.

```
def func(x):
    x[0]=1

A = [10, 11, 12]
func(A)
print(A)
> [1, 11, 12]
```

A.2.2.14 Libraries

Sometimes an attribute is needed in programming which is not defined in Python's standard library. Python library is a collection of functions and methods that allows us to use them without any effort. There are so many libraries written in Python language that can easily be installed, and then be imported codes.

In the following sections, some basic libraries that are used in the next chapter for data gathering, preprocessing, creating arrays are presented.

A.3 Pandas

Pandas is an open-source Python library that delivers data structures and data analysis tools for the Python programmer. Pandas is used in various sciences including finance, statistics, analytic, data science, machine learning, etc. Pandas is installed easily by using conda or pip:

```
> conda install pandas
```

or

```
> pip install pandas
```

For importing it we usually use a shorter name as follows:

```
import pandas as pd
```

The major two components of pandas are *Series* and *DataFrame*. A *Series* is essentially a column of data, and a *DataFrame* is a multi-dimensional table. in other words it is a collection of *Series*. A pandas Series can be created using the following constructor:

```
pandas.Series(data, index, dtype, copy)
```

Where the parameters of the constructor are described as:

| | |
|-------|--|
| data | data takes various forms like ndarray, list, constants |
| index | Index values must be unique and hashable, same length as data. Default np.arange(n) if no index is passed. |
| dtype | dtype is for data type. If None, data type will be inferred. |
| copy | Copy data. Default False. |

When we want to create a *Series* from a ndarray, the index should have the same length with ndarray. However, by default index are equal to range(n) where n is array length.

```
import pandas as pd
import numpy as np
mydata = np.array(['a', 'b', np.nan, 0.2])
s = pd.Series(mydata)
print(s)

> 0    a
  1    b
  2    NaN
  3    0.2
dtype: object
```

As we can see *Series* can be a complication of multi-type data. A *DataFrame* is a two-dimensional data structure which can be created using the following constructor:

```
pandas.DataFrame( data, index, columns, dtype, copy)
```

The parameters of the constructor are as follows:

| | |
|---------|---|
| data | Take various forms like ndarray, series, map, lists, dict, constants, etc. |
| index | Labels of rows. The index default value is np.arange(n). |
| columns | Labels of column. its default value is np.arange(n). This is only true if no index is passed. |
| dtype | Data type of each column. |
| copy | Copy data. |

The DataFrame can be created using a single list or a list of lists or a dictionary.

```
import pandas as pd
data =[['Alex',10],['Bob',12],['Clarke',13]]
df = pd.DataFrame(data,columns=['Name','Age'])
df2 = pd.DataFrame({
'A': 1.,
'B': pd.Timestamp('2013-01-02'),
'C': pd.Series(1, index=list(range(4)), dtype='float32'),
'D': np.array([3] * 4, dtype='int32'),
'E': pd.Categorical(["test", "train", "test", "train"]),
'F': 'foo'})
```

```
print(df)
print(df2)

>      Name      Age
0      Alex      10
1      Bob       12
2    Clarke      13

>      A          B      C      D      E      F
0  1.0  2013-01-02  1.0    3  test  foo
1  1.0  2013-01-02  1.0    3  train  foo
2  1.0  2013-01-02  1.0    3  test  foo
3  1.0  2013-01-02  1.0    3  train  foo
```

Examples are taken from ⁸ and ⁹. Here we mention some attributes that are useful for a better understanding of the data frame:

⁸ https://www.tutorialspoint.com/Python_pandas/Python_pandas_dataframe.html

⁹ https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html

```

df2.dtypes # show columns data type
df.head() # show head of data frame
df.tail(1) # show tail of data frame
df.index # show index
df.columns # show columns
df.describe() # shows a quick statistic summary of your data

> A          float64
B      datetime64[ns]
C          float32
D          int32
E        category
F          object
dtype: object

>      Name  Age
0    Alex   10
1     Bob   12
2  Clarke   13

>      Name  Age
2  Clarke   13

> RangeIndex(start=0, stop=3, step=1)

> Index(['Name', 'Age'], dtype='object')

>          Age
count    3.000000
mean    11.666667
std     1.527525
min    10.000000
25\%   11.000000
50\%   12.000000
75\%   12.500000
max    13.000000

```

You can use `df.T` to transposing your data:

```

df.T

>      0      1      2
Name  Alex  Bob  Clarke
Age    10    12    13

```

In *Pandas*, creating csv file from data frame or reading a csv file into a data frame is done as follow:

```
df.to_csv('data.csv')
pd.read_csv('data.csv')

10
```

A.4 Numpy

Numerical Python or *NumPy*, is a library consisting of multidimensional array objects, the ability to perform mathematical and logical operations on arrays. In *NumPy* dimensions are called axes. A list like

```
[1, 2, 3]
```

has one axis which has three elements. Lists inside a list like

```
[[1, 2, 3], [1, 2, 3]]
```

has two axes, each with a length of three.

NumPy's array class called *ndarray* is also known by the alias *array*. Note that *numpy.array* is not the same as the Standard Python Library class *array.array*, which only handles one-dimensional arrays and has less functionality. Some important attribute of an *ndarray* object are

- *ndarray.reshape* changes shape of dimensions of array
- *ndarray.ndim* contains the number of axes (dimensions) of the array.
- *ndarray.shape* is a tuple of integers indicating the size of the array in each dimension.
- *ndarray.size* consists of the total number of elements in the array.(product of the elements of *shape*)
- *ndarray.dtype* describes the type of the elements in the array.
- *ndarray.itemsize* shows the size of each element of the array in bytes.

```
import numpy as np
a = np.arange(15)
a = a.reshape(3, 5)
print(a)

> array([[ 0,  1,  2,  3,  4],
       [ 5,  6,  7,  8,  9],
       [10, 11, 12, 13, 14]])

a.shape

> (3, 5)

a.ndim
```

¹⁰ https://pandas.pydata.org/pandas-docs/stable/getting_started/10min.html#min

```
> 2
a.dtype.name
> 'int64'
a.size
> 15
```

A user defined array can be constructed by using the *np.array* function or converting an existing list into a ndarray by the *np.asarray* function:

```
import numpy as np
a = np.array([2, 3, 5])
b = [3, 5, 7]
c = np.asarray(b)
```

¹¹

A.5 Matplotlib

Matplotlib is a library for making 2D plots of arrays in Python. Matplotlib is an inclusive library for creating static, animated, and interactive visualizations in Python. We have a deeper look into the Matplotlib library in order to use it for visualizing our data so that we can understand the data structure better. Matplotlib and its dependencies can be installed using conda or pip:

```
> conda install -c conda-forge matplotlib
or
> pip install pandas
```

Matplotlib is a grading of functions that makes Matplotlib to work like MATLAB.

A.5.1 Pyplot

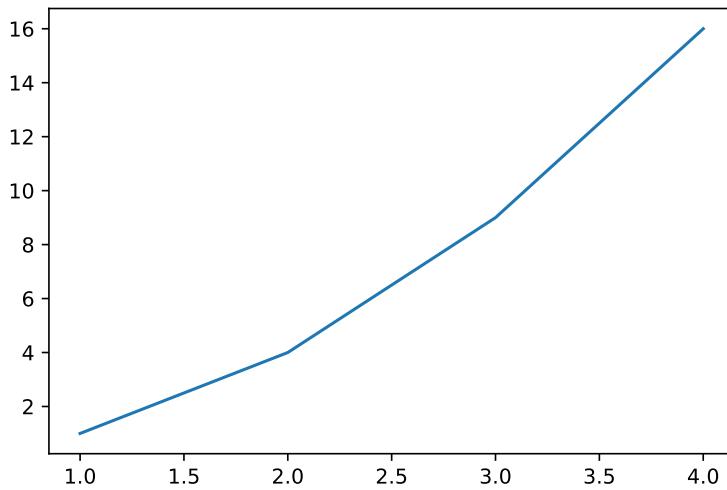
In order to import this library, usually, it is common to use a shorter name as follows:

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

For plot x versus y, easily the *plot()* function can be utilized in the following way:

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16])
```

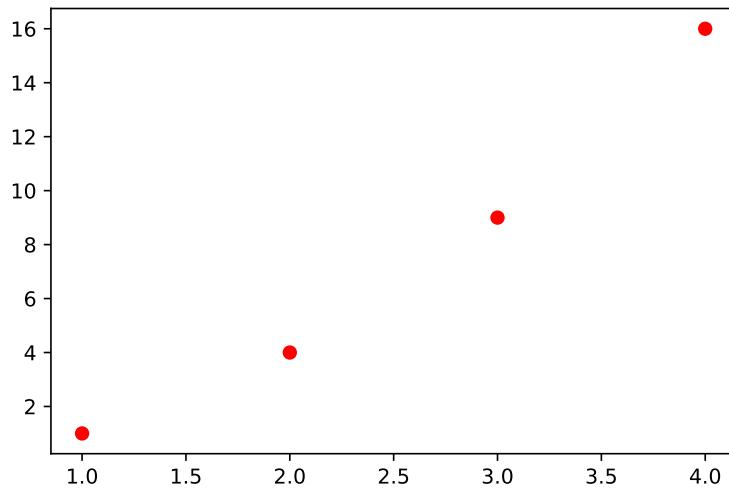
¹¹ <https://numpy.org/devdocs/user/quickstart.html>



A.5.1.1 Formatting the style of your plot

In order to distinguish each plot, there is an optional third argument that indicates the color and the shape of the plot. The letters and symbols of the format string are like MATLAB.

```
plt.plot([1, 2, 3, 4], [1, 4, 9, 16], 'ro')
```

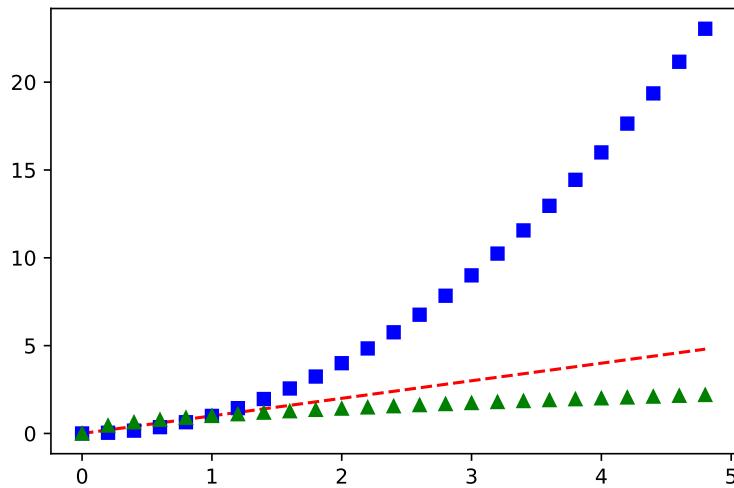


Numpy arrays also can be used to plot any sequenced data.

```
import numpy as np

# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green Radical.
plt.plot(t, t, 'r--', t, t**2, 'bs', t, np.sqrt(t), 'g^')
plt.show()
```

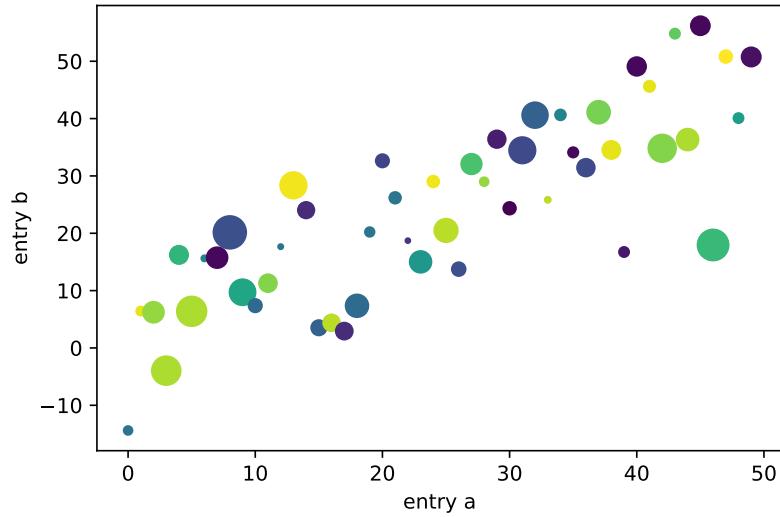


A.5.1.2 Plotting with keyword strings

There are some cases in which the format of the data lets accessing particular variables with strings. With the Matplotlib library, you can skillfully plot them. There is a sample demonstrating how it is carried out.

```
data = {'a': np.arange(50),
        'c': np.random.randint(0, 50, 50),
        'd': np.random.randn(50)}
data['b'] = data['a'] + 10 * np.random.randn(50)
data['d'] = np.abs(data['d']) * 100

plt.scatter('a', 'b', c='c', s='d', data=data)
plt.xlabel('entry a')
plt.ylabel('entry b')
plt.show()
```



A.5.1.3 Plotting with categorical variables

It is also possible to create a plot using categorical variables. Matplotlib allows us to pass categorical variables directly to different plotting functions.

```

names = ['group_a', 'group_b', 'group_c']
values = [1, 10, 100]

plt.figure(figsize=(9, 3))

plt.subplot(131)
plt.bar(names, values)
plt.subplot(132)
plt.scatter(names, values)
plt.subplot(133)
plt.plot(names, values)
plt.suptitle('Categorical Plotting')
plt.show()

```

