Facultad de Ciencias Económicas y Empresarial

Doble Grado en ADE y Business Analytics (E-2 Analytics)

AutoKAN: The power of Kolmogorov-Arnold Networks and Bayesian optimization.

Author: Adriana Fernández de Caleya Vázquez
Director: Eduardo César Garrido Merchán

Clave: 202005333

MADRID | Abril 2025

Abstract

In recent years, Kolmogorov-Arnold Networks (KANs) have emerged as a promising alternative to traditional neural networks, offering a more interpretable and potentially more efficient approach to function approximation. However, tuning KANs remains a complex challenge due to their unique architecture and sensitivity to hyperparameter configurations. This project proposes AutoKAN, a custom Python class that combines KANs with Bayesian optimization in order to automate the search for optimal configurations.

To evaluate its effectiveness, AutoKAN is benchmarked against three baseline models: a default KAN, linear regression, and random forest. A structured experimental methodology is applied using both synthetic and real-world datasets for regression tasks. The results suggest that while AutoKAN underperforms in comparison to simpler models, this outcome is primarily due to technical limitations in the optimization process and the complexity of the KAN hyperparameter space. These findings highlight the importance of designing customized optimization strategies for novel model architectures and point toward potential directions for future work.

Keywords: Kolmogorov-Arnold Networks, AutoKAN, Bayesian Optimization, Regression, Hyperparameter Tuning, Model Benchmarking.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing dependence on artificial intelligence (AI) highlights the need for models that are both efficient and interpretable, particularly when handling complex, high-dimensional data. Traditional models like Multilayer Perceptrons (MLPs), while foundational, often struggles with these challenges. These limitations, combined with computational inefficiencies, can lead to issues such as overfitting, which reduces the reliability of predictions in critical applications (Li et al., Dec 2019). Furthermore, many machine learning models are viewed as "black boxes", offering little transparency in their decision-making processes. This lack of interpretability becomes problematic in industries where decisions must be explained and trusted, such as in finance and healthcare, where stakeholders require clear reasoning behind AI-driven decisions (ROCHA et al., 2012).

Conventional models also face difficulties in high-dimensional environments, where tuning hyperparameters can be time-consuming and inefficient (Lu & Zhan, 2024). These challenges underline the need for new approaches that can improve both efficiency and interpretability. This research is motivated by the opportunity to explore innovative architectures, such as Kolmogorov-Arnold Networks (KANs), which address both these issues by providing models that are more transparent and computationally effective (Chen, 2024). By enabling clearer decision-making and better management of complex data, KANs are crucial for building trust among stakeholders like developers and business users.

Unlike traditional “black box” models, KANs not only make the decision-making process more understandable but also reduce the computational load, making them ideal for solving real-world tasks (Liu et al., 2024b). Their ability to handle high-dimensional, nonlinear data positions them as a strong candidate for improving AI performance across various fields, offering both developers and end-users more trustworthy and reliable outcomes.

In addition to these advantages, the integration of Bayesian optimization strengthens this approach by addressing one of the key challenges in machine learning: hyperparameter tuning. In high-dimensional settings, traditional tuning methods are often computationally expensive and time-consuming, which can slow down the development and implementation of AI models. Bayesian optimization provides a more efficient solution by using probabilistic models to systematically explore the parameter space and identify optimal configurations with fewer evaluations (Daulton et al., 2021). This not only accelerates the model development process but also ensures more robust performance, particularly when paired with KANs. By improving the accuracy and efficiency of hyperparameter tuning, Bayesian

optimization allows KANs to achieve their full potential, reducing the costs and time required to deploy high-performing AI systems.

By integrating Bayesian optimization, KANs offer a promising solution to many limitations in traditional AI models. This research will explore these advancements through theoretical analysis and practical applications, highlighting how KANs can boost the development of more transparent and efficient AI systems.

The thesis is structured as follows: first, a comprehensive state of the art is presented, reviewing the evolution of neural networks, the theoretical foundations of KANs, and their applications. This is followed by the thesis definition, which defines the research objectives, hypotheses, assumptions, and constraints. Next, the methodology outlines the experimental framework and evaluation metrics employed to compare the performance of AutoKAN with the default KAN architecture. The analysis of results examines the findings in depth, highlighting the differences and improvements observed between these models. Finally, the conclusions synthesize the insights gained and outline the research's contributions and potential future directions.

Chapter 2

State of the Art Review

Neural networks have evolved significantly since their origin in the mid-20th century. The earliest models, such as the perceptron developed by Rosenblatt in 1957, aimed to mimic the brain's ability to learn and classify patterns, laying the foundation for modern neural network research (Wason, 2018). The perceptron introduced the idea of a simple, single-layer network capable of binary classification, but its limitations became evident, particularly in handling non-linearly separable data. Interest in neural networks temporarily declined during the late 1960s, a period often referred to as the "AI winter" (Toosi et al., 2021).

However, renewed interest in neural networks emerged in the 1980s with the development of MLPs, which utilized the backpropagation algorithm for training. This breakthrough allowed neural networks to handle more complex, multi-class problems by adjusting weights iteratively (Schmidhuber, 2015). The introduction of backpropagation marked the beginning of a new era, where neural networks, now with multiple layers, could solve problems that earlier models could not, such as image and speech recognition (Wason, 2018).

The 1990s and early 2000s saw further advancements, especially with the introduction of Convolutional Neural Networks (CNNs) and Recurrent Neural Networks (RNNs). CNNs, introduced by Fukushima in 1979 and later refined by LeCun in the late 1980s, transformed computer vision by enabling networks to identify patterns and reduce data size through layers (Schmidhuber, 2015). RNNs, designed for tasks like speech recognition, excelled at modeling temporal dependencies by maintaining memory across inputs (Schmidhuber, 2015). The adoption of deep learning models was significantly accelerated by the rise of large datasets and improved computing power, such as the use of Graphics Processing Units (GPUs) in the mid-2000s. These advancements allowed neural networks to show remarkable performance improvements across diverse tasks including image, video, and natural language processing (Wason, 2018). Innovations such as unsupervised pre-training and new optimization techniques further drove these models to achieve human-level performance in domains such as image recognition and strategic game playing (Schmidhuber, 2015).

Today, neural networks are the foundation of AI applications, with architectures like CNNs and RNNs being widely used across many fields, from healthcare and autonomous driving to finance. Their ability to adapt to diverse challenges underlines the importance in modern AI. However, as AI faces increasingly complex and high-dimensional data, traditional models are showing limitations in terms of efficiency and scalability. These challenges have led to the exploration of new architectures, like KANs, to address these limitations.

KANs are a recent innovation in neural networks, offering an alternative to traditional models like MLPs. Their main advantage lies in how to handle complex, nonlinear relationships in data. KANs are based on the Kolmogorov-Arnold representation theorem, which states that any continuous multivariate function can be broken down into a finite sum of univariate functions (Cambrin et al., 2024). Unlike traditional models, where activation functions are applied at the nodes, KANs introduce learnable activation functions directly on the edges between nodes, gaining more flexibility (Liu et al., 2024a). This new approach allows KANs to better manage nonlinear relationships in high-dimensional settings. This structure allows KANs to manage complex data without significantly increasing model complexity (Pourkamali-Anaraki, 2024). This shift in architecture opens new possibilities for improving both accuracy and interpretability.

One of the key advantages of KANs is their ability to adapt to different types of data, particularly in high-dimensional environments. MLPs and other conventional models typically rely on fixed activation functions, which restricts the model's ability to adapt to the specific characteristics of the data being processed (Cacciatore et al., 2024). KANs address this problem by introducing learnable functions along the edges, enabling the network to adjust dynamically based on input data. This flexibility makes KANs both efficient and scalable, as they require fewer resources while achieving more accurate results (Azam & Akhtar, 2024).

In addition to being computationally efficient, KANs offer significant advantages in terms of interpretability, which is critical in many AI applications. Traditional models, particularly deep learning architectures, are often criticized for their "black box" nature, where the decision-making process is unclear and difficult to interpret (Liu et al., 2024a). KANs address this by providing more transparency, making it easier to understand and analyze the contribution of individual variables. This transparency makes KANs ideal for applications where both accuracy and explainability are required (Sulaiman et al., 2024).

Despite their relatively recent introduction, KANs have already shown promising results in a variety of applications. For instance, in energy consumption prediction, KANs have outperformed traditional models by achieving higher accuracy while maintaining computational efficiency (Sulaiman et al., 2024). Similarly, in tasks like image segmentation and recognition, KAN-based architecture has demonstrated superior performance compared to conventional neural networks (Azam & Akhtar, 2024). These successes highlight the flexibility of KANs across different domains and their potential to become a foundational technology in AI development. As research continues, KANs are expected to evolve into even more advanced models, addressing long-standing challenges in neural networks, such as the curse of dimensionality and the need for improved interpretability (Pourkamali-Anaraki, 2024).

KANs have also shown notable improvements in both accuracy and adaptability when compared to traditional models like MLPs and CNNs. For example, in financial tasks such as option pricing, KANs have outperformed MLPs and Time-Delay Neural Networks (TDNNs), providing more accurate predictions of European call options. This advantage comes from KANs' ability to capture complex, nonlinear relationships in data, something traditional models often struggle with, especially in high-dimensional settings (Ter-Avanesov & Beigi, 2024). While traditional models are inclined to issues like overfitting, KANs manage these challenges more effectively by optimizing their structure and reducing unnecessary model complexity (Cacciatore et al., 2024). Figure 1 illustrates these structural differences, highlighting how KANs place activation functions on edges, unlike the node-based activations in MLPs.

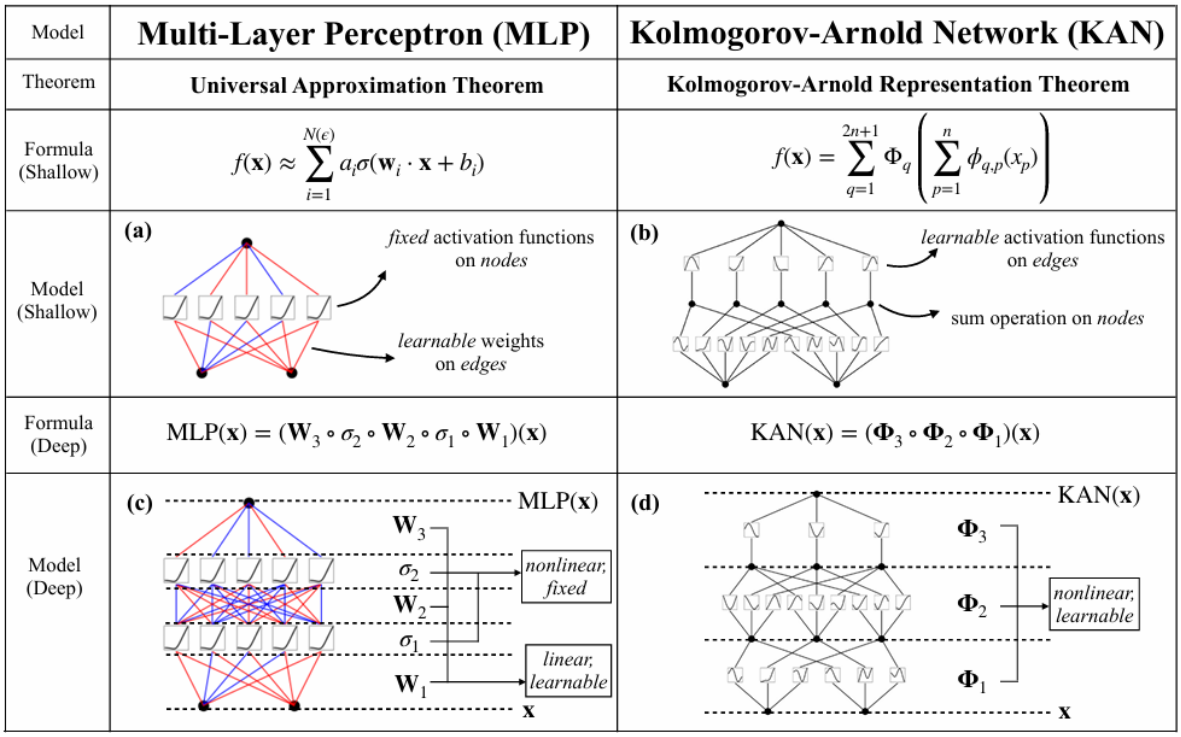


Figure 2.1: Comparative Architectures of MLPs and KANs.

Beyond financial tasks, KANs also demonstrate strong potential in Natural Language Processing (NLP). While RNNs and Long Short-Term Memory networks (LSTMs) are effective for sequential data, they are limited by fixed activation functions that restrict adaptability. In contrast, KANs' learnable activation functions allow them to adjust dynamically, making them particularly effective in tasks like machine translation and sentiment analysis. This adaptability not only improves performance but also enhances the interpretability of the model's decisions, which is an important factor in NLP applications (Zohuri & Moghaddam, 2020).

KANs have also demonstrated superior performance in image processing tasks. When compared to CNNs, which are widely used in computer vision, KANs show a more advanced ability to handle the complexity of high-dimensional image data. CNNs, while powerful, often require large datasets and can struggle with overfitting when data is limited. KANs, with their adaptable architecture, overcome this issue by adjusting more effectively to the data, leading to better results in tasks like image classification and segmentation (Pourkamali-Anaraki, 2024). Additionally, KANs provide a clearer understanding of how different features contribute to the final decision, making them especially useful in areas like healthcare and autonomous driving, where transparency and accountability are essential (Sulaiman et al., 2024).

KANs' flexibility has also been demonstrated in the energy sector. For example, KANs have been successfully applied to predict chiller energy consumption in commercial buildings, outperforming conventional models such as Artificial Neural Networks (ANNs) and hybrid deep learning algorithms. In this application, KANs were able to capture the complex nonlinear dynamics of energy consumption, achieving higher accuracy with a reduced error rate (Sulaiman et al., 2024). Figure 2 below illustrates the comparative performance of KANs and traditional neural networks across different settings and parameter configurations, emphasizing the robustness and efficiency of KANs in handling predictive tasks in the energy sector.

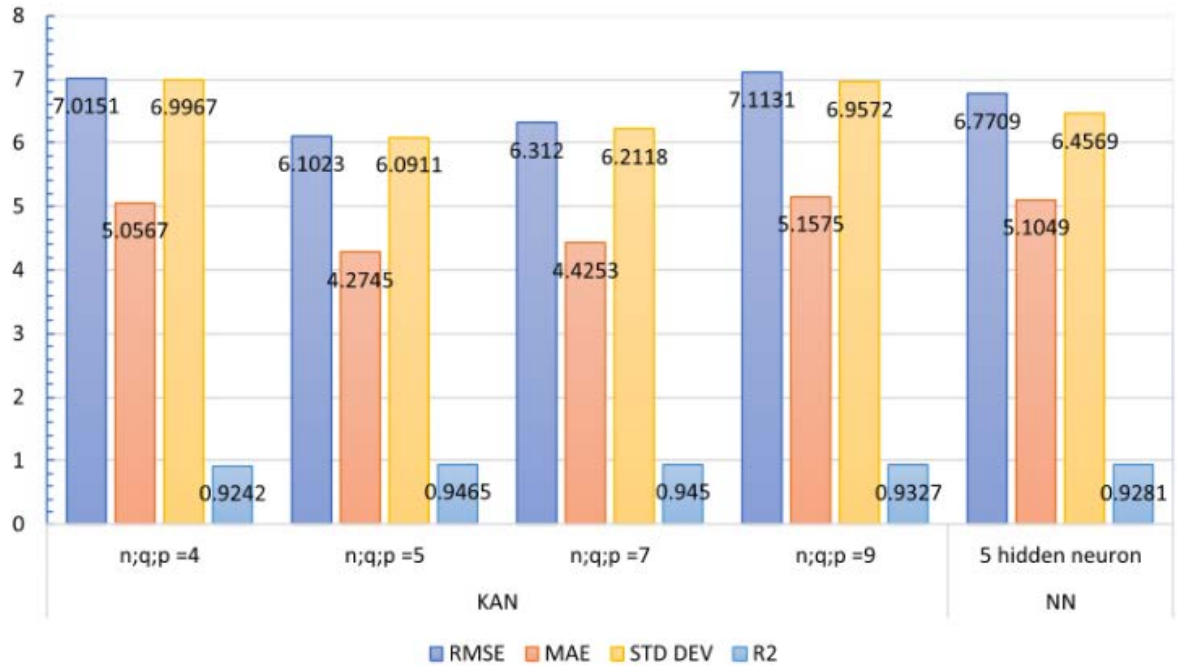


Figure 2.2: Performance of KANs compared to traditional neural networks with 5 hidden neurons across various metrics such as RMSE, MAE, STD DEV, and R2.

Such empirical evidence supports the superior capability of KANs in complex applications. This higher level of accuracy is essential for optimizing energy usage and reducing operational costs, which are vital objectives in commercial building management. By modeling energy consumption patterns more effectively than other models, KANs enable better decision-making in energy management, making them a valuable tool in the pursuit of sustainability.

KANs' versatility extends beyond the energy sector to agriculture. In this field, KANs have proven to be quite helpful in segmenting crop fields using satellite images. When integrated into U-Net architectures (U-KAN), KANs were used to process data from two satellites for precise crop field segmentation. Compared to traditional U-Net models, U-KANs showed a 2% improvement in Intersection-Over-Union (IoU), a measure of segmentation accuracy, while requiring fewer numerical calculations, making them both more accurate and computationally efficient (Cambrin et al., 2024). This improvement in IoU is crucial for farmers and agricultural analysts, as it allows for better monitoring of crop health and more efficient management of resources like water and fertilizers. Furthermore, the transparency of KANs makes them an excellent choice for real-world agricultural applications where clear, explainable predictions are key for guiding sustainable practices.

KANs have also demonstrated significant promises in healthcare, especially in predictive diagnostics. By incorporating Bayesian inference, Bayesian-KANs (BKANs) not only enhance the accuracy of medical diagnostic predictions but also bring much-needed interpretability to these predictions. For example, in applications ranging from cancer detection to predictive patient outcomes, BKANs have demonstrated superior accuracy over traditional deep learning models by effectively quantifying the uncertainty of predictions, which is crucial for making informed medical decisions (Hassan, 2024). This dual capability of providing high accuracy and clear interpretability makes BKANs invaluable in healthcare, where they support critical decision-making with reliable, transparent data.

Building on the success of KANs, new versions like BKANs and Gaussian Process-KANs (GP-KANs), have been developed to extend the capabilities of KANs. BKANs use Bayesian optimization to help the model adjust its parameters more efficiently, allowing them to provide more confident predictions, which is especially important in areas like healthcare and finance, where accurate and trustworthy predictions are necessary (Li et al., Dec 2019). This makes BKANs better at handling complex data while minimizing issues like overfitting (Hassan, 2024).

Another key development is GP-KANs, which incorporate Gaussian Processes to better manage non-linear patterns in data. GP-KANs are particularly useful in tasks where it is important to estimate how confident a model is, such as image recognition or noisy data environments. For instance, in tasks like classifying images from the MNIST dataset, GP-KANs have shown high accuracy while using fewer parameters, making them more efficient

than traditional models (Chen, 2024). These improvements allow KANs to be more flexible and effective in a wide range of applications, addressing issues like overfitting and offering clearer, more reliable predictions (Pourkamali-Anaraki, 2024).

While KANs have demonstrated promise across various applications, several challenges remain. One significant issue is the susceptibility to noise in the training data. Even small amounts of noise can drastically reduce their performance, especially when modeling complex, high-dimensional data (Shen et al., 2024). To mitigate this, techniques like kernel filtering and oversampling have been proposed, though these methods come with trade-offs, like the need for a larger dataset, which increases the computational cost (Shen et al., 2024). Finding the right balance between noise reduction and efficiency is still an active area of research.

Another challenge is scaling KANs for more complex tasks. Although KANs are flexible in modeling nonlinear relationships, they often require more complex architectures to handle certain types of data, like images with significant variation (Azam & Akhtar, 2024). Researchers are currently exploring ways to simplify KAN structures without sacrificing accuracy by combining KANs with other neural networks or applying more efficient optimization methods. These efforts reflect the ongoing development of KANs as researchers aim to enhance computational efficiency while handling noisy or diverse data (Shen et al., 2024).

Table 2.1: Key References in Neural Networks and KAN Research.

Key Contributor	**Findings**
(Schmidhuber, 2015)	Discussed the introduction of backpropagation in the 1980s, which revolutionized neural networks by enabling MLPs to solve complex problems. Also highlighted advancements in CNNs and RNNs during the 1990s and 2000s, which transformed image recognition and temporal modeling tasks.
(Wason, 2018)	Discussed the perceptron, developed by Rosenblatt in 1957, as a foundational model for neural networks, capable of binary classification. Explained how GPUs and large datasets in the mid-2000s accelerated neural network adoption, enabling breakthroughs in image, video, and NLP tasks.
(Li et al., 2019)	Proposed Bayesian-KANs (BKANs), which utilize Bayesian optimization for parameter adjustment, achieving more confident and accurate predictions. Demonstrated BKANs' superior performance in healthcare and finance tasks, minimizing overfitting and addressing complex data challenges.
(Zohuri & Moghaddam, 2020)	Highlighted the limitations of traditional RNNs and LSTMs in sequential data tasks due to their fixed activation functions, which restrict adaptability. Demonstrated how KANs' learnable activation functions improve performance and interpretability in NLP tasks like machine translation and sentiment analysis.
(Toosi et al., 2021)	Highlighted the "AI winter" of the late 1960s due to limitations in early neural network models, which temporarily decreased interest in the field.
(Azam & Akhtar, 2024)	Demonstrated KANs' success in image segmentation tasks, outperforming CNNs with fewer data requirements and improved interpretability.
(Cacciatore et al., 2024)	Compared KANs with traditional models like MLPs and CNNs, showing their advantages in handling overfitting and improving interpretability.
(Cambrin et al., 2024)	Highlighted KANs' use of the Kolmogorov-Arnold theorem for representing multivariate functions, improving flexibility and efficiency in high-dimensional data.

(Chen, 2024)	Explored GP-KANs for managing non-linear data patterns, showcasing improved efficiency and confidence estimation in tasks like image recognition.
(Hassan, 2024)	Introduced Bayesian-KANs (BKANs), which combine KAN architectures with Bayesian inference to enhance accuracy and uncertainty quantification in medical diagnostics.
(Liu et al., 2024a) and (Liu et al., 2024b)	Proposed learnable activation functions on KAN edges to address nonlinear relationships in data, enhancing model adaptability.
(Pourkamali-Anaraki, 2024)	Highlighted KANs' scalability and reduced complexity, making them effective in high-dimensional and diverse data settings.
(Shen et al., 2024)	Discussed challenges faced by KANs, such as sensitivity to noise and scalability issues, along with proposed solutions like kernel filtering and oversampling.
(Sulaiman et al., 2024)	Showcased KANs' superior performance in energy consumption prediction, offering computational efficiency and higher accuracy compared to traditional models.
(Ter-Avanesov & Beigi, 2024)	Demonstrated that KANs outperform MLPs and Time-Delay Neural Networks (TDNNs) in financial tasks such as option pricing, providing more accurate predictions for European call options by capturing complex, nonlinear relationships in high-dimensional data.

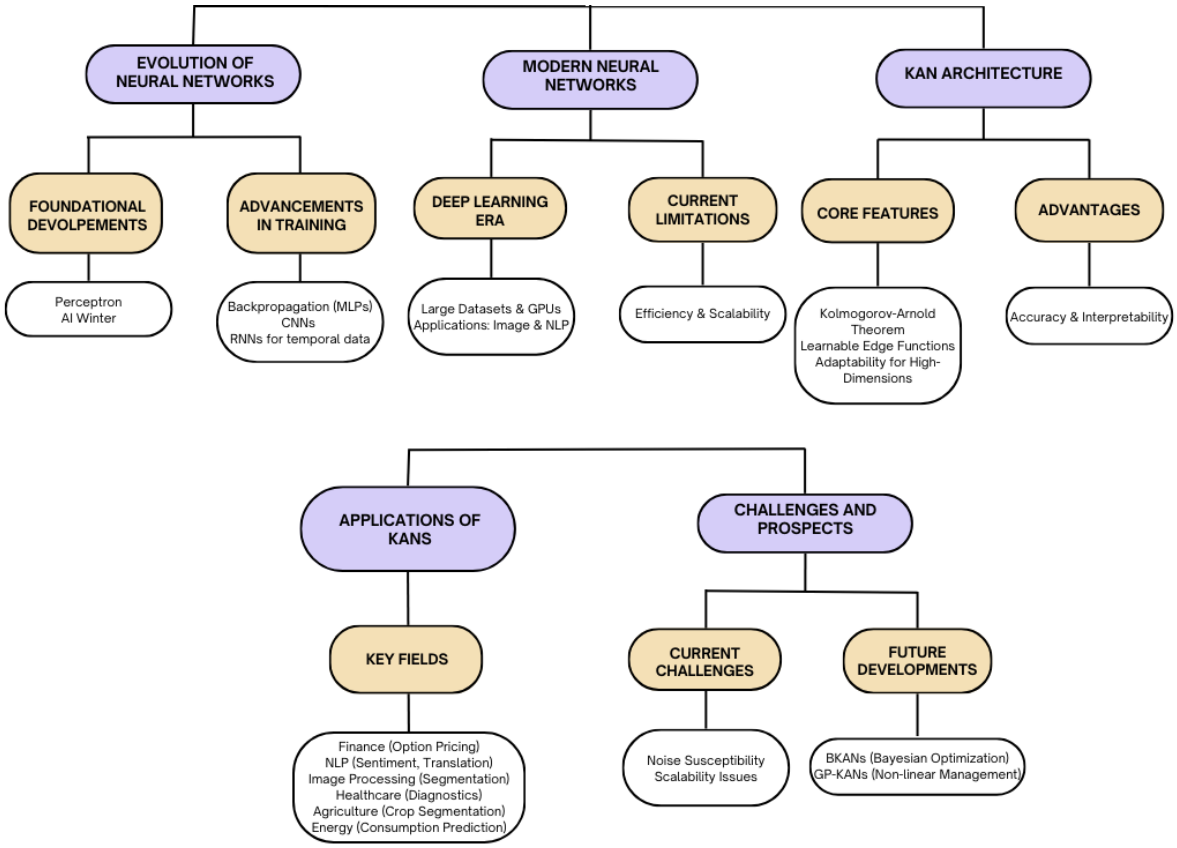


Figure 2.3: Organization of the State-of-the-art review

Chapter 3

Thesis definition

This chapter defines the scope and framework of the thesis, which centers on the development, testing, and evaluation of AutoKAN, a model designed to automate the optimization of Kolmogorov-Arnold Networks' (KANs) hyperparameters through Bayesian optimization. The discussion will also outline the specific goals, constraints, hypotheses, and underlying assumptions that are crucial to guiding the research process, setting the stage for a comprehensive exploration of AutoKAN's potential to boost machine learning practices.

3.1 General Objective

The general objective of this thesis is to evaluate the efficacy of AutoKAN in comparison to default KAN models. This includes a detailed investigation into the theoretical foundations of KANs, a performance assessment against the default configurations, and an exploration of AutoKAN's practical applications in various real-world scenarios. The aim extends to demonstrating how enhancements provided by AutoKAN, such as improved efficiency, accuracy, and broader applicability, can significantly advance the field of neural networks. The specific objectives are as follows:

3.2 Specific Goals

1. Document the Development Process: Write a technical report detailing the design and development of the AutoKAN class in Python, including its integration with Bayesian optimization techniques and the theoretical foundation of KANs.

2. Implementation of AutoKAN: Implement the AutoKAN class, ensuring its ability to initialize, configure, and optimize KANs using Bayesian optimization.

3. Empirical Benchmarking: Conduct benchmark experiments to statistically compare the performance of AutoKAN against default KAN configurations. Evaluate performance across metrics such as accuracy, computational efficiency, and resource utilization in various scenarios.

4. General Framework for Application: Propose a framework for applying AutoKAN in real-world scenarios, emphasizing its computational efficiency and interpretability.

3.3 Constraints

1. Time Limitation: The thesis is to be completed within the current academic year 2024-2025, which may limit the extent of iterative testing or long-term performance analysis.

2. Budgetary Constraints: With no external funding available, the scope of empirical testing, dataset acquisition, and access to advanced computational resources will be limited.

3. Resource Availability: All computations will be performed using Python 3 on a personal computer, which may limit the complexity of models tested and the speed of simulations and optimizations.

4. Data Availability: Due to the above constraints, the study used pre-cleaned and relatively small benchmark datasets. While these are widely used for testing regression models, they may not fully represent the complexity and heterogeneity of real-world data in more demanding applications.

3.3.1 Hypothesis

In this research, a statistical comparison of AutoKAN's performance in optimizing KANs against the default KAN configuration will be provided. The hypothesis to be tested suggests that the optimized KANs using AutoKAN will outperform the default KANs in terms of accuracy and efficiency. Therefore, the hypothesis is framed as follows:

Null Hypothesis: H_0: $\mu_{AUTOKAN} \leq \mu_{DEFAULTKAN}$

Alternative Hypothesis: H_1: $\mu_{AUTOKAN} > \mu_{DEFAULTKAN}$

3.3.2 Assumptions

1. Data Representativeness: It is assumed that the datasets used in this study are representative of real-world scenarios, allowing the results to be extrapolated to other applications where KANs may be deployed.

2. Model Stability: It is assumed adding additional variables to the models does not significantly alter the outcomes, indicating robustness in the face of data variability.

3. Underlying Theory: The fundamental principles of the Kolmogorov-Arnold representation theorem and neural network theory that underpin KANs are assumed to be valid and applicable in the contexts tested in this study.

4. System Reliability: Python-based implementation and all dependent libraries function as intended without major errors affecting the experimental results.

Chapter 4

Methodology

The main objective of this work is to develop and implement AutoKAN, a custom Python class capable of initializing and optimizing Kolmogorov-Arnold Networks using Bayesian optimization. The effectiveness of this solution will be compared against default KAN configurations as well as other benchmark models such as logistic regression and random forest.

This process is structured into clearly defined phases, which will be developed and analyzed in detail in Chapter 5:

- **Data Selection:** The first phase involves selecting suitable datasets for both the initial development and the final benchmarking of the models. The datasets used will have varied characteristics to test the generalizability of the models. The objective is to work with data that is both diverse and standardized, enabling meaningful evaluation across different models. Datasets like *make_regression* are used initially for model development as a synthetic dataset, followed by real-world datasets like the Boston Housing, Diabetes, and California Housing datasets, which offer more complexity and allow testing in realistic conditions.

- **Descriptive Data Analysis:** Before building any models, it is important to understand the underlying structure and distribution of each dataset. This step provides insights into variable relationships and potential modeling challenges, helping to contextualize the results obtained afterwards.

- **Models Design:** In this section, the architecture of AutoKAN is introduced. The AutoKAN class is designed to automate the initialization and optimization of KANs using Bayesian optimization. This class handles tasks such as splitting datasets into training and testing sets, converting data into PyTorch tensors, and performing hyperparameter optimization using the *gp_minimize* function.

  The core of AutoKAN is based on the Bayesian optimization method, which aims to minimize the error function (in this case, mean squared error) by adjusting the hyperparameters of the KAN. A set of hyperparameters, such as the number of layers,

neurons per layer, splines per neuron, and the spline degree, are tuned during optimization to find the best configuration for the task at hand.

Mathematical formulation of the objective function

The objective function *f(x)* to minimize is the MSE calculated during the k-fold cross-validation:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$$

Where:

- N is the number of data points in the test set.
- $\hat{y}_i$ is the predicted value for the *i*-th data point.
- y_i is the actual value for the *i*-th data point.

Bayesian optimization, through *gp_minimize*, is used to find the optimal set of hyperparameters by evaluating this function iteratively. The default KAN and benchmark models, such as random forest and logistic regression, are evaluated under the same conditions to ensure fair comparison.

- **Benchmarking:** Once the AutoKAN class and the other models (like the default KAN, random forest, and linear regression) are implemented, they are evaluated using the mean squared error (MSE) as the performance metric. The benchmarking process involves comparing these models across the selected datasets to evaluate their generalization and performance. By comparing the error values obtained for each model, I aim to assess the relative strengths and weaknesses of AutoKAN in comparison to simpler, explainable models like random forest and linear regression.

- **Evaluation of Results:** In the final step, the results of the benchmarking phase are analyzed and interpreted. Beyond simply comparing scores, this section reflects on the behavior of AutoKAN under different conditions, highlights key findings, and discusses technical limitations that may have influenced the outcomes.

Chapter 5

Implementation and Results

This chapter presents the methodology followed throughout the experimental phase of this study, along with the analysis of the obtained results. The goal is to evaluate the performance of the AutoKAN model optimized via Bayesian optimization and compare it against a baseline KAN, a random forest, and a linear regression model.

The section begins with the selection and description of the dataset used for benchmarking, followed by a brief exploratory analysis to better understand its structure. Then, the design of the models is detailed, including both the custom AutoKAN class and the implementation of the three benchmark models. Afterward, the benchmarking setup is described, specifying how the experiments were conducted and how the models were compared. Finally, the results of the experiments are evaluated and discussed, providing insights into the relative performance of each method and highlighting potential limitations.

5.1 Data Selection

The experimental evaluation in this project is based on three datasets, each serving a different purpose in the overall development and testing of the models.

- **Synthetic dataset**: This dataset was used during the initial stages of code development and refinement, particularly for building and testing the AutoKAN class. The *make_regression* function from *sklearn.datasets* generates a linear problem with continuous outputs and controlled noise. Since it produces perfectly clean data when noise is zero, it is ideal for validating the basic functionality of the model before transitioning to real-world data. However, due to its simplicity, it is not suitable for assessing generalization.

- **Diabetes dataset**: This real-world dataset, also from *sklearn.datasets*, is commonly used as a benchmark in regression tasks. It contains ten baseline variables (like age, sex, BMI, blood pressure, or blood serum measurements) and a quantitative measure of disease progression one year after baseline.

- **Boston Housing dataset**: This dataset, also commonly used for regression tasks, includes 13 features that describe various aspects of residential homes in Boston suburbs. It is accessed through OpenML using the *fetch_openml* function. Its moderate size and real-world nature make it an ideal candidate for comparing model performance without incurring long training times.

These datasets were selected due to their simplicity, accessibility, and widespread use in benchmarking regression algorithms. They are clean and standardized, which eliminates the need for extensive preprocessing and ensures that the performance differences observed can be attributed mainly to the models and not to data quality issues.

Because of this, no specific preprocessing techniques (e.g., normalization, encoding, or imputation) have been applied. This decision is justified by the fact that all three datasets come from the *sklearn* library and are already well-prepared for regression tasks.

5.2 Descriptive Data Analysis

Before delving into model design and training, it was essential to perform an exploratory analysis of the datasets selected for experimentation. This step provides preliminary insights into the distributional properties of the data, relationships among features, and the behavior of the target variable.

For the synthetic dataset, which was generated with one thousand samples, five features and a noise level of 30, a histogram of the target variable was plotted to analyze its distribution (Figure 5.1). As expected, the distribution remains approximately normal, although the introduction of noise increased the dispersion and slightly flattened the peak. This validates that the dataset maintains the general structure of a linear regression problem, while adding enough variability to challenge the models.

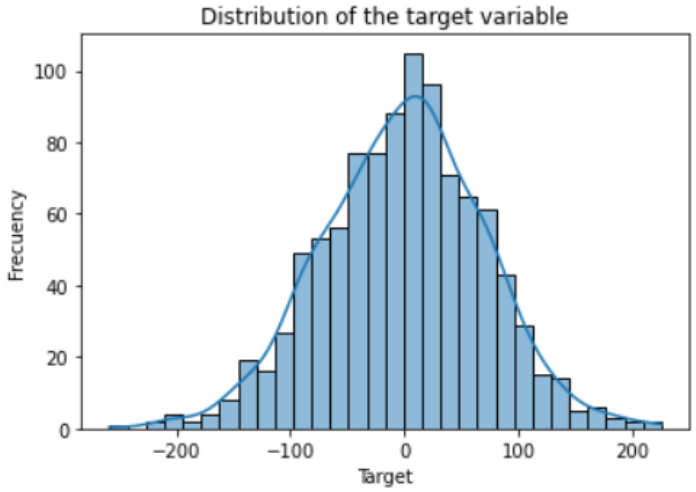


Figure 5.1: Histogram from the synthetic dataset.

The Boston Housing dataset, in contrast, presents a more complex and realistic scenario. It contains 506 observations, and 13 variables related to housing conditions in the Boston area. According to the summary statistics of the target variable (Table 5.1), which represents the median value of owner-occupied homes in thousands of dollars, the values range from 5 to 50 with a mean of approximately 22.53. The histogram in Figure 5.2 shows a slightly skewed distribution, with most house prices concentrated between 18 and 25.

Summary Statistics	
count	506.000000
mean	22.532806
std	9.197104
min	5.000000
25%	17.025000
50%	21.200000
75%	25.000000
max	50.000000

Table 5.1: Summary Statistics.

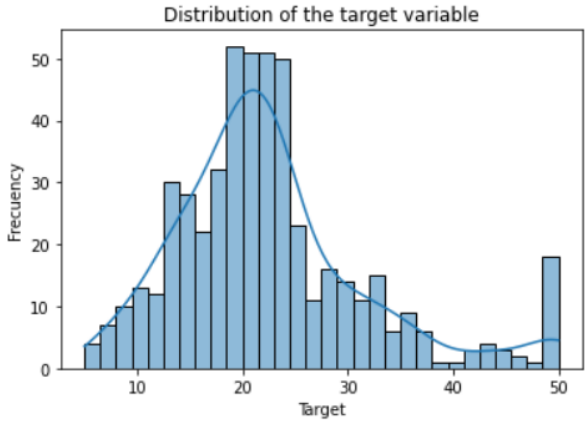


Figure 5.2: Histogram from the Boston Housing dataset.

Additionally, the correlation matrix (Figure 5.3) reveals several meaningful relationships between features and the target variable. For example, there is a strong positive correlation between the number of rooms per dwelling (Feature 5) and housing prices, which is consistent with expectations. Conversely, the percentage of lower-status population (Feature 12) shows a strong negative correlation with the target, indicating that areas with a higher proportion of disadvantaged individuals tend to have lower house prices. Notably, the highest

correlation is observed between the accessibility to radial highways (Feature 8) and the property tax rate (Feature 9), which also aligns with economic intuition. These patterns suggest the presence of multicollinearity and nonlinear dependencies that could complicate the performance of simpler models.

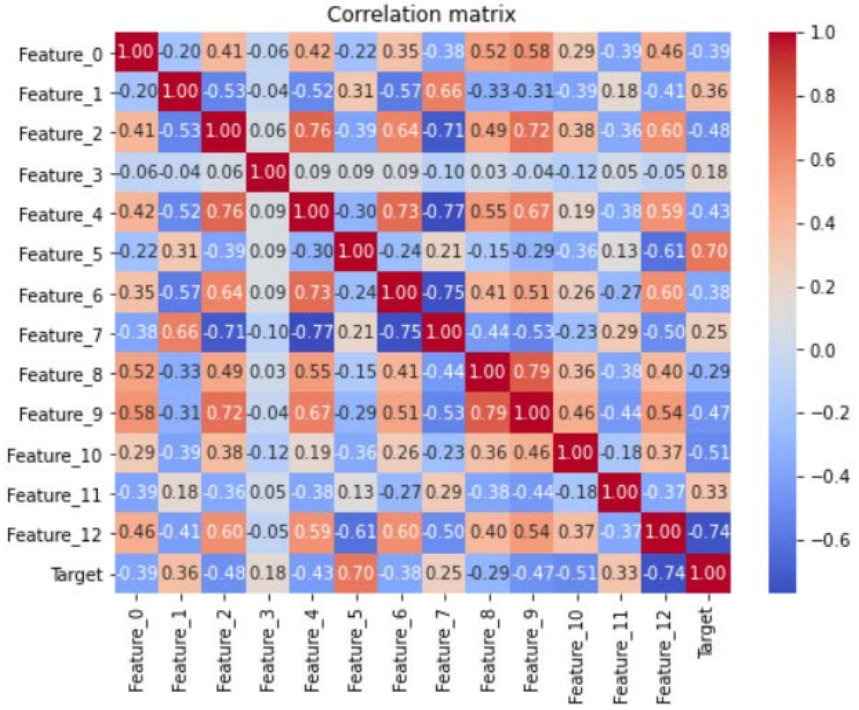


Figure 5.3: Correlation matrix from the Boston Housing dataset.

Lastly, the Diabetes dataset includes 442 samples and 10 continuous features measuring various physiological variables and baseline measurements such as age, sex, BMI, blood pressure, and different blood serum metrics. The target variable indicates a quantitative measure of disease progression one year after baseline. As summarized in Table 5.2, the target variable ranges from 25 to 346, with a mean of 152 and a relatively wide standard deviation, indicating high dispersion in disease progression. The histogram in Figure 5.4 shows a right-skewed distribution, suggesting that most patients fall into the lower range of disease progression, but a few cases present much higher severity.

Summary Statistics

count	442.000000
mean	152.133484
std	77.093005
min	25.000000
25%	87.000000
50%	140.500000
75%	211.500000
max	346.000000

Table 5.2: Summary Statistics

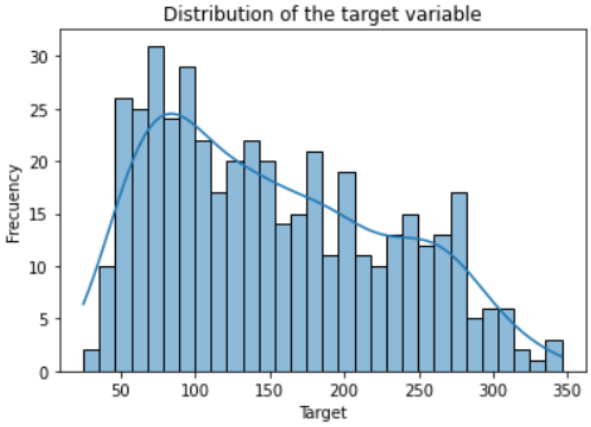


Figure 5.4: Histogram from the Diabetes dataset.

In the correlation matrix (Figure 5.5), I can observe that BMI (Feature 2) and certain blood serum metrics (Feature 8) show strong positive correlations with the target, aligning with medical expectations that higher BMI or specific blood markers may signal greater disease severity.

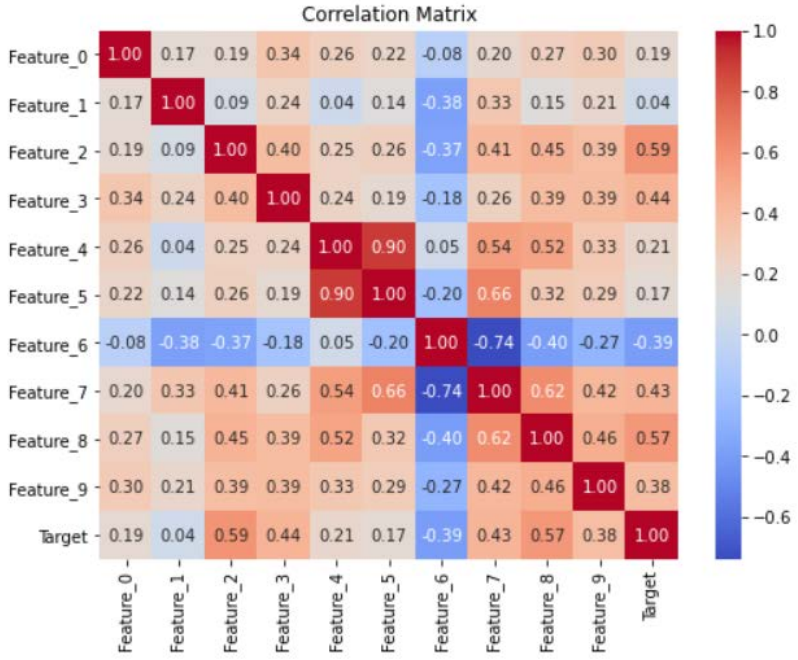


Figure 5.5: Correlation matrix from the Diabetes dataset.

Overall, this descriptive analysis confirms the diversity and structure of the selected datasets. The synthetic dataset offers a controlled environment, while the Boston and Diabetes datasets provide richer, real-world complexity, which will allow for robust evaluation and comparison of the models.

5.3 Model Design

5.3.1 AutoKAN Class Implementation

The core of this project revolves around the development of the AutoKAN class, a Python implementation designed to automate the configuration and training of KANs using Bayesian optimization. The objective was to create a flexible and extensible system capable of identifying the optimal hyperparameter configuration for a given regression task, improving the standard KAN configuration in terms of accuracy and generalization.

The class receives the input features X and target variable y, along with optional parameters such as the search space, number of Bayesian optimization iterations, and random seed. Upon initialization, the dataset is split into training and testing sets (80/20), which are then converted into *PyTorch* tensors. These tensors are loaded into *DataLoaders* and later concatenated into unified training and testing tensors that are stored in a structured dictionary compatible with the KAN library. Additionally, the code ensures that all data is moved to GPU if available, defaulting to CPU otherwise.

A key component of the implementation is the Bayesian optimization itself, which is carried out using the *gp_minimize* function. This function iteratively evaluates possible hyperparameter combinations based on the performance of the model, attempting to minimize the test error. However, during early testing, it was observed that some model evaluations returned NaN values which caused the entire optimization process to crash. Initially, various potential fixes were explored: avoiding automatic model saving, altering the seed at every iteration, changing the model directory path, and even reloading the data at each step. None of these worked reliably.

Ultimately, a fallback solution was introduced by wrapping the model training in a *try-except* block and assigning a large error value (1.0) when the output was NaN. This patch ensured the optimization process could continue even if some configurations failed, which was crucial to allow experimentation to proceed.

Another critical adjustment involved simplifying the hyperparameter search space. It was found that overly complex architectures were more likely to cause instabilities or return NaN errors. Therefore, to improve stability and ensure Bayesian optimization could successfully explore viable configurations, the upper bounds for each parameter were reduced. This included limiting the number of hidden layers, the number of neurons per layer, the splines per neuron and the degree of splines.

Additionally, the number of Bayesian optimization iterations was set to just 10, a deliberately low value chosen for computational feasibility. Exploring high-dimensional hyperparameter spaces typically requires a large number of evaluations to converge to optimal regions, but due to the runtime and instability of certain KAN configurations, a smaller number was selected to balance practicality with effectiveness. While suboptimal in terms of search efficiency, this choice was suitable for illustrative purposes.

Together, these design decisions allowed AutoKAN to function reliably within the constraints of the environment, enabling experimental comparison with other baseline models.

5.3.2 Benchmark Models Implementation

To evaluate the performance of AutoKAN, I implemented three benchmark models: a linear regression model, a random forest regressor, and a default KAN configuration. All models were trained and evaluated using the same train-test split to ensure a fair comparison.

The linear regression and random forest models were implemented using the *scikit-learn* library. Their implementation was straightforward: the dataset was split into training and testing sets, and the models were trained using their respective *.fit()* methods. For the random forest, I used 100 estimators and a fixed random state for reproducibility.

The default KAN model required a slightly more elaborate setup. The input features and target values were converted into PyTorch tensors and reshaped appropriately to fit the KAN library's expected input format. These tensors were then grouped into a dataset dictionary containing training and test inputs and labels. The default KAN model was configured with a simple architecture consisting of one hidden layer of 5 neurons, a spline grid size of 3, and spline degree of 3. The model was trained using the *L-BFGS* optimization method with 50 steps and without saving intermediate states. This served as a baseline comparison to assess whether the AutoKAN optimization process could yield improvements over a manually configured model.

5.4 Benchmarking

Once all models were implemented, their performance was assessed using the Mean Squared Error (MSE) as the evaluation metric. The goal was to compare how accurately each model predicted the target variable on the test data.

The evaluation began with training the AutoKAN model on the training set. Once the best hyperparameters were found through Bayesian optimization, the model was used to generate predictions on the test set. The default KAN model, previously trained, was also evaluated by performing forward propagation on the test input tensor, ensuring that gradient computation was disabled during inference.

For the random forest and linear regression models, predictions were generated directly using their *.predict()* methods on the test set.

All predictions were compared to the true target values using the *mean_squared_error* function, and the resulting MSE values were collected in a results dictionary. Additionally, the best hyperparameters selected by AutoKAN were stored for reference.

5.5 Evaluation of Results

Once all models were trained on the three datasets, their performance was evaluated using the Mean Squared Error (MSE). The results are visually summarized in bar charts, allowing for direct comparison between the benchmark models and the AutoKAN implementation.

For the synthetic dataset (Figure 5.6), the best-performing model was linear regression (MSE = 998), followed by the default KAN (MSE = 1199) and random forest (MSE = 1437). AutoKAN, despite the optimization process, performed considerably worse, with an MSE of 4777. This is likely due to the simplicity of the underlying data structure, which favors linear models and penalizes more flexible architectures that are prone to overfitting or instability under suboptimal hyperparameter configurations.

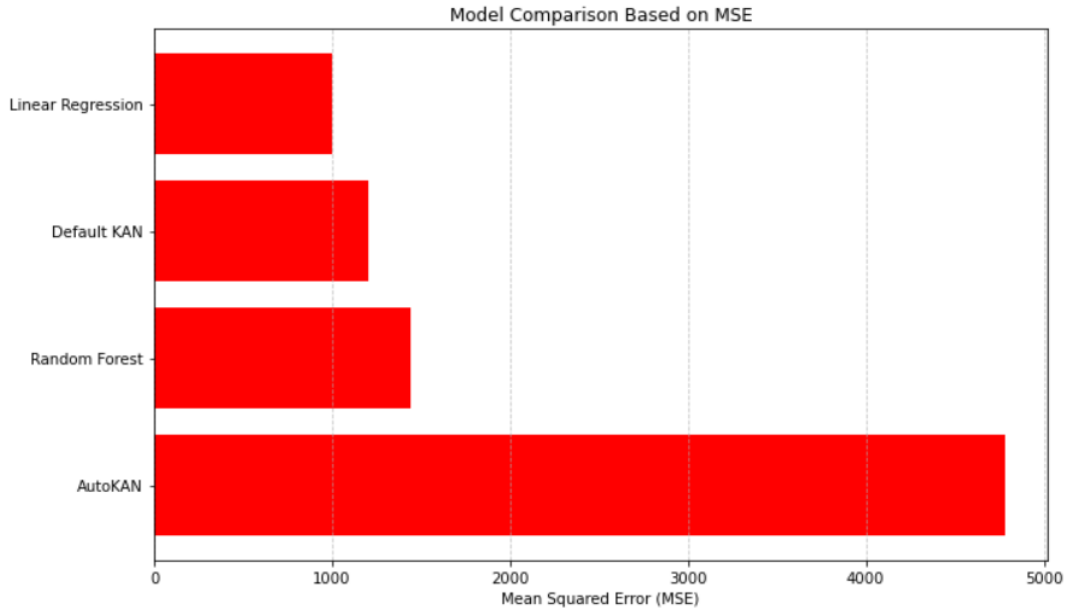


Figure 5.6: Benchmark for the synthetic dataset.

In the Boston Housing dataset (Figure 5.7), the results are even more striking. Random forest achieved the lowest MSE (7.9), followed by the default KAN (18.4) and linear regression (25.0). Again, AutoKAN fell significantly behind (535.4). This underperformance is surprising, especially given that the dataset exhibits meaningful relationships between features and the target variable. Despite this, the optimized model failed to capture these patterns effectively.

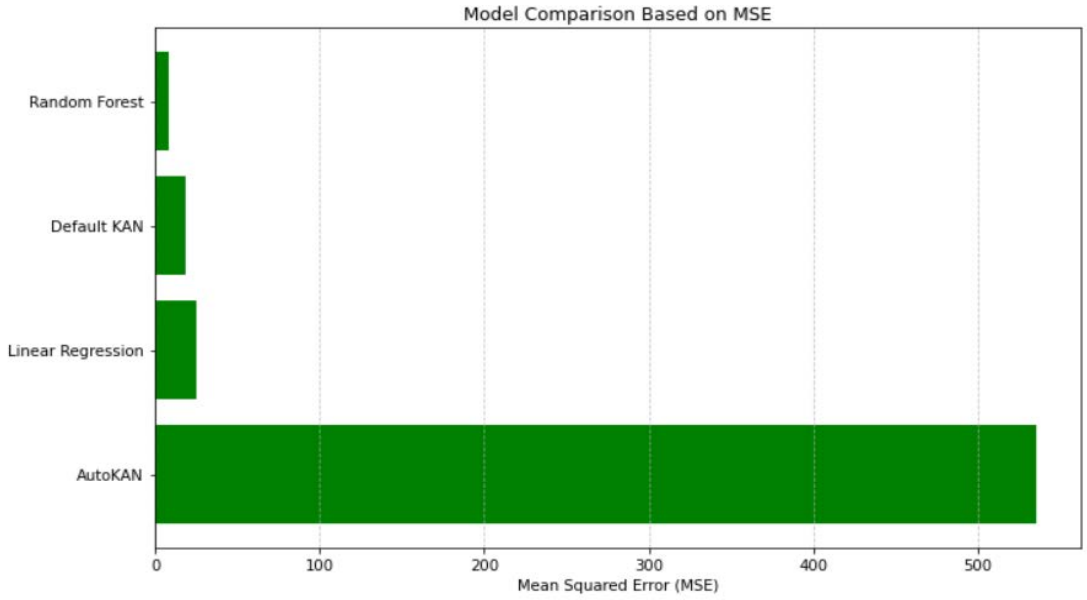


Figure 5.7: Benchmark for the Boston Housing dataset.

In the Diabetes dataset (Figure 5.7), a similar pattern emerged. Linear regression (MSE = 2900) and random forest (MSE = 2952) were again the best-performing models. The default KAN followed with a much higher error (MSE = 13086), while AutoKAN performed the worst (MSE = 26544). This suggests that AutoKAN was unable to generalize well, even though it selected a relatively simple architecture (1 layer, 2 neurons per layer). This poor performance could be attributed to the variability and noise inherent in clinical datasets like this one.

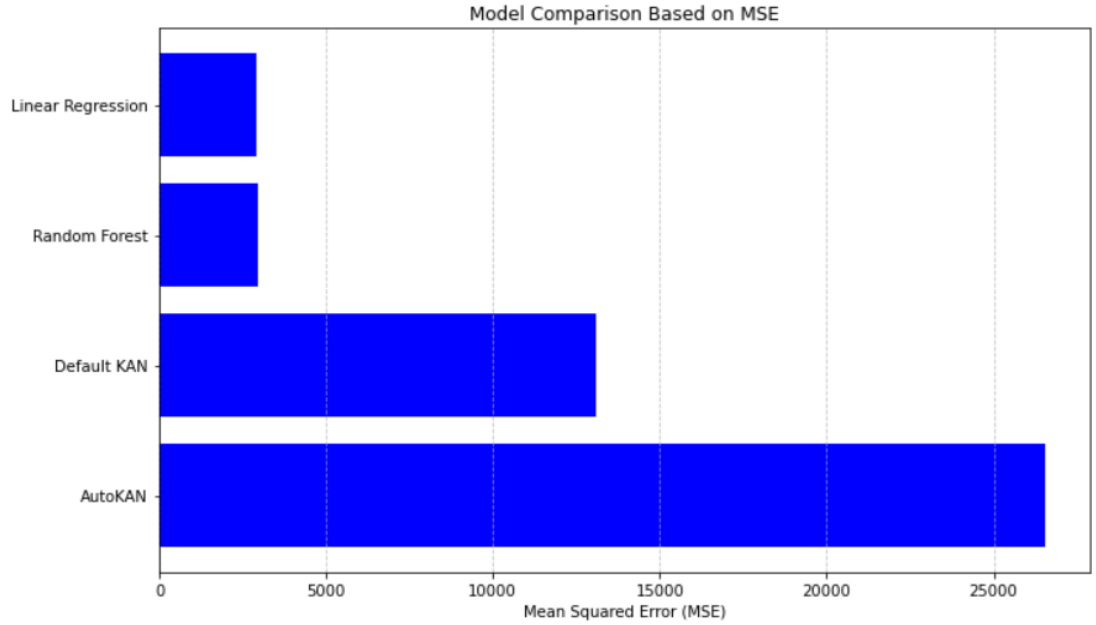


Figure 5.8: Benchmark for the Diabates dataset.

Taken together, these results highlight several important points. First, traditional models like linear regression and random forest proved to be more robust and reliable across all datasets. Second, the default KAN, even without tuning, provided competitive performance, particularly on the Boston dataset, suggesting that its architecture is reasonable for certain real-world problems.

On the other hand, AutoKAN consistently failed to outperform any of the benchmarks. While this might initially seem like a failure of the algorithm, it is more accurately attributed to two key technical limitations that arose during experimentation:

1. **Limited number of optimization iterations**: Due to computational constraints, Bayesian optimization was limited to just 10 iterations. Searching over a 6-dimensional hyperparameter space with only 10 evaluations is extremely inefficient. If most of the space leads to poor results, the optimizer has little chance of discovering

a high-performing region unless it gets extremely lucky early on.

2. **Inadequate prior for the Gaussian process**: The default kernel used by the Gaussian process in *gp_minimize* is better suited for other model types, such as neural networks. However, KAN's hyperparameter space is fundamentally different and far more complex. This mismatch makes it difficult for the optimizer to model the objective function effectively. To overcome this, one would need to study the KAN space in more detail and design a custom kernel or use a more informed prior, potentially leveraging the default KAN configuration as a starting point for sampling.

As a temporary path, I addressed some of the training failures by simplifying the architectures (like reducing the number of neurons per layer) and adding exception handling to prevent crashes when NaN values were encountered during training. While these fixes allowed experiments to continue, they did not significantly improve performance and highlighted the model's sensitivity to poor hyperparameter settings.

This underscores the difficulty of navigating such a complex search space with a limited computational budget, especially when the default Gaussian Process kernel used in the Bayesian optimization is not well-suited to the characteristics of KAN's hyperparameter space. As discussed, the AutoKAN's performance may also be constrained by the default Gaussian Process kernel used in the Bayesian optimization, which is not well-suited to the characteristics of KAN's hyperparameter space. Addressing these technical challenges would likely be essential to achieving the true potential of AutoKAN.

Chapter 6

Conclusions

This thesis explored the potential of AutoKAN, a custom Python implementation that automates the hyperparameter tuning process for Kolmogorov-Arnold Networks (KANs) using Bayesian optimization. The project was built on the premise that KANs, due to their flexible and interpretable architecture, can offer competitive performance compared to traditional machine learning models, and that an automated tuning approach could further improve their effectiveness.

To evaluate this approach, I used three datasets: a synthetic dataset, the Boston Housing dataset, and the Diabetes dataset. I compared the performance of AutoKAN with three benchmark models: linear regression, random forest, and a standard version of KAN. All models were tested using Mean Squared Error (MSE) on a separate test set.

Across all datasets, traditional models such as linear regression and random forest consistently achieved lower MSE scores. Interestingly, even the default KAN, without any tuning, often outperformed AutoKAN. These results underscore the stability and competitiveness of the default KAN configuration but also highlight the current limitations of the AutoKAN approach.

After testing, two key technical constraints were identified that help explain the underperformance of AutoKAN. First, the number of Bayesian optimization iterations was limited to just 10 due to computational constraints. In a 6-dimensional hyperparameter space, this small number of evaluations offers very limited coverage and makes it unlikely to find optimal configurations. Second, the kernel used by the Gaussian Process in the optimization process is not well suited for the complex and irregular hyperparameter space of KANs. As a result, the optimizer struggles to model the objective function accurately and efficiently.

To reduce the number of errors (NaN values) that occurred during training, I also simplified the KAN architecture and added exception handling. These changes helped the code run more smoothly, but did not improve AutoKAN’s performance enough.

These findings may reflect a structural mismatch between the optimization method and the model’s architecture rather than a failure of the overall AutoKAN approach. In fact, the observed sensitivity of KANs to their hyperparameters is well documented in prior research and could justify the difficulty in optimizing them with standard Bayesian tools.

Looking forward, while AutoKAN did not outperform benchmark models in this iteration, it laid the groundwork for future research. Increasing the number of optimization iterations, using a different kernel that fits KANs better, or even starting the search from the default KAN configuration could significantly improve results. Furthermore, this work reinforces the idea that hyperparameter optimization is not one-size-fits-all, as models as structurally unique as KANs may require equally unique tuning strategies.

AutoKAN demonstrates that with the right refinements, automated tuning for KANs could become a powerful tool, capable of harnessing their full potential in high-dimensional and complex machine learning tasks. This thesis contributes both a functional prototype and a critical evaluation of its current limitations, offering a clear path for future improvement and experimentation.

Declaration of the use of AI tools

Declaración de Uso de Herramientas de Inteligencia Artificial Generativa en Trabajos Fin de Grado

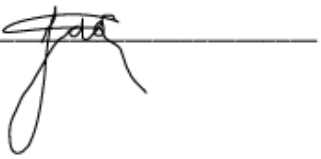
ADVERTENCIA: Desde la Universidad consideramos que ChatGPT u otras herramientas similares son herramientas muy útiles en la vida académica, aunque su uso queda siempre bajo la responsabilidad del alumno, puesto que las respuestas que proporciona pueden no ser veraces. En este sentido, NO está permitido su uso en la elaboración del Trabajo fin de Grado para generar código porque estas herramientas no son fiables en esa tarea. Aunque el código funcione, no hay garantías de que metodológicamente sea correcto, y es altamente probable que no lo sea.

Por la presente, yo, Adriana Fernández de Caleya Vázquez, estudiante de E2 Analytics de la Universidad Pontificia Comillas al presentar mi Trabajo Fin de Grado titulado "AutoKAN: The power of Kolmogorov-Arnold Networks and Bayesian optimization", declaro que he utilizado la herramienta de Inteligencia Artificial Generativa ChatGPT u otras similares de IAG de código sólo en el contexto de las actividades descritas a continuación:

1. **Brainstorming de ideas de investigación:** Utilizado para idear y esbozar posibles áreas de investigación.
2. **Referencias:** Usado conjuntamente con otras herramientas, como Science, para identificar referencias preliminares que luego he contrastado y validado.
3. **Estudios multidisciplinares:** Para comprender perspectivas de otras comunidades sobre temas de naturaleza multidisciplinar.
4. **Constructor de plantillas:** Para diseñar formatos específicos para secciones del trabajo.
5. **Corrector de estilo literario y de lenguaje:** Para mejorar la calidad lingüística y estilística del texto.
6. **Sintetizador y divulgador de libros complicados:** Para resumir y comprender literatura compleja.
7. **Revisor:** Para recibir sugerencias sobre cómo mejorar y perfeccionar el trabajo con diferentes niveles de exigencia.
8. **Traductor:** Para traducir textos de un lenguaje a otro.

Afirmo que toda la información y contenido presentados en este trabajo son producto de mi investigación y esfuerzo individual, excepto donde se ha indicado lo contrario y se han dado los créditos correspondientes (he incluido las referencias adecuadas en el TFG y he explicitado para que se ha usado ChatGPT u otras herramientas similares). Soy consciente de las implicaciones académicas y éticas de presentar un trabajo no original y acepto las consecuencias de cualquier violación a esta declaración.

Fecha: 09 de abril de 2025

Firma: 

References

Azam, B., & Akhtar, N. (2024). Suitability of KANs for computer vision: A preliminary investigation. *arXiv (Cornell University)*, https://10.48550/arxiv.2406.09087

Cacciatore, A., Morelli, V., Paganica, F., Frontoni, E., Migliorelli, L., & Berardini, D. (2024). A preliminary study on continual learning in computer vision using kolmogorov-arnold networks.https://10.48550/arxiv.2409.13550

Cambrin, D. R., Poeta, E., Pastor, E., Cerquitelli, T., Baralis, E., & Garza, P. (2024). KAN you see it? KANs and sentinel for effective and explainable crop field segmentation.https://10.48550/arxiv.2408.07040

Chen, A. S. (2024). Gaussian process kolmogorov-arnold networks. https://10.48550/arxiv.2407.18397

Daulton, S., Eriksson, D., Balandat, M., & Bakshy, E. (2021). Multi-objective bayesian optimization over high-dimensional search spaces. *arXiv (Cornell University)*, https://10.48550/arxiv.2109.10964

Efron, B., Hastie, T., Johnstone, I., & Tibshirani, R. (2004). Least angle regression. The Annals of Statistics, 32(2), 407–499.

Frazier, P. I. (2018). A tutorial on Bayesian optimization. arXiv preprint arXiv:1807.02811.

Harrison, D., & Rubinfeld, D. L. (1978). Hedonic housing prices and the demand for clean air. Journal of Environmental Economics and Management, 5(1), 81–102.

Hassan, M. M. (2024). Bayesian kolmogorov arnold networks (Bayesian_KANs): A probabilistic approach to enhance accuracy and interpretability. https://10.48550/arxiv.2408.02706

Head, T., Louppe, G., Shcherbatyi, I., Charras, F., & Varoquaux, G. (2018). Scikit-optimize: Sequential model-based optimization with a scikit-learn interface. GitHub. https://github.com/scikit-optimize/scikit-optimize

Li, H., Li, J., Guan, X., Liang, B., Lai, Y., & Luo, X. (Dec 2019). (Dec 2019). Research on overfitting of deep learning. Paper presented at the 78–81. https://10.1109/CIS.2019.00025 https://ieeexplore.ieee.org/document/9023664

Liu, Z., Ma, P., Wang, Y., Matusik, W., & Tegmark, M. (2024b). KAN 2.0: Kolmogorov-arnold networks meet science. https://10.48550/arxiv.2408.10205

Liu, Z., Wang, Y., Vaidya, S., Ruehle, F., Halverson, J., Soljačić, M., Hou, T. Y., & Tegmark, M. (2024a). KAN: Kolmogorov-arnold networks. *arXiv (Cornell University)*, https://10.48550/arxiv.2404.19756

Lu, Y., & Zhan, F. (2024). Kolmogorov arnold networks in fraud detection: Bridging the gap between theory and practice.https://10.48550/arxiv.2408.10263

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., & Duchesnay, É. (2011). Scikit-learn: Machine learning in Python. Journal of Machine Learning Research, 12, 2825–2830.

Pourkamali-Anaraki, F. (2024). Kolmogorov-arnold networks in low-data regimes: A comparative study with multilayer perceptrons.https://10.48550/arxiv.2409.10463

ROCHA, A., PAPA, J. P., & MEIRA, L. A. A. (2012). How far do we get using machine learning black-boxes? *International Journal of Pattern Recognition and Artificial Intelligence, 26*(2), 1261001–1261023. https://10.1142/S0218001412610010

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, *61*, 85–117. https://10.1016/j.neunet.2014.09.003

Shen, H., Zeng, C., Wang, J., & Wang, Q. (2024). Reduced effectiveness of kolmogorov-arnold networks on functions with noise.https://10.48550/arxiv.2407.14882

Snoek, J., Larochelle, H., & Adams, R. P. (2012). Practical Bayesian optimization of machine learning algorithms. Advances in Neural Information Processing Systems, 25.

Sulaiman, M. H., Mustaffa, Z., Saealal, M. S., Saari, M. M., & Ahmad, A. Z. (2024). Utilizing the kolmogorov-arnold networks for chiller energy consumption prediction in commercial building. *Journal of Building Engineering*, *96*, 110475. https://10.1016/j.jobe.2024.110475

Ter-Avanesov, B., & Beigi, H. (2024). *MLP, XGBoost, KAN, TDNN, and LSTM-GRU hybrid RNN with attention for SPX and NDX european call option pricing.* (). Ithaca: Cornell University Library, arXiv.org. https://10.48550/arxiv.2409.06724 Retrieved from Publicly Available Content Database https://www.proquest.com/docview/3103644209/abstract/

Toosi, A., Bottino, A. G., Saboury, B., Siegel, E., & Rahmim, A. (2021). A brief history of AI: How to prevent another winter (A critical review). *PET Clinics, 16*(4), 449–469. https://10.1016/j.cpet.2021.07.001

Wason, R. (2018). Deep learning: Evolution and expansion. *Cognitive Systems Research, 52*, 701–708. https://10.1016/j.cogsys.2018.08.023

Zohuri, B., & Moghaddam, M. (2020). Deep learning limitations and flaws. *Mod.Approaches Mater.Sci, 2*, 241–250.

Annex

AutoKAN - The power of KAN and Bayesian Optimization

In [32]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import torch
from kan import KAN
from tqdm import tqdm
from sklearn.datasets import make_regression
from sklearn.model_selection import cross_val_score, KFold, train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from skopt import gp_minimize
from skopt.space import Integer
```

1. AutoKAN Implementation

In [33]:
```
class AutoKAN:
    def __init__(self, X, y, param_space=None, n_iter=10, n_splits=2, seed=42):
        self.X = X
        self.y = y
        self.n_iter = n_iter
        self.n_splits = n_splits
        self.seed = seed
        self.iterations = 0

        if torch.cuda.is_available():
            self.device = torch.device("cuda")
        else:
            self.device = torch.device("cpu")

        #Hay que comparar un random forest, un KAN por defecto, un MLP y la regla de la mayoria cor
        self.train_data, self.test_data, self.train_target, self.test_target = train_test_split(se

        self.train_data = torch.tensor(self.train_data)
        self.test_data = torch.tensor(self.test_data)
        self.train_target = torch.tensor(self.train_target)
        self.test_target = torch.tensor(self.test_target)

        # Create data loaders (optional, if you want to batch and shuffle the data)
        train_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(self.train_data,
        test_loader = torch.utils.data.DataLoader(torch.utils.data.TensorDataset(self.test_data, se

        self.train_inputs = torch.empty(0, self.X.shape[1], device=self.device)
        self.train_labels = torch.empty(0, dtype=torch.long, device=self.device)
        self.test_inputs = torch.empty(0, self.X.shape[1], device=self.device)
        self.test_labels = torch.empty(0, dtype=torch.long, device=self.device)

        # Concatenate all data into a single tensor on the specified device
        for data, labels in tqdm(train_loader):
            self.train_inputs = torch.cat((self.train_inputs, data.to(self.device)), dim=0)
            self.train_labels = torch.cat((self.train_labels, labels.to(self.device)), dim=0)

        for data, labels in tqdm(test_loader):
            self.test_inputs = torch.cat((self.test_inputs, data.to(self.device)), dim=0)
            self.test_labels = torch.cat((self.test_labels, labels.to(self.device)), dim=0)

        self.dataset = {}
        self.dataset['train_input'] = self.train_inputs.float()
        self.dataset['test_input'] = self.test_inputs.float()
        self.dataset['train_label'] = self.train_labels.reshape(-1, 1).float()
        self.dataset['test_label'] = self.test_labels.reshape(-1, 1).float()

        # Default hyperparameter space, meter aqui como real (ver skopt) lamb y lamb_entropy (ver r
        self.param_space = param_space or [
        Integer(1, 2), #name="num_layers"
        Integer(2, 6), #name="neurons_per_layer"
        Integer(3, 10), #name="splines_per_neuron"
        Integer(1, 5) #name="spline_degree"
        ]

        self.best_params = None
        self.best_score = None
        self.best_model = None
        self.width_vector = [self.X.shape[1]]


    def _train_mse(self):
        with torch.no_grad():
            predictions = self.model(self.dataset['train_input'])
            mse = torch.nn.functional.mse_loss(predictions, self.dataset['train_label'])
        return mse

    def _test_mse(self):
        with torch.no_grad():
            predictions = self.model(self.dataset['test_input'])
            mse = torch.nn.functional.mse_loss(predictions, self.dataset['test_label'])
        return mse

    def _evaluate_model(self, params):

        # Convert parameters to a dictionary
        param_dict = {
```

```
            "num_layers": params[0],
            "neurons_per_layer": params[1],
            "splines_per_neuron": params[2],
            "spline_degree": params[3]
        }

        hidden_layers = np.full(param_dict["num_layers"], param_dict["neurons_per_layer"]).tolist()
        self.width_vector = self.width_vector + hidden_layers + [1]
        self.model = KAN(width = self.width_vector, grid=param_dict["splines_per_neuron"], k=param_
        self.iterations += 1

        try:
          results = self.model.fit(self.dataset, opt="LBFGS", metrics=(self._train_mse, self._test_
                    loss_fn=torch.nn.MSELoss(), steps=50, lamb=0.01, lamb_entropy=2.)

          error = -results['_test_mse'][-1]

          if torch.isnan(torch.tensor(error)).item():
            error = 1.0 # Esto habria que hacerlo mejor...

        except:
          error = 1.0

        import shutil, os
        if os.path.isdir('model'):
          shutil.rmtree('model')
        self.model = None

        return error  # Minimize the error


    def train(self):
        # Perform Bayesian optimization
        result = gp_minimize(
            self._evaluate_model,
            self.param_space,
            n_calls=self.n_iter,
            random_state=self.seed
        )

        # Save the best results, meter tambien aqui los otros dos hiperparametros.
        self.best_params = {
            "num_layers": result.x[0],
            "neurons_per_layer": result.x[1],
            "splines_per_neuron": result.x[2],
            "spline_degree": result.x[3]
        }
        self.best_score = -result.fun

        # Train the best KAN model
        hidden_layers = np.full(self.best_params["num_layers"], self.best_params["neurons_per_layer
        self.width_vector = self.width_vector + hidden_layers + [1]

        self.best_model = KAN(width = self.width_vector, grid=self.best_params["splines_per_neuron"

        return self.best_score, self.best_params

    def predict(self, X_new):
        if not self.best_model:
            raise ValueError("The model has not been trained yet. Call the 'train' method first.")

        X_tensor = torch.tensor(X_new, dtype=torch.float32, device=self.device)
        self.best_model.eval()

        with torch.no_grad():
            predictions = self.best_model(X_tensor).cpu().numpy()

        return predictions
```

2. Benchmark Models Implementation

DATASET MAKE_REGRESSION + NOISE

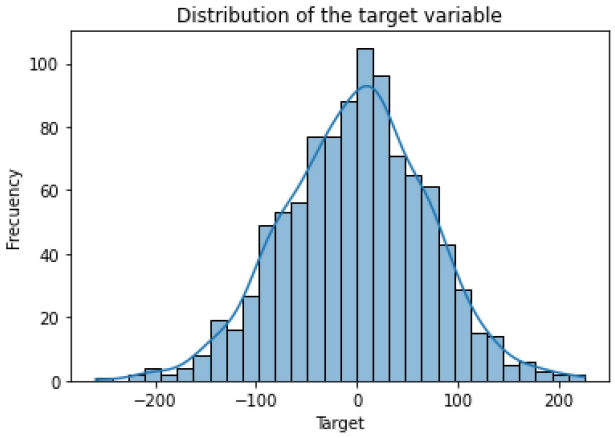
In [34]:
```
X, y = make_regression(n_samples=1000, n_features=5, noise=30, random_state=42)
```

In [35]:
```
df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(X.shape[1])])
df['Target'] = y
df.describe()
```

Out[35]:

	Feature_0	Feature_1	Feature_2	Feature_3	Feature_4	Target
count	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000	1000.000000
mean	0.006531	0.004543	0.001297	0.032846	-0.017207	-1.001639
std	0.959842	1.005321	0.999219	0.988835	1.029248	70.862407
min	-3.007632	-3.241267	-3.170426	-3.176704	-2.940389	-258.924050
25%	-0.649871	-0.693862	-0.645736	-0.607606	-0.711821	-46.868721
50%	0.064768	0.022395	0.027885	-0.006855	-0.034162	2.659962
75%	0.679256	0.642029	0.649413	0.690789	0.663808	46.155573
max	3.078881	3.152057	3.926238	3.852731	3.243093	225.591851

In [36]:
```
# Histograma
plt.figure(figsize=(6,4))
sns.histplot(df['Target'], bins=30, kde=True)
plt.title("Distribution of the target variable")
plt.xlabel("Target")
plt.ylabel("Frecuency")
plt.show()
```



In [37]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Linear Regression
linreg = LinearRegression()
linreg.fit(X_train, y_train)

# Random Forest Regressor
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)
rf_reg.fit(X_train, y_train)

# KAN por defecto
train_data = torch.tensor(X_train, dtype=torch.float32)
train_labels = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
test_data = torch.tensor(X_test, dtype=torch.float32)
test_labels = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)

dataset = {
    'train_input': train_data,
    'train_label': train_labels,
    'test_input': test_data,
    'test_label': test_labels
}

# Definir la KAN (estructura básica sugerida para regresión)
kan_default = KAN(width=[X.shape[1], 5, 1], grid=3, k=3, device='cpu', auto_save=False)

# Entrenar la KAN por defecto
kan_default.fit(dataset, opt='LBFGS', steps=50);
```

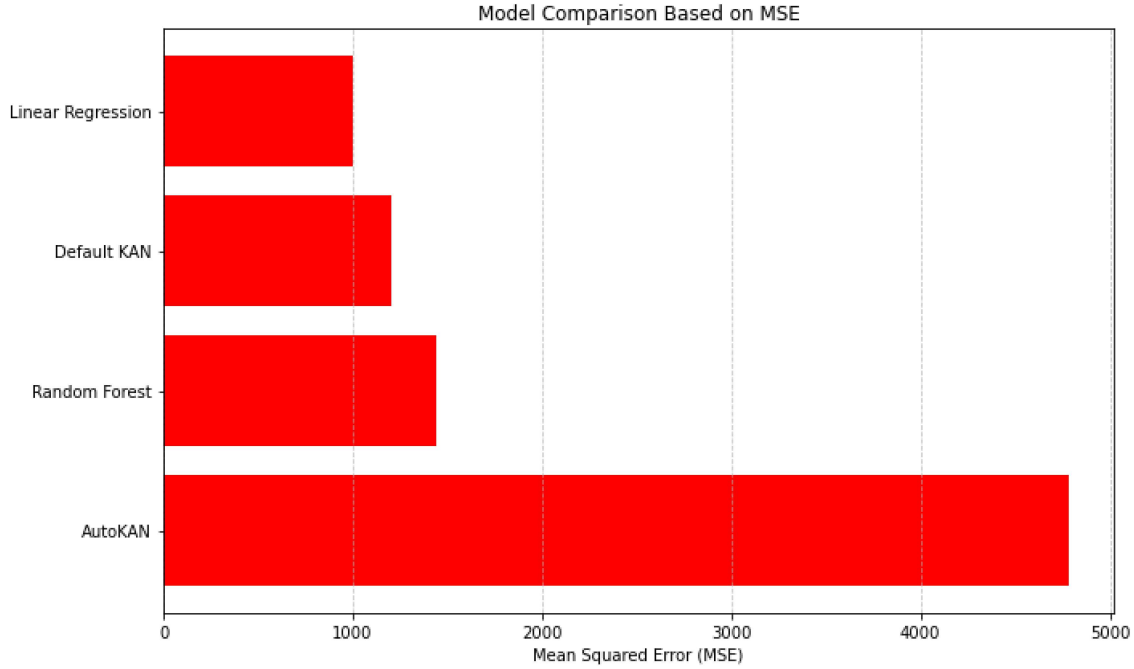
```
| train_loss: 2.87e+01 | test_loss: 3.46e+01 | reg: 2.08e+02 | : 100%|| 50/50 [00:18<00:00,  2.7
6it
```

In [38]:
```
auto_kan = AutoKAN(X_train, y_train)
auto_score, auto_params = auto_kan.train()

# Evaluar todos los modelos
results = {}

# AutoKAN
y_pred_auto = auto_kan.predict(X_test)
results["AutoKAN"] = mean_squared_error(y_test, y_pred_auto)

# KAN por defecto
with torch.no_grad():
    y_pred_defaultkan = kan_default(test_data).cpu().numpy()
results["Default KAN"] = mean_squared_error(y_test, y_pred_defaultkan)

# Random Forest
y_pred_rf = rf_reg.predict(X_test)
results["Random Forest"] = mean_squared_error(y_test, y_pred_rf)

# Linear Regression
y_pred_lr = linreg.predict(X_test)
results["Linear Regression"] = mean_squared_error(y_test, y_pred_lr)

# Mostrar resultados
print("\nResultados de MSE para cada modelo:\n")
for model, mse in results.items():
    print(f"{model}: MSE = {mse:.4f}")

# Mostrar mejores hiperparámetros encontrados para AutoKAN
print("\nMejores hiperparámetros AutoKAN:", auto_params)
```

```
100%|██████████| 640/640 [00:00<00:00, 11325.72it/s]
100%|██████████| 160/160 [00:00<00:00, 11429.79it/s]
| train_loss: 2.57e+01 | test_loss: 3.57e+02 | reg: 1.02e+02 | : 100%|█| 50/50 [00:34<00:00,  1.4
6it
| train_loss: 3.11e+01 | test_loss: 6.07e+01 | reg: 1.17e+02 | : 100%|█| 50/50 [00:45<00:00,  1.1
1it
| train_loss: 4.34e+01 | test_loss: 5.89e+01 | reg: 2.12e+02 | : 100%|█| 50/50 [01:23<00:00,  1.6
7s/
| train_loss: 6.95e+01 | test_loss: 7.77e+01 | reg: 1.30e+01 | : 100%|█| 50/50 [00:34<00:00,  1.4
6it
| train_loss: 6.95e+01 | test_loss: 7.77e+01 | reg: 1.68e+01 | : 100%|█| 50/50 [00:32<00:00,  1.5
2it
| train_loss: 6.95e+01 | test_loss: 7.77e+01 | reg: 1.37e+01 | : 100%|█| 50/50 [00:35<00:00,  1.4
2it
| train_loss: nan | test_loss: nan | reg: nan | :  10%|▉          | 5/50 [00:25<03:47,  5.05
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|▉          | 5/50 [00:21<03:14,  4.33
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  20%|█▊         | 10/50 [00:31<02:04,  3.11s/
it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|▉          | 5/50 [00:29<04:27,  5.94
s/it]
```

```
lstsq failed

Resultados de MSE para cada modelo:

AutoKAN: MSE = 4777.7807
Default KAN: MSE = 1199.5101
Random Forest: MSE = 1437.6175
Linear Regression: MSE = 998.1549

Mejores hiperparámetros AutoKAN: {'num_layers': 2, 'neurons_per_layer': 3, 'splines_per_neuron':
8, 'spline_degree': 3}
```

In [40]:
```
sorted_results = dict(sorted(results.items(), key=lambda item: item[1]))

model_names = list(sorted_results.keys())
mse_values = list(sorted_results.values())

plt.figure(figsize=(10, 6))
plt.barh(model_names, mse_values, color='red')
plt.xlabel('Mean Squared Error (MSE)')
plt.title('Model Comparison Based on MSE')
plt.gca().invert_yaxis()  # Mejor modelo arriba
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



DATASET BOSTON HOUSING

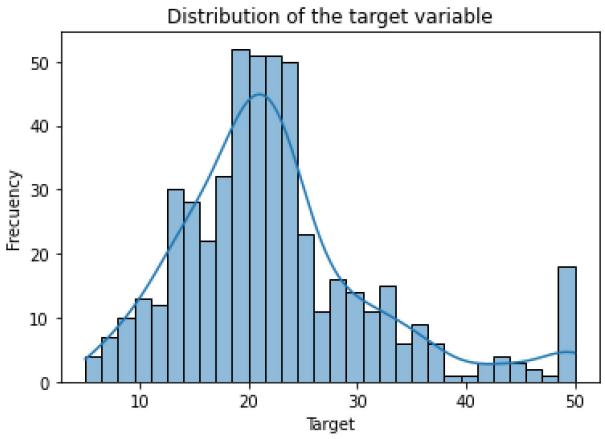
In [41]:
```
#PROBLEMA ÉTICO: tiene una variable de porcentaje de población afroamericana que influye en un menc
from sklearn.datasets import fetch_openml
boston = fetch_openml(name="boston", version=1, as_frame=False)
X = boston.data
y = boston.target
```

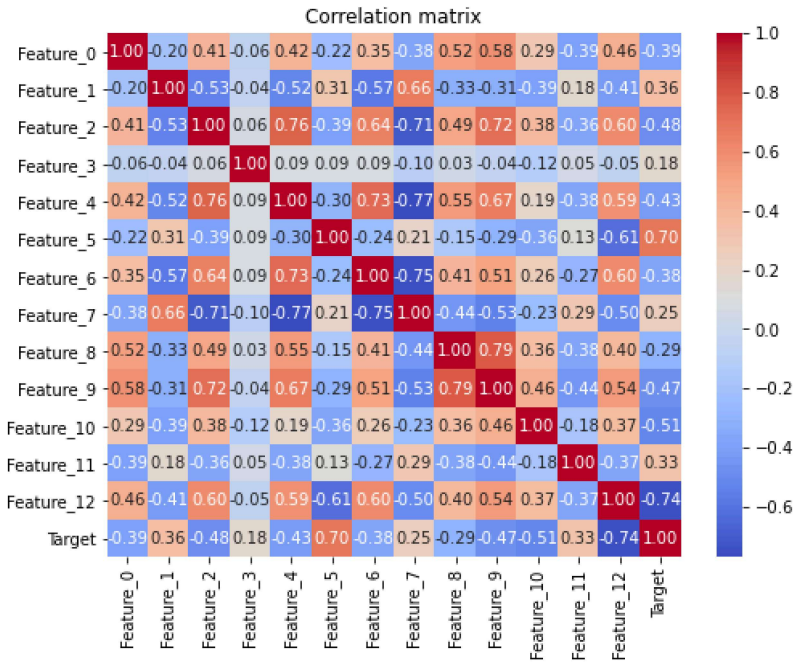
```
C:\Users\Adriana\anaconda3\lib\site-packages\sklearn\datasets\_openml.py:1022: FutureWarning: The
default value of `parser` will change from `'liac-arff'` to `'auto'` in 1.4. You can set `parser
='auto'` to silence this warning. Therefore, an `ImportError` will be raised from 1.4 if the data
set is dense and pandas is not installed. Note that the pandas parser may return different data t
ypes. See the Notes Section in fetch_openml's API doc for details.
  warn(
```

In [42]:
```
df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(X.shape[1])])
df['Target'] = y
print("Summary Statistics\n")
print(df['Target'].describe())
```

```
Summary Statistics

count    506.000000
mean      22.532806
std        9.197104
min        5.000000
25%       17.025000
50%       21.200000
75%       25.000000
max       50.000000
Name: Target, dtype: float64
```

In [43]:
```
# Histograma
plt.figure(figsize=(6,4))
sns.histplot(df['Target'], bins=30, kde=True)
plt.title("Distribution of the target variable")
plt.xlabel("Target")
plt.ylabel("Frecuency")
plt.show()

# Mapa de correlaciones
plt.figure(figsize=(8,6))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation matrix")
plt.show()
```





In [44]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Linear Regression
linreg = LinearRegression()
linreg.fit(X_train, y_train)

# Random Forest Regressor
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)
rf_reg.fit(X_train, y_train)

# KAN por defecto
train_data = torch.tensor(X_train, dtype=torch.float32)
train_labels = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
test_data = torch.tensor(X_test, dtype=torch.float32)
test_labels = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)

dataset = {
    'train_input': train_data,
    'train_label': train_labels,
    'test_input': test_data,
    'test_label': test_labels
}

# Definir la KAN (estructura básica sugerida para regresión)
kan_default = KAN(width=[X.shape[1], 5, 1], grid=3, k=3, device='cpu', auto_save=False)

# Entrenar la KAN por defecto
kan_default.fit(dataset, opt='LBFGS', steps=50);
```

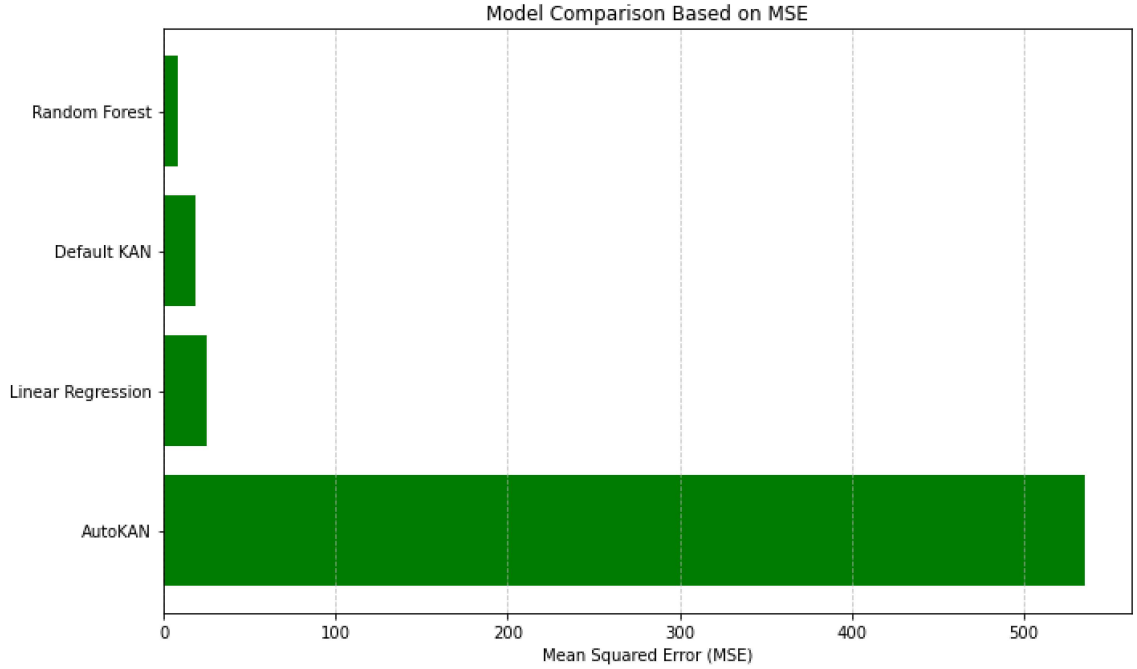
```
| train_loss: 3.94e+00 | test_loss: 4.29e+00 | reg: 4.76e+01 | : 100%|| 50/50 [00:14<00:00,  3.3
5it
```

In [45]:
```
auto_kan = AutoKAN(X_train, y_train)
auto_score, auto_params = auto_kan.train()

# Evaluar todos los modelos
results = {}

# AutoKAN
y_pred_auto = auto_kan.predict(X_test)
results["AutoKAN"] = mean_squared_error(y_test, y_pred_auto)

# KAN por defecto
with torch.no_grad():
    y_pred_defaultkan = kan_default(test_data).cpu().numpy()
results["Default KAN"] = mean_squared_error(y_test, y_pred_defaultkan)

# Random Forest
y_pred_rf = rf_reg.predict(X_test)
results["Random Forest"] = mean_squared_error(y_test, y_pred_rf)

# Linear Regression
y_pred_lr = linreg.predict(X_test)
results["Linear Regression"] = mean_squared_error(y_test, y_pred_lr)

# Mostrar resultados
print("\nResultados de MSE para cada modelo:\n")
for model, mse in results.items():
    print(f"{model}: MSE = {mse:.4f}")

# Mostrar mejores hiperparámetros encontrados para AutoKAN
print("\nMejores hiperparámetros AutoKAN:", auto_params)
```

```
100%|██████████| 323/323 [00:00<00:00, 9949.84it/s]
100%|██████████| 81/81 [00:00<00:00, 10126.34it/s]
| train_loss: 2.82e+00 | test_loss: 3.73e+00 | reg: 6.28e+01 | : 100%|█| 50/50 [00:27<00:00,  1.8
5it
| train_loss: nan | test_loss: nan | reg: nan | :  20%|██          | 10/50 [00:08<00:32,  1.25i
t/s]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: 7.83e+00 | test_loss: 7.98e+00 | reg: 6.19e+01 | : 100%|█| 50/50 [01:04<00:00,  1.3
0s/
| train_loss: 2.46e+01 | test_loss: 2.43e+01 | reg: 2.91e+01 | : 100%|█| 50/50 [00:42<00:00,  1.1
7it
| train_loss: 2.47e+01 | test_loss: 2.43e+01 | reg: 3.33e+01 | : 100%|█| 50/50 [00:34<00:00,  1.4
6it
| train_loss: nan | test_loss: nan | reg: nan | :  10%|█            | 5/50 [00:09<01:28,  1.96
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: 2.47e+01 | test_loss: 2.43e+01 | reg: 2.48e+01 | : 100%|█| 50/50 [01:36<00:00,  1.9
2s/
| train_loss: nan | test_loss: nan | reg: nan | :  10%|█            | 5/50 [00:14<02:14,  2.98
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|█            | 5/50 [00:16<02:24,  3.21
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|█            | 5/50 [00:20<03:04,  4.10
s/it]
```

```
lstsq failed

Resultados de MSE para cada modelo:

AutoKAN: MSE = 535.4177
Default KAN: MSE = 18.3914
Random Forest: MSE = 7.9017
Linear Regression: MSE = 25.0491

Mejores hiperparámetros AutoKAN: {'num_layers': 2, 'neurons_per_layer': 2, 'splines_per_neuron':
10, 'spline_degree': 3}
```

In [46]:
```python
sorted_results = dict(sorted(results.items(), key=lambda item: item[1]))

model_names = list(sorted_results.keys())
mse_values = list(sorted_results.values())

plt.figure(figsize=(10, 6))
plt.barh(model_names, mse_values, color='green')
plt.xlabel('Mean Squared Error (MSE)')
plt.title('Model Comparison Based on MSE')
plt.gca().invert_yaxis()
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



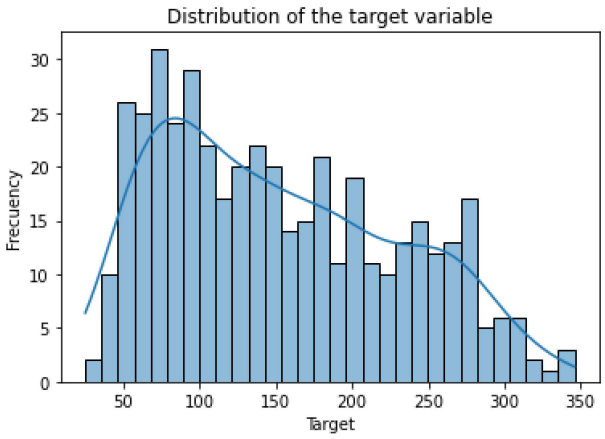
DATASET DIABETES

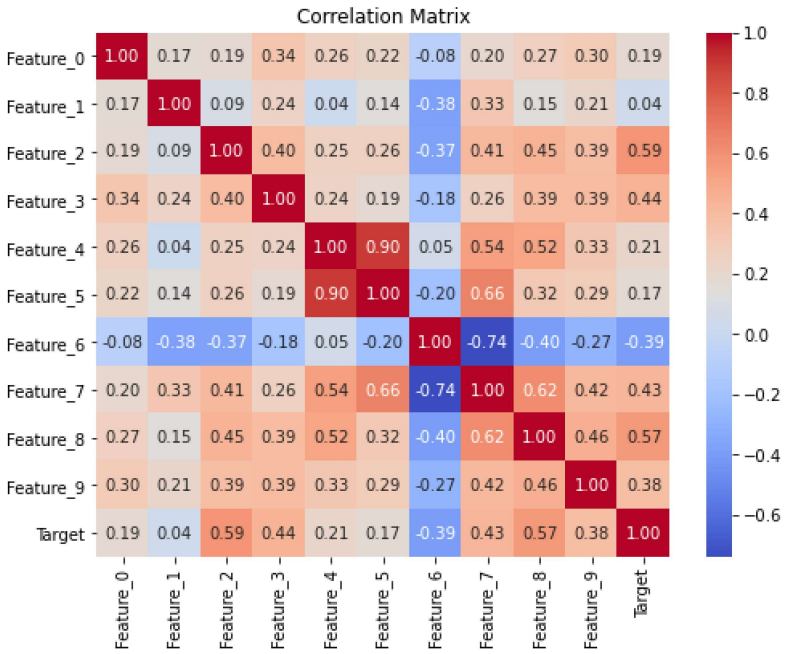
In [47]:
```python
from sklearn.datasets import load_diabetes
data = load_diabetes()
X = data.data
y = data.target
```

In [48]:
```
df = pd.DataFrame(X, columns=[f'Feature_{i}' for i in range(X.shape[1])])
df['Target'] = y
print("Summary Statistics\n")
print(df['Target'].describe())
```

```
Summary Statistics

count    442.000000
mean     152.133484
std       77.093005
min       25.000000
25%       87.000000
50%      140.500000
75%      211.500000
max      346.000000
Name: Target, dtype: float64
```

In [49]:
```
# Histograma
plt.figure(figsize=(6,4))
sns.histplot(df['Target'], bins=30, kde=True)
plt.title("Distribution of the target variable")
plt.xlabel("Target")
plt.ylabel("Frecuency")
plt.show()

# Mapa de correlaciones
plt.figure(figsize=(8,6))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", fmt=".2f")
plt.title("Correlation Matrix")
plt.show()
```





In [50]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Linear Regression
linreg = LinearRegression()
linreg.fit(X_train, y_train)

# Random Forest Regressor
rf_reg = RandomForestRegressor(n_estimators=100, random_state=42)
rf_reg.fit(X_train, y_train)

# KAN por defecto
train_data = torch.tensor(X_train, dtype=torch.float32)
train_labels = torch.tensor(y_train, dtype=torch.float32).reshape(-1, 1)
test_data = torch.tensor(X_test, dtype=torch.float32)
test_labels = torch.tensor(y_test, dtype=torch.float32).reshape(-1, 1)

dataset = {
    'train_input': train_data,
    'train_label': train_labels,
    'test_input': test_data,
    'test_label': test_labels
}

# Definir la KAN (estructura básica sugerida para regresión)
kan_default = KAN(width=[X.shape[1], 5, 1], grid=3, k=3, device='cpu', auto_save=False)

# Entrenar la KAN por defecto
kan_default.fit(dataset, opt='LBFGS', steps=50);
```

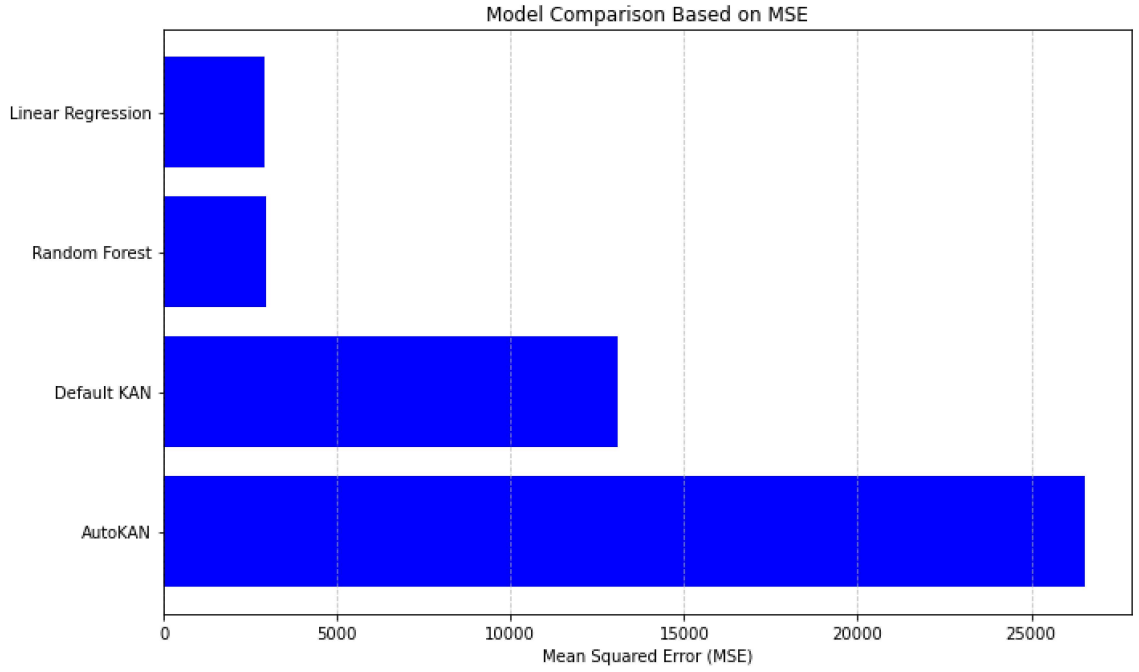
```
| train_loss: 3.30e+01 | test_loss: 1.14e+02 | reg: 4.89e+02 | : 100%|| 50/50 [00:11<00:00,  4.2
2it
```

In [51]:
```
auto_kan = AutoKAN(X_train, y_train)
auto_score, auto_params = auto_kan.train()

# Evaluar todos los modelos
results = {}

# AutoKAN
y_pred_auto = auto_kan.predict(X_test)
results["AutoKAN"] = mean_squared_error(y_test, y_pred_auto)

# KAN por defecto
with torch.no_grad():
    y_pred_defaultkan = kan_default(test_data).cpu().numpy()
results["Default KAN"] = mean_squared_error(y_test, y_pred_defaultkan)

# Random Forest
y_pred_rf = rf_reg.predict(X_test)
results["Random Forest"] = mean_squared_error(y_test, y_pred_rf)

# Linear Regression
y_pred_lr = linreg.predict(X_test)
results["Linear Regression"] = mean_squared_error(y_test, y_pred_lr)

# Mostrar resultados
print("\nResultados de MSE para cada modelo:\n")
for model, mse in results.items():
    print(f"{model}: MSE = {mse:.4f}")

# Mostrar mejores hiperparámetros encontrados para AutoKAN
print("\nMejores hiperparámetros AutoKAN:", auto_params)
```

```
100%|██████████| 282/282 [00:00<00:00, 7940.08it/s]
100%|██████████| 71/71 [00:00<00:00, 6454.31it/s]
| train_loss: 4.50e+01 | test_loss: 7.38e+01 | reg: 4.44e+02 | : 100%|█| 50/50 [00:22<00:00,  2.2
7it
| train_loss: 1.68e+02 | test_loss: 1.83e+02 | reg: 6.22e+00 | : 100%|█| 50/50 [00:31<00:00,  1.5
9it
| train_loss: 1.69e+02 | test_loss: 1.84e+02 | reg: 6.30e+00 | : 100%|█| 50/50 [00:41<00:00,  1.2
1it
| train_loss: 1.69e+02 | test_loss: 1.85e+02 | reg: 5.19e+00 | : 100%|█| 50/50 [00:35<00:00,  1.4
2it
| train_loss: 1.69e+02 | test_loss: 1.85e+02 | reg: 6.04e+00 | : 100%|█| 50/50 [00:24<00:00,  2.0
2it
| train_loss: 1.69e+02 | test_loss: 1.85e+02 | reg: 7.21e+00 | : 100%|█| 50/50 [00:38<00:00,  1.3
0it
| train_loss: nan | test_loss: nan | reg: nan | :  10%|█             | 5/50 [00:16<02:25,  3.24
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|█             | 5/50 [00:14<02:06,  2.81
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|█             | 5/50 [00:15<02:20,  3.12
s/it]
description:   0%|                                                    | 0/50 [00:00<?, ?i
t/s]

lstsq failed

| train_loss: nan | test_loss: nan | reg: nan | :  10%|█             | 5/50 [00:19<02:56,  3.92
s/it]
```

```
lstsq failed

Resultados de MSE para cada modelo:

AutoKAN: MSE = 26544.7647
Default KAN: MSE = 13086.5385
Random Forest: MSE = 2952.0106
Linear Regression: MSE = 2900.1936

Mejores hiperparámetros AutoKAN: {'num_layers': 1, 'neurons_per_layer': 2, 'splines_per_neuron':
10, 'spline_degree': 2}
```

In [53]:
```
sorted_results = dict(sorted(results.items(), key=lambda item: item[1]))

model_names = list(sorted_results.keys())
mse_values = list(sorted_results.values())

plt.figure(figsize=(10, 6))
plt.barh(model_names, mse_values, color='blue')
plt.xlabel('Mean Squared Error (MSE)')
plt.title('Model Comparison Based on MSE')
plt.gca().invert_yaxis()
plt.grid(axis='x', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



In [ ]: