Ridge Regression

Hemraj Verma

Submitted for the Degree of Master of Science in

Data Science and Analytics



Department of Computer Science

Royal Holloway University of London

Egham, Surrey TW20 0EX, UK

September 04, 2023

**Acknowledgment**

I am immensely grateful for the opportunity to embark on this journey, a journey that has been enriched by the guidance and support of remarkable individuals. My sincere appreciation goes to my professor, Nicolo Colombo, whose insights and mentorship have deepened my understanding and ignited my curiosity in ways that I never thought possible. Your expertise has been a guiding light on this path of learning and growth.

To my father, Gopilal Verma, and my mother, Sunita Verma, I owe a debt of gratitude that words can hardly capture. Your unwavering belief in me, coupled with your boundless love and encouragement, have been the foundation of my strength and determination. Your sacrifices and support have been my driving force, propelling me forward even in the face of challenges.

**Declaration**

This report has been prepared on the basis of my own work. Where other published and unpublished source materials have been used, these have been acknowledged.

**Word Count**: 12502

**Student Name**: Hemraj Verma

**Date of Submission**: 04th Sep, 2023

**Signature**:

**Abstract**

This study presents a comprehensive investigation into the behavior, performance, and optimization of Ridge Regression models. Through a series of meticulous experiments, we examine the impact of various hyperparameters on model performance across multiple datasets, including synthetic salary data and the Boston Housing dataset. The study extends to compare Ridge Regression with other prevalent regression models, such as Kernel Ridge, Lasso, and Random Forest, among others. A novel aspect of this research includes the exploration of both batch and online learning modes for Ridge Regression. Our findings indicate that Ridge Regression shows significant promise in predictive modeling but is highly sensitive to hyperparameter tuning. While Ridge Regression consistently performs competitively, the study also uncovers avenues for improvement, particularly in the realms of feature engineering, computational efficiency, and model generalizability. This work serves as a foundational study for both academics and practitioners aiming to employ Ridge Regression in complex predictive tasks.

Contents

[1 Introduction 1](#_Toc144727102)

[1.1 Aim and Objectives 1](#_Toc144727103)

[1.2 Significance of the Project 2](#_Toc144727104)

[2 Ridge Regression 4](#_Toc144727105)

[2.1 Overview of Machine Learning 4](#_Toc144727106)

[2.2 Linear Regression 5](#_Toc144727107)

[2.3 Issues with Multicollinearity 5](#_Toc144727108)

[2.3.1 Kernel Ridge Regression 6](#_Toc144727109)

[2.4 Traditional Machine Learning Models 8](#_Toc144727110)

[2.4.1 Random Forest Regression 8](#_Toc144727111)

[2.4.2 Support Vector Regression 9](#_Toc144727112)

[2.4.3 K Nearest Neighbour Regression 9](#_Toc144727113)

[2.4.4 Lasso 9](#_Toc144727114)

[3 Design and Methodology 10](#_Toc144727115)

[3.1 Selection of Tools 10](#_Toc144727116)

[3.2 Data Pre-Processing Techniques 10](#_Toc144727117)

[3.2.1 Min-Max Scaling 10](#_Toc144727118)

[3.2.2 Outlier Clipping 10](#_Toc144727119)

[3.2.3 Logarithmic Transformation 11](#_Toc144727120)

[3.2.4 Standardization 11](#_Toc144727121)

[3.3 Design of Ridge Regression 11](#_Toc144727122)

[3.4 Kernel Ridge Regression 12](#_Toc144727123)

[3.5 Dataset 1 (salary\_data.csv) 13](#_Toc144727124)

[3.6 Dataset 2 (Boston Dataset) 13](#_Toc144727125)

[3.7 Bias-Variance Trade-off 15](#_Toc144727126)

[3.8 Overfitting and Underfitting 15](#_Toc144727127)

[3.9 Error Metrics 16](#_Toc144727128)

[3.10 Hyper-Parameter Tuning 17](#_Toc144727129)

[4 Analysis of Results 19](#_Toc144727130)

[4.1 Experiment 1: Ridge Regression (Dataset 1) 19](#_Toc144727131)

[4.2 Experiment 2: KRR (Dataset 1) 19](#_Toc144727132)

[4.3 Comprehensive Feature Scaling for RR in Dataset 1 20](#_Toc144727133)

[4.4 Comprehensive Feature Scaling for KRR in Dataset 1 21](#_Toc144727134)

[4.5 Ridge Regression on Dataset 2 23](#_Toc144727135)

[4.6 KRR on Dataset 2 23](#_Toc144727136)

[4.7 Comparison of Models: 24](#_Toc144727137)

[4.8 Batch mode and online mode 25](#_Toc144727138)

[5 Conclusion and Future Scope 27](#_Toc144727139)

[6 Self-Assessment 29](#_Toc144727140)

[6.1 Strengths 29](#_Toc144727141)

[6.2 Weaknesses 29](#_Toc144727142)

[6.3 Opportunities 29](#_Toc144727143)

[6.4 Threats 29](#_Toc144727144)

[7 References 31](#_Toc144727145)

[8 Appendix 33](#_Toc144727146)

# Introduction

In today's landscape, the terms 'Data' and 'Machine Learning' have become ubiquitous, signifying their central role in various industries and academic fields. The exponential growth in computational power and data availability has transformed machine learning from its roots in statistical analysis, which was the dominant method for making predictions in the 20th century. Early attempts to forecast phenomena like weather relied heavily on these statistical techniques, but their limitations became evident over time. With the advent of modern computational capabilities, there's a renewed optimism that machine learning models can reach unprecedented levels of accuracy and efficiency. The integration of data science and machine learning has become a focal point in various sectors, from healthcare to finance (Dhar, 2013). The transition from traditional statistical methods to machine learning is a natural progression facilitated by breakthroughs in computational power and data availability (Jordan and Mitchell, 2015).

Ridge Regression, first formulated by Hoerl and Kennard in 1970, serves as a pivotal method in statistical learning, addressing the inherent issue of multicollinearity in multiple linear regression models (Hoerl and Kennard, 1970). This technique's importance has persisted, even in the era of deep learning, and has found applications in a variety of fields such as bioinformatics and finance (Hastie et al., 2009; Tibshirani, 1996).

The resurgence of Ridge Regression in the context of big data is noteworthy. As datasets continue to grow in size and complexity, the challenges of multicollinearity become more pronounced, making Ridge Regression an invaluable tool for building robust models (James et al., 2013). This project aims to offer an in-depth exploration of Ridge Regression, touching upon its mathematical underpinnings, real-world applicability, and performance metrics under various conditions.

## Aim and Objectives

The primary aim of this project is to conduct an in-depth analysis and optimization of Ridge Regression for predictive modeling. The project seeks to explore the intricacies of Ridge Regression, from its theoretical underpinnings to practical applications, using various datasets. Additionally, the project aims to evaluate Ridge Regression's performance against other machine learning models and discuss the bias-variance trade-offs involved.

Objectives:

* Theoretical Understanding: To offer a comprehensive explanation of Ridge Regression, detailing its evolution from Linear Regression and the mathematical foundations that govern it.
* Data Preprocessing: To employ various data preprocessing techniques like scaling, normalization, and handling missing values, to prepare the datasets for model implementation.
* Hyperparameter Tuning: To systematically fine-tune the hyperparameters like learning rate, regularization strength, and maximum number of iterations, using cross-validation techniques to optimize the Ridge Regression model.
* Performance Metrics: To calculate the Mean Squared Error (MSE) for each model configuration to assess the model's performance and robustness.
* Comparison Study: To compare the performance of Ridge Regression with other machine learning models such as Kernel Ridge, Lasso, ElasticNet, and Random Forest, using the same datasets.
* Batch vs Online Learning: To analyze the effects of different learning modes (batch and online) on Ridge Regression performance, measured by MSE.
* Interpretability and Insights: To provide an interpretive layer to the model, explaining feature importance, and to offer insights into how Ridge Regression can be effectively used in various domains.
* Validation: To use established Python libraries such as scikit-learn to validate the findings of the custom-built Ridge Regression model.

## Significance of the Project

In a data-driven world profoundly impacted by events like the COVID-19 pandemic, the importance of robust predictive modeling techniques like Ridge Regression is increasingly evident (Wang et al., 2020). As organizations and institutions scramble to make data-informed decisions, the need for reliable and interpretable machine learning models has never been greater.

Healthcare Analytics: Ridge Regression has proven invaluable in healthcare analytics, particularly for predicting disease outcomes and drug discovery (Wang et al., 2020). In the post-2020 environment, healthcare systems are increasingly reliant on data analytics for timely and effective decision-making.

Supply Chain Optimization: The pandemic-induced disruptions to global supply chains have heightened the importance of predictive analytics (Ivanov, 2020). Ridge Regression can be applied for demand forecasting, thus contributing to more resilient supply chains.

Financial Risk Assessment: The volatile economic landscape has made financial risk modeling a priority (Kou et al., 2020). Ridge Regression's robustness and ability to handle multicollinearity render it well-suited for this domain.

Climate Change and Environmental Monitoring: As climate change remains a focal point of global conversations, predictive models like Ridge Regression are vital for environmental forecasting and disaster preparedness (Reichstein et al., 2019).

Cybersecurity: With escalating cyber threats, predictive modeling for network intrusion detection and other cybersecurity applications has gained importance (Garcia et al., 2020). Ridge Regression's computational efficiency makes it ideal for real-time analytics.

Ethical and Responsible AI: Given burgeoning concerns about the ethical implications of AI, the interpretability Ridge Regression offers is crucial for responsible AI deployments (Mittelstadt et al., 2019).

# Ridge Regression

## Overview of Machine Learning

Machine Learning, a cornerstone of artificial intelligence, aims to empower computer systems with the capacity for human-like intelligent behavior. As we delve into the age of data-driven decision-making, machine learning algorithms have permeated a multitude of sectors ranging from healthcare and finance to autonomous vehicles and cybersecurity. These algorithms generally follow one of two paradigms based on the type of data they are trained on: supervised learning and unsupervised learning. Each approach has its unique merits and is better suited for certain types of problems, thereby offering a wide canvas for problem-solving in various contexts.

Supervised learning is a dominant category in contemporary machine learning. In this paradigm, algorithms learn from labeled datasets, meaning each training example is paired with an output label. The algorithm iteratively makes predictions or decisions based on the input data, is corrected when its predictions are incorrect, and adjusts its model to make more accurate predictions in the future. As this learning process evolves, the algorithm becomes increasingly proficient at making accurate predictions. Supervised learning is versatile and can be employed in a myriad of applications. For example, in classification problems, it can be used to identify spam emails, diagnose diseases, or categorize images into predefined classes. On the other hand, in regression problems, it can predict stock prices, estimate house values, or forecast sales numbers. In essence, supervised learning is akin to a student learning under the guidance of a teacher; the algorithm learns to make decisions based on the feedback it receives for its actions.

Unsupervised learning, in contrast, deals with algorithms designed to identify patterns in datasets without labeled responses. The system tries to learn the patterns and the structure from the data without any supervision. Unsupervised learning algorithms are particularly useful for exploring the underlying structure of the data. They are employed extensively in clustering problems where the objective is to divide a data set into groups based on similarities among the data points. For instance, customer segmentation in marketing, social network analysis, or astronomical data analysis can be carried out effectively using unsupervised learning algorithms. Another intriguing application is in association rule mining, where the algorithm explores the data to find interesting relationships or associations among a set of items. This is frequently seen in market basket analysis where retailers can discover associations between different products purchased and thereby design more effective marketing strategies. Furthermore, unsupervised learning plays a pivotal role in dimensionality reduction, a technique that simplifies the input data by converting a high-dimensional space into a lower-dimensional form, while retaining the essential features of the data. This not only improves model performance but also provides valuable insights into the data.

In summary, machine learning is an expansive field with a wide array of techniques tailored for various kinds of problems. While supervised learning thrives on labeled data and excels in prediction and classification tasks, unsupervised learning leverages unlabeled data to unravel the inherent structure and relationships within the data. Each of these paradigms has its unique strengths and limitations, making them indispensable tools in the toolkit of anyone aiming to derive actionable insights from data.

## Linear Regression

Linear regression holds a venerable position in the realm of machine learning as a cornerstone technique for regression analysis. The algorithm's utility stems from its simplicity and interpretability, yet its applicability is contingent upon certain preconditions. Primarily, it presupposes a linear relationship between the dependent and independent variables. Moreover, the algorithm demands the absence of multicollinearity among predictors, no autocorrelation in the data, and a finite variance. Provided these conditions are met, linear regression can deliver powerful results with minimal computational overhead.

Mathematically, linear regression can be conceptualized as an equation that resembles a linear plane or a line. Formally, it is expressed as

where f(x) represents the predicted output corresponding to each Xi in the input vector X. The elements Xi are p-dimensional data points, and the vector X comprises n such points. The βi's are unknown parameters to be estimated, and β0 serves as a constant term. The objective in machine learning is to establish the relationship between the input and output variables by learning these parameters.

To gauge the model's performance, a loss function is essential. For linear regression, the Residual Sum of Squares (RSS) is commonly employed.

Loss function of the Linear regression model is Represented by

RSS =

=

The ultimate aim is to optimize the model by minimizing this loss function. To put it more explicitly, the goal is to find the optimal weights w and bias b that minimize the value of RSS. However, linear regression has its limitations, especially when confronted with real-world complexities that violate its underlying assumptions. One such issue that often plagues linear regression is multicollinearity. This problem, fortunately, finds a more robust solution in Ridge Regression, which serves as an extension and refinement of the classical linear regression model.

## Issues with Multicollinearity

One of the most pervasive challenges in linear regression is the issue of multicollinearity, which compromises the stability and interpretability of the model. In a typical regression scenario, the assumption is that altering an independent variable will consequently influence the dependent variable. However, multicollinearity emerges when there is a strong correlation among independent variables. This interconnectedness results in a situation where a change in one independent variable is inextricably linked to shifts in others, thereby clouding the unique contributions each makes to the dependent variable.

Multicollinearity manifests in two distinct forms: structural and data-driven. Structural multicollinearity arises from the inherent relationships among variables. For instance, if one independent variable is x and another is x2 , they are structurally linked; altering one will necessarily impact the other. Data-driven multicollinearity, on the other hand, is a product of the data sample itself. Either form of multicollinearity can lead to highly sensitive coefficient estimates in the linear equation. In other words, a minor change in one variable can induce drastic fluctuations in the calculated coefficients, rendering them statistically insignificant. To diagnose multicollinearity, one often uses statistical measures like Variance Inflation Factors (VIF).

Ridge Regression serves as an exemplary remedy for multicollinearity, adding a penalty term to the conventional Residual Sum of Squares (RSS) loss function. Mathematically, this is expressed as

where α is a hyperparameter introduced to regulate the loss function. This added term, often referred to as the shrinkage penalty, aims to mitigate the overfitting caused by multicollinearity. It essentially attenuates the influence of each variable's coefficient, w[j], on the loss function, thereby stabilizing the model.

Several critical observations can be drawn from this regularization term. When

α=0, the model reverts to ordinary least squares regression. Conversely, as

α approaches infinity, the coefficients w[j] tend towards zero. This penalty term represents the L2 norm or Euclidean distance of the coefficient vector from the origin and is often referred to as L2 regularization. One implication of using L2 regularization is that it moderates the coefficients of less important variables, making them approach zero but not exactly zero. Regression effectively counters the challenges posed by multicollinearity through the introduction of a regularization term. This approach is particularly beneficial for large datasets where multicollinearity is a recurring issue. By mitigating the adverse effects of multicollinearity, Ridge Regression enhances model stability and interpretability, making it a valuable tool in the machine learning toolkit for regression problems.

### Kernel Ridge Regression

Kernel Ridge Regression synergizes linear least squares with L2 norm regularization, offering an optimized approach to regression problems. The efficacy of this technique hinges considerably on hyperparameter tuning and the size of the training dataset. In this section, the mathematical underpinnings of Kernel Ridge Regression are elaborated upon.

The foundational assumptions for data representation mirror those in linear regression. For the loss function, we adopt the formula

Where is the actual value

is the predicted value

is the number of variables in the test set

Let n be the total number of observations, with in the test set and T in the training set such that n= +T. Given a training set ()….( ), IR. RSS is denoted by

To transition to ridge regression, a shrinkage penalty term is incorporated into the RSS equation, yield.ing

RSS = + ------------------------(1)

where is a positive constant.

Utilizing Vapnik's structural risk minimization framework, we derive the dual version of ridge regression. Substituting this into the modified RSS equation, we obtain

+ -----------------------------------------(2)

A Lagrange multiplier αt is introduced to form a new equation incorporating these terms. Optimization then proceeds through the Kuhn-Tucker theorem, ultimately yielding the dual formulation in terms of α.

+ + -------------------------(3)

The optimization of this dual formulation reveals the optimal values. we will equality when = .

-------------------------------(4)

Substitute (4) in (3),

+ + +

-

= + + - --(5)

Differentiating (5) in ,

------------------------------------(6)

Substituting (6) in (5),

- + ---------------- (7)

Differentiating (7) in , we will get the condition as below,

- -------------------------------(8)

where K is the T T matrix of dot product (Kernel Matrix)

Further decomposing of equation (8) will give the value which is,

----------------------------------------(9)

The prediction of on some new unlabelled example is,

Substituting from (4),

=

Substituting from (9)

= ().

ypredict = =

This is the most comprehensive derivation with regard to Ridge Regression

## Traditional Machine Learning Models

### Random Forest Regression

Random Forest Regression serves as a potent tool in machine learning for addressing regression problems. The algorithm operates on an ensemble approach, aggregating insights from multiple decision trees to make a more accurate prediction. Essentially, it computes the average of predictions from individual trees to arrive at an output that is likely to be close to the true value. One of the primary advantages of Random Forest Regression is its high accuracy. Its robustness is evident in its capability to perform well even when applied to large and complex datasets (Liaw and Wiener, 2002; Breiman, 2001).

### Support Vector Regression

Support Vector Machines (SVM) have been widely regarded as a pivotal classification algorithm in the field of machine learning (Cortes and Vapnik, 1995; Schölkopf et al., 2001). Its regression counterpart, Support Vector Regression (SVR), adapts the foundational principles of SVM to predict continuous output variables. SVR functions by finding an optimal fit between the initial and threshold lines, with key parameters such as the kernel, hyperplane, and boundary lines significantly influencing model performance (Drucker et al., 1997; Smola and Schölkopf, 2004).

### K Nearest Neighbour Regression

K-Nearest Neighbors (KNN) is a well-regarded classification algorithm celebrated for its simplicity and interpretability (Cover and Hart, 1967; Altman, 1992). For regression tasks, the algorithm operates by computing the average distance among the 'k' closest data points based on a chosen distance metric. In this context, 'k' serves as a hyperparameter that can be fine-tuned to minimize the mean squared error (Dudani, 1976).

### Lasso

Likewise, Lasso (Least Absolute Shrinkage and Selection Operator) is a regression technique that not only minimizes the sum of squared residuals but also imposes a penalty on the magnitude of the coefficients, thereby encouraging sparsity (Tibshirani, 1996). ElasticNet is a hybrid technique that combines the strengths of Ridge and Lasso regression by adding both L1 and L2 regularization terms to the loss function (Zou and Hastie, 2005). These methods offer robust solutions for regression problems with different characteristics and constraints.

The Loss function for Lasso is represented by

ElasticNet is the combination of Lasso and Ridge regression. In this we add both L1 norm and L2 norm of the coefficient vector. Here there will be two hyper parameters, both β and α will be the hyper parameters. The equation for elastic net is given by

# Design and Methodology

## Selection of Tools

The curriculum of the course has been instrumental in broadening my skill set across multiple programming languages, notably MATLAB, R, and Python. Initially, I considered conducting this experiment in R due to its statistical robustness and data visualization capabilities. However, as I delved deeper into the project, Python emerged as the more fitting choice. My familiarity with Python, gained through extensive prior work, offered me a level of confidence that was indispensable for this research. Additionally, Python's comprehensive machine learning libraries, such as scikit-learn, presented the opportunity to validate my project outcomes against established algorithms. This decision not only streamlined the development process but also enhanced the credibility of my results, thereby reinforcing Python as the optimal language for this endeavour.

## Data Pre-Processing Techniques

In data modeling, normalization stands as a pivotal preprocessing step, particularly salient when the feature set exhibits heterogeneity in scale or units. The primary objective is to standardize disparate variables into a uniform scale, thereby enhancing the numerical stability of the model's computations and fortifying the reliability of its predictive inferences. Various forms of normalization may be invoked, contingent upon the domain-specific requirements, value range stipulations, and the data's statistical distribution. Predominant normalization techniques encompass Min-Max Scaling, Outlier Clipping, Logarithmic Transformation, and Standardization (Z-Score Normalization).

### Min-Max Scaling

Min-Max Scaling is typically employed when there is a pronounced discrepancy in the range among various features. In this scheme, each feature value is transmuted to lie within a specified range, generally between 0 and 1. Mathematically, this is formulated as

Xnormalized

This methodology is computationally efficient and is most efficacious when the dataset is devoid of extreme outliers and adheres to a uniform distribution.

### Outlier Clipping

Outlier Clipping is recommended in instances where the dataset is replete with anomalous values. By designating an upper (and occasionally lower) bound for feature values, this technique proficiently attenuates the outlier-induced skewness.

Upper Bound=Q3+1.5×IQR

Lower Bound=Q1−1.5×IQR

where Q1 and Q3 are the first and third quartiles, respectively, and

IQR=Q3−Q1.

Xclipped = { Upper bound if x>Upper bound,

Lower bound if x<Lower Bound,

x otherwise

### Logarithmic Transformation

Logarithmic Transformation is advantageous when the feature distribution is heavily skewed, encompassing a preponderance of values within a constrained range juxtaposed with sparser values in an extended range. This method enhances the interpretability by redistributing the data along a logarithmic scale.

x ′=log(x+c)

### Standardization

Z Standardization, commonly referred to as Z-Score Normalization, is often the preferred technique when the dataset contains a moderate presence of outliers. It transmutes the feature distribution into a Gaussian distribution with a mean (μ) of zero and a standard deviation (σ) of one, as per the formula score is one of the popular techniques of normalization used, this is more appropriate in practical standpoint of view, as this is used when there are outliers but not so many, this makes the distribution convert into a gaussian distribution, basically meaning the mean to 0 and the standard deviation to 1.

Xstandardised = x- µ / Ω

## Design of Ridge Regression

The `RidgeRegression` class is a Python implementation designed to execute Ridge Regression, a popular variant of linear regression that includes L2 regularization. It employs gradient descent to optimize the cost function. The class is modular and contains methods for initialization, model fitting, prediction, and utility functions for adding intercepts and computing gradients. The class offers flexibility with hyperparameters like learning rate, regularization strength, and maximum iterations, making it versatile for different use-cases.

The \_\_init\_\_ method sets the initial state of the RidgeRegression object. The hyperparameters learning\_rate, reg\_strength, and max\_iter are set during the initialization. Additionally, a placeholder None is used for weights, which will later be updated during the fit method. The fit method is where the core functionality resides. First, it adds an intercept term to the feature matrix X by invoking the \_add\_intercept method. It then initializes the weights to zeros and iteratively updates them using gradient descent. The gradient is computed using the \_compute\_gradient method, which incorporates ridge regularization. The predict method takes a feature matrix X as input, adds an intercept term, and then multiplies it by the trained weights to generate the predicted output. This is a straightforward implementation of the formula y=Xβ for multiple linear regression. The utility methods serve specific roles that support the primary methods. \_add\_intercept adds an intercept term to the feature matrix, enabling the model to fit for a constant term. \_compute\_gradient computes the gradient of the Ridge Regression cost function, taking into account both the data loss and the regularization term.

The code is simple yet versatile, offering flexibility in hyperparameter tuning. The use of gradient descent makes it scalable, and ridge regularization helps control overfitting. The gradient descent approach is susceptible to local minima, although this is less of an issue for convex problems like Ridge Regression. Manual tuning of hyperparameters can be time-consuming. Additionally, Ridge Regression doesn't eliminate irrelevant features but only minimizes their weights. The model assumes a linear relationship between the features and the target. It also lacks a convergence check for gradient descent, making it uncertain whether the optimal solution is reached within the given number of iterations. Lastly, the code doesn't employ any memory optimization strategies, which could be a concern for extremely large datasets.

## Kernel Ridge Regression

The KernelRidgeRegression class is designed to implement Kernel Ridge Regression, an extension of Ridge Regression that leverages the kernel trick to model complex, non-linear relationships. The class provides flexibility in choosing different types of kernels—linear, polynomial, radial basis function (RBF), and quadratic—along with their corresponding parameters. It employs closed-form solutions for fitting and prediction, based on the computed kernel matrices. This makes the class highly versatile and adaptable for tackling both linear and non-linear problems.

The \_\_init\_\_ method sets up the initial configuration of the class. The kernel parameter allows you to specify which type of kernel to use. The C parameter is the regularization strength, similar to Ridge Regression. The gamma and degree parameters are specific to RBF and polynomial kernels, respectively. Finally, alphas serves as a placeholder for the model's coefficients, which are determined during the fitting process. The fit method is responsible for training the model. It first computes the kernel matrix K for the training data using the \_compute\_kernel\_matrix method. Then, it solves for the coefficients alphas using the closed-form equation (K+C⋅I)α=y. The predict method uses the trained alphas to make predictions. It computes a new kernel matrix K between the test data and the training data. Then, it multiplies this K with alphas to obtain the predictions. The \_compute\_kernel\_matrix method is a utility function that computes the kernel matrix based on the specified kernel type and its parameters. It supports four types of kernels and computes the matrix accordingly.

The class offers tremendous flexibility in choosing different types of kernels, which makes it suitable for a wide array of problems, including non-linear ones. The use of closed-form solutions avoids iterative optimization. Kernel Ridge Regression can be computationally expensive for large datasets due to the need to invert matrices. Also, like many machine learning models, it requires careful hyperparameter tuning. The model's performance can be highly sensitive to the choice of hyperparameters like C, gamma, and degree. Additionally, the method is not particularly scalable for extremely large datasets because of its O(N3) computational complexity for matrix inversion. This problem generally occurs in Kernel Ridge Regression.

## Dataset 1 (salary\_data.csv)

The `salary\_data.csv` dataset typically serves as a foundational piece for understanding the relationship between years of experience and salary. Consisting of two primary columns—'YearsExperience' and 'Salary'—this dataset is generally straightforward yet incredibly informative. The 'YearsExperience' column provides a numerical representation of the work experience, usually in years, for each individual in the dataset. On the other hand, the 'Salary' column gives the corresponding compensation, commonly in a currency like dollars. The dataset's simplicity makes it an excellent tool for introductory data analysis and machine learning exercises, offering a clean and uncomplicated way to explore how one variable influences another.

Analyzing this dataset with regression models provides dual benefits. First, it serves as a performance gauge for the model itself. Because the dataset is usually clean and simple, any underperformance of the model can often be attributed to the algorithm rather than complexities or noise in the data. This makes it an ideal benchmark dataset. Second, the dataset can deliver valuable insights into workforce compensation trends. By fitting a regression model to this data, one can predict the salary for a given experience level, which is invaluable for job seekers, employers, and policy-makers alike. Even though the dataset is simple, the analysis can be quite rich, shedding light on the rate at which salary increases with experience, identifying any non-linear jumps in salary at certain experience thresholds, and potentially revealing biases if the data is segmented by other factors like location, gender, or industry.

## Dataset 2 (Boston Dataset)

The Boston Housing dataset is one of the most iconic datasets in the field of machine learning and data science. Comprising 13 features and a target variable that represents median house values, the dataset was initially collated in the late 1970s as part of a U.S. census. The dataset’s features encompass a diverse range of variables, including crime rate, average number of rooms per dwelling, accessibility to highways, and even an index measuring air quality. The target variable, usually denoted as 'MEDV,' signifies the median value of owner-occupied homes, measured in thousands of dollars. This dataset is incredibly versatile, offering a multi-dimensional view of what contributes to housing prices, and it has been utilized extensively both for educational and research purposes.

The Boston Housing dataset serves as a valuable resource for multiple reasons. From a model performance standpoint, its diverse set of features makes it an excellent ground for evaluating the effectiveness of various regression algorithms. The dataset's features have different scales, units, and distributions, which makes it a good case for practicing data normalization and transformation techniques, like log transformations or scaling. It's also a great dataset for feature selection and engineering tasks; for instance, one could create new interaction terms between features like 'average number of rooms per dwelling' and 'crime rate' to explore if they collectively explain the variance in the housing price better than when used individually. From an analytical angle, the dataset provides a snapshot of urban housing issues that were pertinent during the late 20th century, many of which remain relevant today. By analyzing the data, one can gain insights into how different neighborhood characteristics affect house pricing. It can help urban planners, policy makers, and even individual homebuyers make informed decisions. Moreover, by segmenting or clustering the data based on specific features, one could identify outlier neighborhoods that defy general trends, which could be a starting point for more localized studies. In essence, the Boston Housing dataset is a rich, multi-faceted resource that allows for both rigorous model evaluation and deeply insightful societal analysis.

The Boston Housing dataset is particularly remarkable for the breadth of its features, each of which offers a distinct lens through which to understand housing prices. For example, the 'CRIM' feature measures per capita crime rate, shedding light on the role of safety in housing costs. 'ZN' represents the proportion of residential land zoned for large lots, providing insights into urban planning. 'INDUS' describes the proportion of non-retail business acres per town, offering a view into the impact of commercial development. 'CHAS' is a binary feature indicating whether the tract bounds the Charles River, revealing the premium placed on waterfront properties. 'NOX' measures nitrogen oxides concentration, serving as a proxy for environmental quality. 'RM,' 'AGE,' 'DIS,' 'RAD,' 'TAX,' 'PTRATIO,' 'B,' and 'LSTAT' cover aspects like average rooms, age of the property, distances to employment centers, accessibility to radial highways, property tax rate, pupil-teacher ratio, proportion of Black residents, and percentage of lower status population, respectively. Each feature is a piece of the puzzle, and together they provide a comprehensive view of the factors influencing housing prices. This multifaceted nature of the dataset makes it not only a challenging task for aspiring data scientists but also a resource for nuanced socio-economic analyses.

## Bias-Variance Trade-off

Bias refers to the systematic error that manifests between the predicted and actual values in a machine learning model. It can arise from a multitude of factors, such as different data splits due to varying random states, and serves as a measure of how far off the model's general predictions are from reality. Developers must actively work to mitigate not just algorithmic bias but also any human biases that could compromise model performance. Identifying bias involves scrutinizing various indicators, such as missing feature values, unusual data points beyond outliers, and data skewness. These factors may not guarantee the presence of bias, but they raise the likelihood. Variance, on the other hand, quantifies a model's prediction variability. In essence, it tells us how much the model's predictions would differ for the same input point when trained on different subsets of the data. High variance often indicates that the model is overly sensitive to the training data, including its noise and fluctuations, which can lead to overfitting.

The concept of the bias-variance trade-off serves as a cornerstone in machine learning model evaluation. High-bias models are generally simplistic, leading to increased errors and underfitting, whereas low-bias models tend to capture the underlying trends more accurately but risk overfitting. Importantly, bias and variance are inversely related, complicating the optimization process. Achieving a model with both low bias and low variance is often an impractical ideal. Solutions to this quandary might involve increasing model complexity, thereby lowering bias at the cost of increased variance, or augmenting the training dataset size to mitigate high variance. The balance between bias and variance is a nuanced decision that varies depending on the specific challenges and requirements of a project. Scott Fortman-Roe's insights on this topic underscore the importance of carefully navigating this trade-off. The ideal model occupies a 'sweet spot' that minimizes both bias and variance to the extent possible, thereby optimizing its predictive performance.

## Overfitting and Underfitting

Overfitting represents a prevalent challenge in machine learning, characterized by a disproportionately high test or validation loss relative to the training loss. This issue commonly arises when a model inadvertently learns patterns that lack generalizability. An effective model should perform well on both training and test sets. Counteracting overfitting requires a well-rounded training dataset, not necessarily voluminous but comprehensive enough to facilitate generalization. This becomes a complex task, given that data is often randomly sampled and the optimal subset for generalization is elusive. Hence, rigorous experimentation is crucial. Overfitting is closely tied to a model's "memorization capacity"; models with high memorization capacity risk becoming mere mapping functions, directly associating input values with output predictions without truly

Shape, circle

Description automatically generated"learning," thereby exhibiting high variance.

**Figure 2: BIAS VARIANCE TRAD E OFF**

Conversely, underfitting occurs when a model fails to capture enough parameters, leading to indiscriminate predictions and resulting in high bias. Striking a balance between overfitting and underfitting is pivotal. The model must absorb all necessary parameters to generalize effectively, yet the precise trade-off is elusive and largely determined through experimental means. There's no universal formula to navigate this conundrum; it necessitates iterative trials with varying training sizes to identify the point just before overfitting occurs, thereby optimizing the model for production use.

## Error Metrics

Error metrics serve as the quantitative benchmarks to evaluate the performance of machine learning models. These metrics provide critical insights into the areas where a model's performance can be enhanced. While accuracy serves as a straightforward metric for classification problems, it falls short in capturing the nuances of regression problems. In regression, the key is to understand how close the model's predictions are to the actual values. Among the commonly used error metrics for regression are Mean Square Error (MSE), Root Mean Square Error (RMSE), and Mean Absolute Error (MAE). Each of these has its own set of advantages and disadvantages.

### Mean Square Error (MSE)

MSE is perhaps the most widely-used error metric in regression analyses, primarily because it is also employed as a loss function in many machine learning algorithms. One of its principal advantages is its non-negativity; the MSE can be zero (indicating a perfect model) or a positive value (indicating the extent of error). Another benefit is its straightforward interpretation: a higher MSE value suggests that the predicted values are far from the actual ones, while a lower MSE signifies that the predictions are accurate. MSE is particularly useful when dealing with Gaussian-distributed errors or when the outliers are not a concern, as it squares the errors and hence gives higher weight to larger errors. However, one drawback is that the MSE units are squared, making them less intuitively interpretable than the original data points.

Root Mean Square Error (RMSE)

RMSE is essentially the square root of MSE. It shares many characteristics with MSE, such as non-negativity and ease of interpretation. The primary advantage of RMSE over MSE is that it is in the same unit as the output variable, making it more interpretable. It indicates the standard deviation of the residuals and provides an idea of the spread of the errors. However, like MSE, RMSE is sensitive to outliers, meaning a single large error can significantly skew the RMSE value, potentially leading to misleading interpretations.

Mean Absolute Error (MAE)

MAE is another robust error metric often used in regression problems, especially in time-series forecasting. One of the key advantages of MAE is its linearity; it doesn't penalize large errors as severely as MSE or RMSE. This makes MAE more robust to outliers. It's also non-negative and gives a direct indication of how well the model is performing. However, because it doesn't square the errors, it might not reflect the impact of larger errors adequately, which could be a disadvantage in many applications where such errors cannot be ignored.

Conclusion on the Selection of MSE

Given the characteristics of each metric, for this project, I have selected MSE over RMSE and MAE. The choice is motivated by several factors, including MSE's computational efficiency and suitability for small datasets. Furthermore, MSE is highly compatible with optimization algorithms commonly used in machine learning, making it an attractive choice for a broad range of applications. The squaring of the error terms in MSE allows the model to focus more on points that are hard to predict, which could be crucial for certain applications. Therefore, based on these considerations, MSE was deemed the most appropriate error metric for this project.

## Hyper-Parameter Tuning

Hyperparameter tuning is an essential step in the machine learning pipeline, particularly for complex algorithms where the optimal settings are not evident. The right hyperparameters can make the difference between mediocre and state-of-the-art performance. Several techniques are widely used for hyperparameter optimization, including Grid Search, Random Search, and Bayesian Optimization. Each of these methods has its own merits and drawbacks, which are important to consider based on the specific needs of the project.

Grid Search

Grid Search is perhaps the most straightforward method for hyperparameter tuning. It involves specifying a set of values for each hyperparameter and then exhaustively evaluating all possible combinations. For instance, if one hyperparameter has 5 possible values and another has 4, Grid Search would evaluate 20 different models. The key advantage of Grid Search is that it is guaranteed to find the optimal combination of hyperparameters within the specified range. However, this comes at the cost of computational resources, especially for models with many hyperparameters or large datasets. Additionally, the method assumes that the optimal values of hyperparameters lie on a grid, which might not always be the case.

Random Search

Random Search is a probabilistic method that randomly selects combinations of hyperparameters to evaluate. Unlike Grid Search, it does not explore all possible combinations but randomly samples from the hyperparameter space. This approach can be computationally more efficient and often finds a good set of hyperparameters more quickly than Grid Search. However, there's no guarantee that Random Search will find the most optimal set of hyperparameters, especially if the number of iterations is too low.

Bayesian Optimization

Bayesian Optimization takes a probabilistic approach to hyperparameter tuning. It constructs a probability model of the objective function and uses it to select the most promising hyperparameters to evaluate in the true objective function. This method is particularly useful when the evaluation of the objective function is expensive in terms of time or resources. Bayesian Optimization aims to find the optimal hyperparameters by making smarter choices and therefore often requires fewer evaluations compared to Grid Search or Random Search. However, the method can be complex to set up and may not be suitable for all types of objective functions.

Conclusion on the Selection of Hyperparameter Tuning Technique

Selecting the appropriate hyperparameter tuning technique depends on various factors such as computational resources, the complexity of the model, and the specific requirements of the project. Grid Search, with its exhaustive nature, is generally more accurate but computationally intensive. Random Search and Bayesian Optimization offer more efficient but probabilistic solutions. For projects where computational resources are limited, or a "good-enough" solution is acceptable, Random Search or Bayesian Optimization may be more appropriate. On the other hand, if computational resources are abundant and the highest level of accuracy is required, Grid Search would likely be the method of choice. Therefore, the choice of hyperparameter tuning technique should be made carefully, considering the trade-offs between computational efficiency and solution quality.

# Analysis of Results

## Experiment 1: Ridge Regression (Dataset 1)

The analysis of the output table reveals intricate relationships between the hyperparameters and the model's performance, as quantified by the Mean Squared Error (MSE). Starting with the learning rate, it is evident that this parameter plays a significant role in the speed of convergence and the overall performance of the Ridge Regression model. Specifically, a ten-fold increase in the learning rate from 0.0001 to 0.001 leads to a substantial reduction in MSE, from the order of \(10^9\) to \(10^8\), even when the number of iterations and regularization strength are held constant. This suggests that the model is highly sensitive to the learning rate, and that a higher learning rate enables the model to converge to a better solution more quickly.

The impact of the regularization strength is subtler. While one might expect stronger regularization to yield higher MSE due to a bias-variance tradeoff, the data doesn't show a clear pattern. For instance, an increase in regularization strength from 0.01 to 1 at a learning rate of 0.0001 doesn't result in a consistent trend in MSE values. This could imply that the model is relatively robust to the choice of regularization strength within the examined range, or that the interplay between regularization and learning rate is more complex than can be captured by a single metric like MSE.

The number of iterations, or 'Max Iterations', also presents an interesting aspect. While more iterations generally lead to lower MSE, the rate of improvement diminishes. For example, at a learning rate of 0.01 and regularization strength of 0.01, the MSE hardly changes when increasing the number of iterations from 500 to 1000. This is indicative of the model reaching a point of diminishing returns in terms of MSE reduction, suggesting that additional iterations beyond a certain point may not provide significant benefits.

## Experiment 2: KRR (Dataset 1)

The focus here is on understanding the performance dynamics across different kernel types—linear, polynomial, RBF, and quadratic—under varying conditions set by hyperparameters like C, γ, and degree.

The linear kernel exhibits a relatively constant mean squared error (MSE) ranging from 5.98×109 to 6.06×109, regardless of the changes in C and γ. This suggests that a linear approximation of the underlying data might be inadequate, as these hyperparameters should ideally show some effect if they were capturing the data's complexity.

When it comes to the polynomial kernel, there's a noticeable fluctuation in MSE, spanning from 3.18×10 7 to as high as 8.02×10 8 The wide range indicates a high sensitivity to the polynomial degree and the regularization parameter C. Specifically, as the degree and C increase, the MSE also rises, hinting at a potential overfitting scenario where the model is too complex for the given data.The radial basis function (RBF) kernel displays a significant sensitivity to both C and γ, with MSEs ranging from 3.80×10 7 to 3.77×10 9 . A crucial observation is that the MSE increases as both C and γ rise, suggesting that the model might be fitting the noise instead of the underlying pattern in the data.

Finally, the quadratic kernel's performance is more stable, with MSE values between 3.55×10 7 and 7.07×10 8. Unlike the polynomial and RBF kernels, the quadratic kernel shows less sensitivity to the hyperparameters, indicating a balanced trade-off between model complexity and fitting quality. In essence, Experiment 2 offers a granular view into how Kernel Ridge Regression models react to different kernels and hyperparameters. While linear kernels seem limited in capturing data complexity, polynomial and RBF kernels are sensitive to hyperparameters and prone to overfitting or noise fitting. The quadratic kernel, on the other hand, offers a more stable performance. This intricate analysis aids in understanding the challenges in hyperparameter tuning for Kernel Ridge Regression models.

## Comprehensive Feature Scaling for RR in Dataset 1

Certainly. The experiment aimed to explore the impact of varying hyperparameters—specifically learning rate, regularization strength, and maximum iterations—on the performance of Ridge Regression when applied to a fully scaled dataset. The results revealed a distinct pattern of influence each hyperparameter holds over the model's accuracy, as measured by the Mean Squared Error (MSE). The lowest MSE was achieved with a learning rate of 1\* 10\*\*-2, a regularization strength of 0.01, and a maximum of 5000 iterations. This particular combination emerges as the most effective, warranting specific attention.

A screenshot of a graph

Description automatically generated

The learning rate of 1 \* 10\*\*-2 appears to strike an ideal balance in the speed of convergence. Lower learning rates typically demand more iterations to converge, increasing the computational time. On the contrary, higher learning rates might overshoot the optimal values, leading to erratic performance. Therefore, this choice of learning rate proves to be optimal for fast yet accurate convergence. The regularization strength of 0.01 adds another layer to this optimization. Regularization serves as a control measure against overfitting by penalizing large coefficient values. A value too small would make the model susceptible to overfitting, while a value too large might lead to underfitting. In this case, 0.01 seems to be a balanced regularization strength that neither over-penalizes nor under-penalizes the model, contributing to its high accuracy.

A graph with red and blue dots

Description automatically generated

Lastly, the maximum iterations parameter set at 5000 rounds off the hyperparameter optimization. With a moderate learning rate and regularization, the model likely needs a sufficient number of iterations to reach or approximate the global minimum. It is plausible that for the chosen learning rate and regularization strength, 5000 iterations provide ample time for the model to stabilize and converge towards the optimal solution. To summarise, the harmonious interplay among a moderate learning rate, balanced regularization strength, and sufficient number of iterations culminates in the most accurate Ridge Regression model for this dataset, as evidenced by the lowest MSE. This underscores the importance of hyperparameter tuning in achieving optimal model performance.

## Comprehensive Feature Scaling for KRR in Dataset 1

The hyperparameters under consideration include the type of kernel (linear, polynomial, rbf, quadratic), regularization parameter (C), gamma for RBF kernel, and the degree for the polynomial kernel. The feature set and the target variable were fully scaled to a standard scale before training the model.

A screenshot of a math table

Description automatically generated

A graph with red and blue dots

Description automatically generatedThe results indicate a variety of Mean Squared Error (MSE) values, ranging from as low as 0.065 for linear kernels to higher values for other types of kernels. The low MSE suggests that the linear kernel type has performed exceptionally well in this context. It's noteworthy that the MSE remained constant across different values of 'Gamma' for a linear kernel, emphasizing that 'Gamma' does not influence the linear kernel. On the contrary, the 'rbf' and 'polynomial' kernels showed a significant change in MSE with the tuning of hyperparameters. Specifically, higher degrees in polynomial kernels and varying 'Gamma' in RBF kernels influenced the MSE, sometimes detrimentally. In terms of the regularization parameter 'C', lower values generally performed better, indicating that a more regularized model was beneficial in this scenario. For instance, with the linear kernel, the MSE was the lowest when C was 0.1, regardless of the Gamma value.

## Ridge Regression on Dataset 2

The experiment is designed to scrutinize the performance of Ridge Regression on the Boston Housing dataset, a widely-used dataset for regression tasks that contains 13 features related to housing and one target variable, the median value of homes (MEDV). The dataset is divided into an 80-20 split for training and testing. Notably, only the input features are scaled using standard scaling, while the target variable remains unscaled. The objective is to identify the optimal combination of hyperparameters that minimizes the mean squared error (MSE). The hyperparameters explored in this experiment include learning rates ranging from10^-6 to 0.1, regularization strengths from 0.001 to 100, and maximum iterations for the gradient descent varying from 100 to 5000. Each combination is subjected to cross-validation on the training set to evaluate its generalizability.

The results indicate that the best-performing hyperparameter combination is a learning rate of 0.1, regularization strength of 1, and a maximum of 5000 iterations. These parameters yield an MSE of approximately 23.65, which is relatively low given the complexity and dimensionality of the Boston Housing dataset. This specific hyperparameter set suggests a balanced approach—neither too aggressive nor too lax—in terms of learning rate and regularization. The need for 5000 iterations indicates that the model requires ample time to converge to an optimal solution, possibly due to the multi-dimensional feature space. This experiment exemplifies the sensitivity of Ridge Regression to hyperparameters, especially in a high-dimensional dataset like Boston Housing. It also shows that feature scaling restricted to input variables can still result in a well-performing model, underscoring the versatility of Ridge Regression in handling datasets with varying scales and dimensions.

## KRR on Dataset 2

The experiment focuses on Kernel Ridge Regression applied to the Boston Housing dataset, again with only input features scaled using standard scaling. The dataset comprises 13 features related to housing conditions and one target variable, the median value of homes (MEDV). It is split into an 80-20 ratio for training and testing. The objective is to identify the best hyperparameters that minimize the MSE.

For this model, the hyperparameters under investigation are the types of kernels ('linear,' 'polynomial,' and 'rbf'), regularization parameters C ranging from 0.01 to 100, and gamma values varying from 0.01 to 10. Each combination of hyperparameters is evaluated using cross-validation on the training set.

The results reveal that the best-performing hyperparameters are an 'rbf' kernel, a C value of 0.01, and a gamma value of 0.01. These parameters produce an MSE of approximately 11.04, which is notably lower than the MSE obtained in the Ridge Regression experiment. The 'rbf' kernel with low C and gamma values suggests that the model requires a high degree of regularization and is sensitive to A graph of a training and validation

Description automatically generated with medium confidencethe similarity between data points in the feature space. The low MSE indicates that the Kernel Ridge Regression model with these parameters is highly effective in capturing the complexities of the Boston Housing dataset. Kernel Ridge Regression with an 'rbf' kernel and specific regularization and similarity parameters offers a superior model for this particular dataset. It not only outperforms the Ridge Regression model but also adapts well to the high-dimensional feature space, providing a more accurate prediction of housing prices in Boston.

## Comparison of Models:

The final experiment provides a comprehensive comparison of different regression models applied to the Boston Housing dataset. The models compared are the custom Ridge and Kernel Ridge models, their Scikit-learn equivalents, and additional models such as Lasso, ElasticNet, Linear Regression, Support Vector Regression (SVR), and Random Forest Regressor. All models are trained on the same feature-scaled training data, and their performance is evaluated based on the Mean Squared Error (MSE) on the test set.

The MSE values are a direct measure of a model's ability to generalize well to new, unseen data. Among the models compared, the Random Forest Regressor shows the lowest MSE at approximately 8.59, indicating the best predictive accuracy. It is followed closely by the custom Kernel Ridge model with an MSE of around 10.53 and the Sklearn Kernel Ridge model with an MSE of about 11.52. These are the top three performers, significantly outperforming the other models in terms of MSE.

A graph of a graph with text

Description automatically generated with medium confidenceInterestingly, Linear Regression and its regularized counterparts (Ridge and Lasso) have higher MSE values ranging from 24.29 to 27.58, suggesting that they are not as effective at capturing the underlying complexities of the dataset. The SVR and ElasticNet models perform the worst, with MSE values of 25.67 and 28.36, respectively.

To sum up, this experiment reveals that ensemble methods like Random Forest and non-linear models like Kernel Ridge are more adept at navigating the feature space of the Boston Housing dataset, thereby providing more accurate predictions. The custom implementations also hold their ground well when compared to their Sklearn counterparts, particularly the custom Kernel Ridge model, which almost matches the performance of the Random Forest model. This comparison serves as a robust testament to the effectiveness of these machine learning algorithms in tackling real-world regression problems.

## Batch mode and online mode

The final experiment in the series aims to dissect the performance of custom Ridge Regression under two different learning modes: batch and online. Both models are initialized with the same optimal hyperparameters derived from previous experiments. The batch learning mode updates the weights based on the entire dataset in each iteration, whereas the online mode (also known as Stochastic Gradient Descent) updates the weights sample-by-sample. Analyzing the Mean Squared Error (MSE) for each model reveals a stark contrast in their predictive accuracies. The batch mode Ridge Regression model achieves an MSE of approximately 24.31, which is consistent with the performance observed in earlier experiments. On the other hand, the online mode Ridge Regression model displays a significantly higher MSE of around 772.05.

This difference in performance can be attributed to several factors. Batch learning is often more stable, as it calculates the gradient based on the entire dataset, thereby reducing the variance in weight updates. Online learning, while computationally more efficient per iteration, can be more susceptible to the noisy or stochastic nature of single-sample updates. This can lead to a more erratic search in the weight space, which may explain the elevated MSE in this case. It's worth noting that while online learning generally requires more passes over the data (or epochs) to converge to a good solution, the experiment used the same maximum number of iterations for both batch and online modes. This could potentially disadvantage the online mode, as it might not have had enough iterations to converge to a more optimal solution.

# Conclusion and Future Scope

Ridge Regression holds a venerable position in the machine learning landscape due to its straightforward algorithmic architecture and ease of interpretation. In this thesis, I commenced by introducing the foundational concepts of Ridge Regression and providing an in-depth examination of the dataset's domain-specific background. This intellectual groundwork paved the way for a comprehensive discussion of Ridge Regression's historical development as an extension of Linear Regression. Utilizing an artificially generated dataset, I meticulously calculated the requisite metrics for Ridge Regression, successfully deriving the algorithm's formula. The ensuing predictions for output labels proved remarkably accurate, thereby validating the model's efficacy.

In the second phase the series of experiments conducted offered a deep dive into the performance and characteristics of Ridge Regression and Kernel Ridge Regression models, using both custom implementations and standard libraries. By systematically varying hyperparameters and assessing their impacts on predictive performance, these experiments have yielded valuable insights into model behavior and optimization. Starting with Experiment 1, which focused on Ridge Regression with scaled input features, we observed that the learning rate played a pivotal role in convergence speed, while the regularization strength primarily influenced overfitting. The lowest MSE was achieved with a learning rate of 0.1, regularization strength of 1, and 5000 iterations. Experiment 2 further expanded the scope by introducing Kernel Ridge Regression. This experiment demonstrated that the type of kernel used could substantially influence the model's performance. Among the examined kernels, the Radial Basis Function (RBF) with specific hyperparameters yielded the lowest MSE, significantly outperforming linear and polynomial kernels.

Experiment 3 and 4 transitioned to a more complex dataset, the Boston Housing dataset, and incorporated additional models like Lasso, Elastic Net, and Random Forest for comparative evaluation. The custom Kernel Ridge model, when tuned with optimal hyperparameters, came close to the performance of a Random Forest model, emphasizing its efficacy. The final experiment delved into the learning modes of Ridge Regression—batch and online. The batch mode outperformed the online mode significantly, highlighting the importance of choosing the right learning strategy based on dataset characteristics and computational resources.

The importance of hyperparameter tuning cannot be overstated. Each change in hyperparameters like learning rate, regularization strength, or kernel type led to noticeable shifts in performance. While Ridge and Kernel Ridge Regression are powerful techniques, they do not universally outperform other models. As seen in Experiment 4, ensemble methods like Random Forest can sometimes offer better predictive accuracy. The learning mode, as shown in the final experiment, plays a crucial role in determining the model's effectiveness. Batch learning, in this case, offered more stability and lower error rates compared to online learning.

One of the most immediate extensions would be to delve into alternative regularization techniques that could be used in conjunction with Ridge. While our experiments have provided comprehensive insights into the behavior of Ridge Regression under different hyperparameters, the use of advanced hyperparameter tuning methods such as Bayesian Optimization could provide more efficient and potentially more effective model configurations. Feature engineering is another aspect that has not been fully explored in the current experiments. We've used all available features in the datasets, but a systematic analysis focusing on feature selection could yield more parsimonious models without sacrificing predictive power. This could also alleviate issues related to multicollinearity among predictors, which is particularly pertinent to Ridge Regression. Adding polynomial features could be another avenue, especially given that Ridge Regression can handle the increased complexity without as much risk of overfitting as some other models. From a computational standpoint, the efficiency of Ridge Regression algorithms could be significantly improved. Our experiments highlighted the importance of the number of iterations in the optimization process, suggesting that parallel computing or more advanced optimization techniques could reduce the computational burden. Interestingly, the online version of Ridge Regression in our experiments performed poorly compared to its batch counterpart, raising questions about the potential benefits of stochastic methods in Ridge Regression, a subject that would be worthwhile to investigate further.

# Self-Assessment

## Strengths

My affinity for algorithmic problem-solving and mathematical modeling served as the driving force throughout this Ridge Regression project. The foundational skills acquired from coursework, particularly in machine learning and statistical methods, played a pivotal role in approaching and resolving the complex issues inherent to Ridge Regression. My coding skills, honed through rigorous practice and academic exposure, were crucial in implementing the Ridge Regression model from scratch. This was especially beneficial in fine-tuning the model by conducting hyperparameter optimization and cross-validation, all of which were custom-built. The entire journey provided me a robust understanding of the Ridge Regression algorithm, both in theory and in practice.

## Weaknesses

The project was not without its challenges. One specific issue was the initial struggle to optimize the hyperparameters effectively. The Ridge Regression algorithm itself is relatively straightforward, but the nuances of tuning it for optimal performance took some time to grasp. Additionally, time management became a critical factor, as unforeseen personal commitments interfered with the planned project timeline. While the project is complete, the disruptions did impact the depth of certain analyses, particularly in exploring computational efficiencies and additional feature engineering techniques.

## Opportunities

The project has served as a fertile ground for academic and personal growth. It offered a comprehensive view of Ridge Regression, from understanding the mathematical foundations to implementing it in real-world scenarios. The experimentations with different modes of Ridge Regression, such as batch and online, opened avenues for understanding the trade-offs between computational efficiency and model performance. This project also served as an effective bridge between the theoretical knowledge acquired in the classroom and the practical skills required in the industry. I am particularly grateful for the constructive feedback and guidance received from my advisors, which enriched the learning experience. The project has stoked my curiosity to explore other regularization techniques and optimization algorithms from first principles, laying a robust foundation for future work in this area.

## Threats

Potential obstacles in the journey to mastering Ridge Regression could be multi-faceted. Lack of appropriate guidance can lead to inefficient or incorrect implementations. While Ridge Regression is well-documented, the subtleties of its behavior under different conditions can be elusive. Time constraints are another concern; Ridge Regression, while computationally less demanding than some other algorithms, still requires significant time for thorough analysis and tuning. Moreover, the fast-paced evolution of machine learning methodologies poses a risk of obsolescence; what is considered a robust technique today might become obsolete tomorrow.

# References

1. Dhar, V. (2013). Data science and prediction. Communications of the ACM, 56(12), 64-73.
2. Hastie, T., Tibshirani, R., and Friedman, J. (2009). The Elements of Statistical Learning. New York: Springer series in statistics.
3. Hoerl, A. E., and Kennard, R. W. (1970). Ridge regression: Biased estimation for nonorthogonal problems. Technometrics, 12(1), 55-67.
4. Wang, D., Li, Y., and He, F., 2020. Machine Learning for Prediction of Sepsis: A Review. Journal of Healthcare Engineering, pp.1-10.
5. Ivanov, D., 2020. Predicting the impacts of epidemic outbreaks on global supply chains: A simulation-based analysis on the coronavirus outbreak. Transportation Research Part E: Logistics and Transportation Review, 136, p.101922.
6. Kou, G., Yang, P., Xiao, F., Chen, Y., and Alsaadi, F.E., 2020. Machine learning methods for systemic risk analysis in financial sectors. Technological and Economic Development of Economy, 26(2), pp.315-330.
7. Shawe-Taylor, J., & Cristianini, N. (2002). On the generalization of soft margin algorithms. IEEE Transactions on Information Theory, 48(10), 2721-2735.
8. Hahne, J. M., et al. (2014). Linear and Nonlinear Regression Techniques for Simultaneous and Proportional Myoelectric Control. IEEE Transactions on Neural Systems and Rehabilitation Engineering, 22(2), 269-279.
9. Suykens, J. A. K., Lukas, L., & Vandewalle, J. (2000). Sparse approximation using least squares support vector machines. In 2000 IEEE International Symposium on Circuits and Systems (ISCAS), Geneva, Switzerland.
10. Reichstein, M., Camps-Valls, G., Stevens, B., Jung, M., Denzler, J., Carvalhais, N. and Prabhat, 2019. Deep learning and process understanding for data-driven Earth system science. Nature, 566(7743), pp.195-204.
11. Garcia, S., Grill, M., Stiborek, J., and Zunino, A., 2020. Machine Learning in Cybersecurity: A Review. ACM Computing Surveys (CSUR), 63(1), pp.1-26.
12. Mittelstadt, B., Russell, C., and Wachter, S., 2019. Explaining Explanations in AI. Proceedings of the Conference on Fairness, Accountability, and Transparency, pp. 279-288..
13. Jordan, M. I., and Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. Science, 349(6245), 255-260.
14. Stitson, M. O., Gammerman, A., Vapnik, V. N., Vovk, V., Watkins, C., & Weston, J. (1997). Support Vector regression with ANOVA decomposition kernels. Technical report, Royal Holloway, University of London.
15. Girosi, F., Jones, M., & Poggio, T. (1995). Regularization Theory and Neural Networks Architectures. Neural Computation, 7(2), 219-269.
16. Lukman, A. F., Haadi, A., Ayinde, K., Onate, C. A., Gbadamosi, B., & Oladejo, N. (Year unavailable). Some Improved Generalized Ridge Estimators and their comparison. Department of Mathematics, Landmark University, Omu-Aran, Kwara State, Nigeria.
17. Harrison Jr, D., & Rubinfeld, D. L. (1978). Hedonic housing prices and the demand for clean air. Journal of Environmental Economics and Management, 5(1), 81-102.
18. Tibshirani, R. (1996). Regression shrinkage and selection via the lasso. Journal of the Royal Statistical Society: Series B (Methodological), 58(1), 267-288.
19. Liaw, A. and Wiener, M., 2002. Classification and regression by randomForest. R news, 2(3), pp.18-22.
20. Breiman, L., 2001. Random forests. Machine learning, 45(1), pp.5-32.
21. <http://lib.stat.cmu.edu/datasets/boston>
22. Cortes, C. and Vapnik, V., 1995. Support-vector networks. Machine learning, 20(3), pp.273-297.
23. Drucker, H., Burges, C.J., Kaufman, L., Smola, A. and Vapnik, V., 1997. Support vector regression machines. In Advances in neural information processing systems (pp. 155-161).
24. Schölkopf, B., Smola, A.J., Williamson, R.C. and Bartlett, P.L., 2001. New support vector algorithms. Neural computation, 12(5), pp.1207-1245.
25. Smola, A.J. and Schölkopf, B., 2004. A tutorial on support vector regression. Statistics and computing, 14(3), pp.199-222.

# Appendix

# -\*- coding: utf-8 -\*-

## Project Structure

1. Import Essential Libraries

2. Define the Ridge Regression from Scratch

3. Define Kernel Ridge Regression from Scratch

4. Load Dataset-1 and Pre-Process the data

5. Perform Hyper Parameter Tuning of Ridge Regression

6. Perform Hyper Parameter Tuning of Kernel Ridge Regression

7. Comparisions with Sklearn Model

8. Comparisions with Other Models with Custom built Models

9. Load Dataset-2 and Perform all steps from 4 to 9

## Import Essential Libraries

"""

# Importing necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.metrics import mean\_squared\_error

class RidgeRegression:

def \_\_init\_\_(self, learning\_rate: float = 0.01, reg\_strength: float = 0.01, max\_iter: int = 1000):

"""Sets hyperparameters: learning\_rate, reg\_strength, max\_iter."""

self.learning\_rate = learning\_rate

self.reg\_strength = reg\_strength

self.max\_iter = max\_iter

self.weights = None

def fit(self, X: np.ndarray, y: np.ndarray) -> None:

"""Fits model using gradient descent with ridge regularization."""

X = self.\_add\_intercept(X)

self.weights = np.zeros(X.shape[1])

for \_ in range(self.max\_iter):

self.weights -= self.learning\_rate \* self.\_compute\_gradient(X, y)

def predict(self, X: np.ndarray) -> np.ndarray:

"""Returns predicted values for feature matrix X."""

X = self.\_add\_intercept(X)

return X.dot(self.weights)

def \_add\_intercept(self, X: np.ndarray) -> np.ndarray:

"""Adds intercept term to feature matrix X."""

return np.hstack([np.ones((X.shape[0], 1)), X])

def \_compute\_gradient(self, X: np.ndarray, y: np.ndarray) -> np.ndarray:

"""Computes gradient for ridge regression."""

predictions = X.dot(self.weights)

errors = predictions - y

reg\_term = self.reg\_strength \* np.hstack([0, self.weights[1:]])

return (2 / X.shape[0]) \* (X.T.dot(errors) + reg\_term)

"""## Kernel Ridge Regression"""

class KernelRidgeRegression:

# Constructor to initialize kernel type, regularization parameter, gamma, and polynomial degree

def \_\_init\_\_(self, kernel: str = 'linear', C: float = 1.0, gamma: float = 1.0, degree: int = 3):

self.kernel = kernel # Type of kernel used (linear, polynomial, rbf, quadratic)

self.C = C # Regularization parameter

self.gamma = gamma # Parameter for the RBF kernel

self.degree = degree # Degree for the polynomial kernel

self.alphas = None # Coefficients for the regression

# Fit the model to training data using the kernel matrix and regularization

def fit(self, X: np.ndarray, y: np.ndarray) -> None:

K = self.\_compute\_kernel\_matrix(X, X) # Compute kernel matrix for training data

self.alphas = np.linalg.solve(K + self.C \* np.eye(K.shape[0]), y) # Solve for alpha coefficients

# Predict output for test data based on trained alphas and kernel matrix

def predict(self, X\_train: np.ndarray, X\_test: np.ndarray) -> np.ndarray:

K = self.\_compute\_kernel\_matrix(X\_test, X\_train) # Compute kernel matrix for test data

return K.dot(self.alphas) # Multiply kernel matrix by alphas for predictions

# Compute kernel matrix based on selected kernel type

def \_compute\_kernel\_matrix(self, X1: np.ndarray, X2: np.ndarray) -> np.ndarray:

if self.kernel == 'linear':

return X1.dot(X2.T) # Linear kernel

elif self.kernel == 'polynomial':

return (X1.dot(X2.T) + 1) \*\* self.degree # Polynomial kernel with given degree

elif self.kernel == 'rbf':

sq\_dists = np.sum((X1[:, np.newaxis] - X2) \*\* 2, axis=2)

return np.exp(-self.gamma \* sq\_dists) # Radial basis function kernel

elif self.kernel == 'quadratic':

return (X1.dot(X2.T) + 1) \*\* 2 # Quadratic kernel

"""## Dataset 1 : salary\_data.csv"""

# Loading the dataset

salary\_data = pd.read\_csv("salary\_data.csv")

# Extracting features and target

X = salary\_data[["YearsExperience"]].values

y = salary\_data["Salary"].values

# Splitting the data into training and testing sets (80% training, 20% testing)

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Scaling the features using StandardScaler

from sklearn.preprocessing import StandardScaler

feature\_scaler = StandardScaler()

X\_train\_scaled = feature\_scaler.fit\_transform(X\_train)

X\_test\_scaled = feature\_scaler.transform(X\_test)

# Checking the shape of the training and testing data

#X\_train\_scaled.shape, y\_train\_scaled.shape, X\_test\_scaled.shape, y\_test\_scaled.shape

# Hyperparameter combinations for Ridge Regression

learning\_rates = [0.0001, 0.001, 0.01,0.1,1]

reg\_strengths = [0.001,0.01, 0.1, 1]

max\_iters = [100, 500, 1000]

# Table to store the results

ridge\_results = []

# Training and evaluating Ridge Regression with different hyperparameter combinations

for lr in learning\_rates:

for reg in reg\_strengths:

for max\_iter in max\_iters:

model = RidgeRegression(learning\_rate=lr, reg\_strength=reg, max\_iter=max\_iter)

model.fit(X\_train\_scaled, y\_train)

y\_pred = model.predict(X\_test\_scaled)

mse = mean\_squared\_error(y\_test, y\_pred)

ridge\_results.append((lr, reg, max\_iter, mse))

# Creating a DataFrame to display the results

ridge\_results\_df = pd.DataFrame(ridge\_results, columns=['Learning Rate', 'Regularization Strength', 'Max Iterations', 'MSE'])

ridge\_results\_df

# Hyperparameter combinations for Kernel Ridge Regression

kernel\_types = ['linear', 'polynomial', 'rbf', 'quadratic']

Cs = [0.1, 1, 10]

gammas = [0.1, 1, 5]

degrees = [2, 3] # Only relevant for polynomial kernel

# Table to store the results

kernel\_ridge\_results = []

# Training and evaluating Kernel Ridge Regression with different hyperparameter combinations

for kernel in kernel\_types:

for C in Cs:

for gamma in gammas:

for degree in degrees:

model = KernelRidgeRegression(kernel=kernel, C=C, gamma=gamma, degree=degree if kernel == 'polynomial' else 3)

model.fit(X\_train\_scaled, y\_train)

y\_pred = model.predict(X\_train\_scaled, X\_test\_scaled)

mse = mean\_squared\_error(y\_test, y\_pred)

kernel\_ridge\_results.append((kernel, C, gamma, degree if kernel == 'polynomial' else '-', mse))

kernel\_ridge\_results

# Plotting the distribution of the feature (YearsExperience) and target variable (Salary)

fig, axes = plt.subplots(1, 2, figsize=(12, 5))

# Plotting the distribution of YearsExperience

axes[0].scatter(X\_train, y\_train, c='blue')

axes[0].set\_title('Years of Experience vs Salary')

axes[0].set\_xlabel('Years of Experience')

axes[0].set\_ylabel('Salary')

# Plotting the distribution of Salary

axes[1].hist(y\_train, bins=10, edgecolor='black')

axes[1].set\_title('Distribution of Salary')

axes[1].set\_xlabel('Salary')

axes[1].set\_ylabel('Frequency')

plt.tight\_layout()

plt.show()

# Scaling the feature (YearsExperience)

feature\_scaler = StandardScaler()

X\_train\_scaled\_full = feature\_scaler.fit\_transform(X\_train)

X\_test\_scaled\_full = feature\_scaler.transform(X\_test)

# Scaling the target variable (Salary)

target\_scaler = StandardScaler()

y\_train\_scaled\_full = target\_scaler.fit\_transform(y\_train.reshape(-1, 1)).flatten()

y\_test\_scaled\_full = target\_scaler.transform(y\_test.reshape(-1, 1)).flatten()

# Checking the shape of the scaled data

X\_train\_scaled\_full.shape, y\_train\_scaled\_full.shape, X\_test\_scaled\_full.shape, y\_test\_scaled\_full.shape

# Table to store the results for Ridge Regression with fully scaled data

ridge\_results\_scaled = []

learning\_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

reg\_strengths = [0.001, 0.01, 0.1, 1, 10, 100]

max\_iters = [100, 500, 1000, 2000, 5000]

# Training and evaluating Ridge Regression with different hyperparameter combinations

for lr in learning\_rates:

for reg in reg\_strengths:

for max\_iter in max\_iters:

model = RidgeRegression(learning\_rate=lr, reg\_strength=reg, max\_iter=max\_iter)

model.fit(X\_train\_scaled\_full, y\_train\_scaled\_full)

y\_pred = model.predict(X\_test\_scaled\_full)

mse = mean\_squared\_error(y\_test\_scaled\_full, y\_pred)

ridge\_results\_scaled.append((lr, reg, max\_iter, mse))

# Creating a DataFrame to display the results

ridge\_results\_scaled\_df = pd.DataFrame(ridge\_results\_scaled, columns=['Learning Rate', 'Regularization Strength', 'Max Iterations', 'MSE'])

ridge\_results\_scaled\_df

ridge\_results\_scaled\_df.to\_csv('ridge\_results\_scaled\_df.csv')

ridge\_results\_scaled\_df

# Table to store the results for Kernel Ridge Regression with fully scaled data

kernel\_ridge\_results\_scaled = []

# Training and evaluating Kernel Ridge Regression with different hyperparameter combinations

for kernel in kernel\_types:

for C in Cs:

for gamma in gammas:

for degree in degrees:

model = KernelRidgeRegression(kernel=kernel, C=C, gamma=gamma, degree=degree if kernel == 'polynomial' else 3)

model.fit(X\_train\_scaled\_full, y\_train\_scaled\_full)

y\_pred = model.predict(X\_train\_scaled\_full, X\_test\_scaled\_full)

mse = mean\_squared\_error(y\_test\_scaled\_full, y\_pred)

kernel\_ridge\_results\_scaled.append((kernel, C, gamma, degree if kernel == 'polynomial' else '-', mse))

# Creating a DataFrame to display the results

kernel\_ridge\_results\_scaled\_df = pd.DataFrame(kernel\_ridge\_results\_scaled, columns=['Kernel Type', 'C', 'Gamma', 'Degree', 'MSE'])

kernel\_ridge\_results\_scaled\_df

kernel\_ridge\_results\_scaled\_df.to\_csv('kernel\_ridge\_results\_scaled\_df.csv')

# Selecting the best hyperparameters for Ridge Regression

best\_ridge\_params = ridge\_results\_scaled\_df.loc[ridge\_results\_scaled\_df['MSE'].idxmin()][['Learning Rate', 'Regularization Strength', 'Max Iterations']]

# Retraining Ridge Regression with the best hyperparameters

ridge\_model\_best = RidgeRegression(learning\_rate=best\_ridge\_params['Learning Rate'],

reg\_strength=best\_ridge\_params['Regularization Strength'],

max\_iter=int(best\_ridge\_params['Max Iterations']))

ridge\_model\_best.fit(X\_train\_scaled\_full, y\_train\_scaled\_full)

# Making predictions and inverting the scaling

y\_pred\_ridge\_scaled = ridge\_model\_best.predict(X\_test\_scaled\_full)

y\_pred\_ridge\_original = target\_scaler.inverse\_transform(y\_pred\_ridge\_scaled.reshape(-1, 1)).flatten()

# Calculating the MSE on the original scale

mse\_ridge\_original = mean\_squared\_error(y\_test, y\_pred\_ridge\_original)

# Plotting the predicted vs actual values

plt.scatter(X\_test, y\_test, c='blue', label='Actual')

plt.scatter(X\_test, y\_pred\_ridge\_original, c='red', label='Predicted')

plt.title('Ridge Regression: Predicted vs Actual Salaries')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.legend()

plt.show()

mse\_ridge\_original

# Selecting the best hyperparameters for Kernel Ridge Regression

best\_kernel\_ridge\_params = kernel\_ridge\_results\_scaled\_df.loc[kernel\_ridge\_results\_scaled\_df['MSE'].idxmin()][['Kernel Type', 'C', 'Gamma', 'Degree']]

# Retraining Kernel Ridge Regression with the best hyperparameters

kernel\_ridge\_model\_best = KernelRidgeRegression(kernel=best\_kernel\_ridge\_params['Kernel Type'],

C=best\_kernel\_ridge\_params['C'],

gamma=best\_kernel\_ridge\_params['Gamma'],

degree=int(best\_kernel\_ridge\_params['Degree']) if best\_kernel\_ridge\_params['Kernel Type'] == 'polynomial' else 3)

kernel\_ridge\_model\_best.fit(X\_train\_scaled\_full, y\_train\_scaled\_full)

# Making predictions and inverting the scaling

y\_pred\_kernel\_ridge\_scaled = kernel\_ridge\_model\_best.predict(X\_train\_scaled\_full, X\_test\_scaled\_full)

y\_pred\_kernel\_ridge\_original = target\_scaler.inverse\_transform(y\_pred\_kernel\_ridge\_scaled.reshape(-1, 1)).flatten()

# Calculating the MSE on the original scale

mse\_kernel\_ridge\_original = mean\_squared\_error(y\_test, y\_pred\_kernel\_ridge\_original)

# Plotting the predicted vs actual values

plt.scatter(X\_test, y\_test, c='blue', label='Actual')

plt.scatter(X\_test, y\_pred\_kernel\_ridge\_original, c='red', label='Predicted')

plt.title('Kernel Ridge Regression: Predicted vs Actual Salaries')

plt.xlabel('Years of Experience')

plt.ylabel('Salary')

plt.legend()

plt.show()

mse\_kernel\_ridge\_original

# Column names for the Boston Housing dataset

boston\_columns = [

"CRIM", "ZN", "INDUS", "CHAS", "NOX", "RM", "AGE",

"DIS", "RAD", "TAX", "PTRATIO", "B", "LSTAT", "MEDV"

]

# Loading the Boston Housing dataset with proper column names

boston\_data = pd.read\_csv("housing.csv", header=None, names=boston\_columns, delim\_whitespace=True)

# Displaying the first few rows of the dataset

boston\_data.head()

# Extracting features and target variable

X\_boston = boston\_data.drop(columns=["MEDV"]).values

y\_boston = boston\_data["MEDV"].values

# Splitting the data into training and testing sets (80% training, 20% testing)

X\_train\_boston, X\_test\_boston, y\_train\_boston, y\_test\_boston = train\_test\_split(X\_boston, y\_boston, test\_size=0.2, random\_state=42)

# Checking the shape of the training and testing data

X\_train\_boston.shape, X\_test\_boston.shape, y\_train\_boston.shape, y\_test\_boston.shape

# Importing StandardScaler

from sklearn.preprocessing import StandardScaler

# Applying standard scaling to the input features only

scaler\_input\_only = StandardScaler()

X\_train\_boston\_input\_scaled = scaler\_input\_only.fit\_transform(X\_train\_boston)

X\_test\_boston\_input\_scaled = scaler\_input\_only.transform(X\_test\_boston)

learning\_rates = [1e-6, 1e-5, 1e-4, 1e-3, 1e-2, 1e-1]

reg\_strengths = [0.001, 0.01, 0.1, 1, 10, 100]

max\_iters = [100, 500, 1000, 2000, 5000]

# Continuing hyperparameter tuning for Ridge Regression with input normalization only

best\_mse\_ridge\_input\_norm = float('inf')

best\_params\_ridge\_input\_norm = None

for lr in learning\_rates:

for reg in reg\_strengths:

for max\_iter in max\_iters:

mse = ridge\_cross\_validation(X\_train\_boston\_input\_scaled, y\_train\_boston, lr, reg, max\_iter)

if mse < best\_mse\_ridge\_input\_norm:

best\_mse\_ridge\_input\_norm = mse

best\_params\_ridge\_input\_norm = (lr, reg, max\_iter)

best\_params\_ridge\_input\_norm, best\_mse\_ridge\_input\_norm

# Hyperparameter tuning for Kernel Ridge Regression with input normalization only

kernel\_types = ['linear', 'polynomial', 'rbf']

Cs = [0.01, 0.1, 1, 10, 100]

gammas = [0.01, 0.1, 1, 10]

best\_mse\_kernel\_input\_norm = float('inf')

best\_params\_kernel\_input\_norm = None

for kernel in kernel\_types:

for C in Cs:

for gamma in gammas:

mse = kernel\_ridge\_cross\_validation(X\_train\_boston\_input\_scaled, y\_train\_boston, kernel, C, gamma)

if mse < best\_mse\_kernel\_input\_norm:

best\_mse\_kernel\_input\_norm = mse

best\_params\_kernel\_input\_norm = (kernel, C, gamma)

best\_params\_kernel\_input\_norm, best\_mse\_kernel\_input\_norm

# Function to plot the learning curve for Ridge Regression

def plot\_learning\_curve(X, y, learning\_rate, reg\_strength, max\_iter):

kf = KFold(n\_splits=5)

train\_errors = []

val\_errors = []

# Splitting the data into training and validation sets

for train\_index, val\_index in kf.split(X):

X\_train\_fold, X\_val\_fold = X[train\_index], X[val\_index]

y\_train\_fold, y\_val\_fold = y[train\_index], y[val\_index]

model = RidgeRegression(learning\_rate, reg\_strength, max\_iter)

train\_error\_fold = []

val\_error\_fold = []

# Training the model and recording the errors for each iteration

X\_train\_fold\_intercept = model.\_add\_intercept(X\_train\_fold)

model.weights = np.zeros(X\_train\_fold\_intercept.shape[1])

for \_ in range(max\_iter):

model.weights -= learning\_rate \* model.\_compute\_gradient(X\_train\_fold\_intercept, y\_train\_fold)

y\_pred\_train = X\_train\_fold\_intercept.dot(model.weights)

y\_pred\_val = model.predict(X\_val\_fold)

train\_error\_fold.append(mean\_squared\_error(y\_train\_fold, y\_pred\_train))

val\_error\_fold.append(mean\_squared\_error(y\_val\_fold, y\_pred\_val))

train\_errors.append(train\_error\_fold)

val\_errors.append(val\_error\_fold)

# Averaging the errors across the folds

train\_errors\_mean = np.mean(train\_errors, axis=0)

val\_errors\_mean = np.mean(val\_errors, axis=0)

# Plotting the learning curve

plt.plot(train\_errors\_mean, label="Training Error")

plt.plot(val\_errors\_mean, label="Validation Error")

plt.xlabel('Iteration')

plt.ylabel('Mean Squared Error')

plt.title('Learning Curve')

plt.legend()

plt.show()

# Plotting the learning curve for the best Ridge Regression parameters with input normalization only

plot\_learning\_curve(X\_train\_boston\_input\_scaled, y\_train\_boston, \*best\_params\_ridge\_input\_norm[:3])

"""### Comparison with Other Regression Models"""

from sklearn.linear\_model import Ridge, Lasso, ElasticNet, LinearRegression

from sklearn.kernel\_ridge import KernelRidge

from sklearn.svm import SVR

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error

models = [

('Custom Ridge', RidgeRegression(\*best\_params\_ridge\_input\_norm)),

('Custom Kernel Ridge', KernelRidgeRegression(\*best\_params\_kernel\_input\_norm)),

('Sklearn Ridge', Ridge(alpha=best\_params\_ridge\_input\_norm[1])),

('Sklearn Kernel Ridge', KernelRidge(alpha=best\_params\_kernel\_input\_norm[1], kernel=best\_params\_kernel\_input\_norm[0])),

('Lasso', Lasso()),

('ElasticNet', ElasticNet()),

('Linear Regression', LinearRegression()),

('SVR', SVR()),

('Random Forest', RandomForestRegressor())

]

mse\_results = []

model\_names = []

# Train models and compute MSE

for name, model in models:

if name == 'Custom Kernel Ridge':

model.fit(X\_train\_boston\_input\_scaled, y\_train\_boston)

y\_pred = model.predict(X\_train\_boston\_input\_scaled, X\_test\_boston\_input\_scaled)

else:

model.fit(X\_train\_boston\_input\_scaled, y\_train\_boston)

y\_pred = model.predict(X\_test\_boston\_input\_scaled)

mse = mean\_squared\_error(y\_test\_boston, y\_pred)

mse\_results.append(mse)

model\_names.append(name)

plt.bar(model\_names, mse\_results)

plt.xlabel('Model')

plt.ylabel('Mean Squared Error')

plt.title('Comparison of Regression Models')

plt.xticks(rotation=45, ha='right')

plt.tight\_layout()

plt.show()

# Create a dictionary with model names and corresponding MSE

results\_dict = {'Model': model\_names, 'Mean Squared Error': mse\_results}

# Convert the dictionary into a DataFrame

results\_df = pd.DataFrame(results\_dict)

# Sort the results by MSE

results\_df = results\_df.sort\_values(by='Mean Squared Error')

# Display the DataFrame

print(results\_df)

# Importing standard libraries

from sklearn.linear\_model import Ridge, Lasso, ElasticNet, LinearRegression

from sklearn.kernel\_ridge import KernelRidge

from sklearn.svm import SVR

from sklearn.ensemble import RandomForestRegressor

from sklearn.neural\_network import MLPRegressor

from sklearn.metrics import mean\_squared\_error

import matplotlib.pyplot as plt

import pandas as pd

# Placeholder for custom implementations or specific algorithms

class CustomAlgorithm:

def fit(self, X, y):

# Training code

pass

def predict(self, X):

# Prediction code

return predictions

# List of models including custom Ridge and Kernel Ridge models

models = [

('Custom Ridge', RidgeRegression(\*best\_params\_ridge\_input\_norm)),

('Custom Kernel Ridge', KernelRidgeRegression(\*best\_params\_kernel\_input\_norm)),

('Sklearn Ridge', Ridge(alpha=best\_params\_ridge\_input\_norm[1])),

('Sklearn Kernel Ridge', KernelRidge(alpha=best\_params\_ridge\_input\_norm[1], kernel=best\_params\_kernel\_input\_norm[0])),

('Lasso', Lasso()),

('ElasticNet', ElasticNet()),

('Linear Regression', LinearRegression()),

('SVR', SVR()),

('Random Forest', RandomForestRegressor()),

('Neural Network', MLPRegressor()), # Example of a neural network

#('Custom Algorithm', CustomAlgorithm()), # Placeholder for custom or specific algorithm

# Add other models here

]

mse\_results = []

model\_names = []

# Train models and compute MSE

for name, model in models:

if name == 'Custom Kernel Ridge':

model.fit(X\_train\_boston\_input\_scaled, y\_train\_boston)

y\_pred = model.predict(X\_train\_boston\_input\_scaled, X\_test\_boston\_input\_scaled)

else:

model.fit(X\_train\_boston\_input\_scaled, y\_train\_boston)

y\_pred = model.predict(X\_test\_boston\_input\_scaled)

mse = mean\_squared\_error(y\_test\_boston, y\_pred)

mse\_results.append(mse)

model\_names.append(name)

# Plotting the comparison

plt.bar(model\_names, mse\_results)

plt.xlabel('Model')

plt.ylabel('Mean Squared Error')

plt.title('Comparison of Regression Models on Boston Housing Dataset')

plt.xticks(rotation=45, ha='right')

plt.tight\_layout()

plt.show()

# Creating a DataFrame for tabular results

results\_dict = {'Model': model\_names, 'Mean Squared Error': mse\_results}

results\_df = pd.DataFrame(results\_dict)

results\_df = results\_df.sort\_values(by='Mean Squared Error')

print(results\_df)

"""### Ridge Regression for batch and online mode"""

class RidgeRegression:

def \_\_init\_\_(self, learning\_rate=0.01, reg\_strength=0.01, max\_iter=1000):

self.learning\_rate = learning\_rate

self.reg\_strength = reg\_strength

self.max\_iter = max\_iter

self.weights = None

def fit(self, X, y, mode='batch'):

X = self.\_add\_intercept(X)

self.weights = np.zeros(X.shape[1])

if mode == 'batch':

for \_ in range(self.max\_iter):

self.weights -= self.learning\_rate \* self.\_compute\_gradient(X, y)

elif mode == 'online':

for i in range(self.max\_iter):

for xi, yi in zip(X, y):

self.weights -= self.learning\_rate \* self.\_compute\_gradient(xi[np.newaxis, :], yi)

else:

raise ValueError("Invalid mode. Choose 'batch' or 'online'.")

def predict(self, X):

X = self.\_add\_intercept(X)

return X.dot(self.weights)

def \_add\_intercept(self, X):

return np.hstack([np.ones((X.shape[0], 1)), X])

def \_compute\_gradient(self, X, y):

predictions = X.dot(self.weights)

errors = predictions - y

reg\_term = self.reg\_strength \* np.hstack([0, self.weights[1:]])

return (2 / X.shape[0]) \* (X.T.dot(errors) + reg\_term)

# Ridge Regression in Batch Mode

ridge\_model\_batch = RidgeRegression(\*best\_params\_ridge\_input\_norm)

ridge\_model\_batch.fit(X\_train\_boston\_input\_scaled, y\_train\_boston, mode='batch')

mse\_ridge\_batch = mean\_squared\_error(y\_test\_boston, ridge\_model\_batch.predict(X\_test\_boston\_input\_scaled))

# Ridge Regression in Online Mode

ridge\_model\_online = RidgeRegression(\*best\_params\_ridge\_input\_norm)

ridge\_model\_online.fit(X\_train\_boston\_input\_scaled, y\_train\_boston, mode='online')

mse\_ridge\_online = mean\_squared\_error(y\_test\_boston, ridge\_model\_online.predict(X\_test\_boston\_input\_scaled))

print("Ridge Batch MSE:", mse\_ridge\_batch)

print("Ridge Online MSE:", mse\_ridge\_online)