Project 2 - Classification and Regression, from linear and logistic regression to neural networks

Arangan Subramaniam & Krithika Gunesegaran

December 16, 2024

Contents

Abstract

This study investigates the effectiveness of various machine learning algorithms in regression and classification tasks, focusing on optimization methods and neural networks. Building on previous regression analysis work with Ordinary Least Squares (OLS) and Ridge Regression, we incorporate Gradient Descent (GD), Stochastic Gradient Descent (SGD), and adaptive optimization techniques to improve model training efficiency. Additionally, we develop a Feed-Forward Neural Network (FFNN) for comparative analysis, exploring the impact of different activation functions on performance. For regression tasks, Ridge Regression performed best, with the lowest prediction error and the highest R^2 scores, due to its regularization feature that helps prevent overfitting. For classification, Scikit-Learn's Logistic Regression model was the most accurate and reliable, doing better than both the custom Logistic Regression and the FFNN. This study underscores the importance of tuning hyperparameters like learning rate and regularization strength for model stability and generalization, and offers insights into the practical advantages and limitations of each method.

1 Introduction

Machine learning has become a powerful tool for solving diverse real-world problems, ranging from predicting housing prices to diagnosing diseases [1]. Understanding how different models behave, how to effectively train them, and how to select the best approach for a given problem is crucial for leveraging the full potential of machine learning. This study focuses on these core aspects by exploring both regression and classification methods, using traditional models as well as neural networks.

In this study, we build upon our previous study in regression analysis, where we implemented methods like Ordinary Least Squares (OLS) and Ridge Regression to model relationships in data. This time, we extend our focus to include optimization methods, such as Gradient Descent (GD) and Stochastic Gradient Descent (SGD), and evaluate their effectiveness in training models.

The report is organized as follows: First, we cover the theoretical background, including linear models, optimization techniques, and the basics of neural networks. Then, we describe how we developed and implemented these methods, focusing on building the Feed-Forward Neural Network and setting up key training parameters like learning rates, batch sizes, and epochs. After that, we present the results for both regression and classification tasks, comparing the different approaches. Finally, we summarize our findings, discussing the strengths and weaknesses of each model and point out which methods are best suited for handling complex data.

This study builds on our previous work while providing a practical approach to understanding the complexities of machine learning model development. It allows us to explore the effects of optimization strategies and activation functions. This gives us practical insights crucial for tackling both regression and classification challenges.

2 Theory

In this section, we provide an overview of the theoretical concepts applied throughout this study.

2.1 Linear Regression

Linear Regression is a statistical model used to predict the value of a dependent variable based on the values of one or more independent variables [5]. It is one of the simplest and most widely used methods for fitting continuous functions to data. The linear regression model can be expressed as:

$$\boldsymbol{y} = \boldsymbol{X}\beta, \tag{1}$$

where $\boldsymbol{y}$ is the vector of observed values, $\boldsymbol{X}$ is the design matrix containing the input data, and β represents the parameters to be estimated.

To find the best fitting parameters β, we minimize a cost function $C(\beta)$ that measures the differences between the predicted and actual values. In this project, we will focus on two regression methods, Ordinary Least Squares (OLS) and Ridge Regression. The cost function for these regression methods are given in Equation 2 and 4 with their corresponding β when we minimize the cost function in Equation 3 and 5.

$$C_{\text{OLS}}(\beta) = \frac{2}{n}\|y - X\beta\|_2^2, \tag{2}$$

where β is given as,

$$\beta_{\text{OLS}} = (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{3}$$

Ridge Regression is like OLS but adds a regularization term to penalize large coefficients to improve the stability of the data.

$$C_{\text{Ridge}}(\beta) = \frac{1}{n}\|y - X\beta\|_2^2 + \lambda\|\beta\|_2^2, \tag{4}$$

where β is

$$\beta_{\text{Ridge}} = (\boldsymbol{X}^T\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^T\boldsymbol{y}. \tag{5}$$

λ is the regularization parameter in Ridge Regression that controls the strength of the penalty on β.

2.2 Logistic Regression

Logistic Regression is a statistical method primarily used for binary classification, where it predicts the probability that an input belongs to one of two classes, like 0 or 1 [14]. Unlike linear regression which predicts continuous outcomes, logistic regression predicts a probability between 0 and 1, showing how likely it is that the given data point falls into a particular class.

The logistic function, also called as sigmoid function, is central to logistic regression.

This function is defined as:

$$p(t) = \frac{1}{1 + e^{-t}} \tag{6}$$

where t is a linear combination of the input features.

$$t = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n \tag{7}$$

Here, β_0 is the intercept, and β_i are the weights for each feature x_i. This transformation guarantees that the output is always between 0 and 1, making it useful for predicting probabilities.

In binary classification, the probability that a data point x_i belongs to class 1 can be modeled by using Equation 8.

$$p(y_i = 1 | x_i, \beta) = \frac{e^{\beta_0 + \beta_1 x_i}}{1 + e^{\beta_0 + \beta_1 x_i}} \tag{8}$$

Where β_0 is the intercept, and β_1 is the feature weigths. The (β) parameters helps us determine the best fit data, by using methods like Gradient Descent (which will be explained further in this report) or other optimization techniques that minimizes the cost function.

The cost function used in logistic regression, given in Equation 9, measures the error between the predicted probability and the actual class label for each data point. Typically, logistic regression uses the cross-entropy loss, which is minimized to achieve the best model fit.

$$C(\beta) = -\sum_{i=1}^{n} \left(y_i(\beta_0 + \beta_1 x_i) - \log\left(1 + \exp(\beta_0 + \beta_1 x_i)\right) \right). \tag{9}$$

Logistic regression serves as an effective soft classifier by outputting a probability for each class, rather than a strict binary output [12]. This makes it particularly useful when dealing with data that includes some level of uncertainty. Logistic regression can also be extended to multi-class classification with techniques like "one-vs-all", making it a good choice for many classification tasks because it is easy to use and understand. During this project, we will only be focusing on single-layer classification.

2.3 Gradient Descent

Gradient descent (GD) is a widely used optimization algorithm for training machine learning models and neural networks. It works by minimizing the errors between the models predictions and the actual outcomes [4].

The idea behind Gradient Descent is to minimize a function $F(x)$, $x = (x_1, \ldots, x_n)$ by iterating the parameters in the direction that decreases the functions value the fastest, which is the negative gradient $-\nabla F(\mathbf{x})$. This way, the algorithm moves step-by-step towards the lowest point, of the function. This is mathematically expressed in Equation

10.

$$\mathbf{x}_{k+1} = \mathbf{x}_k - \eta_k \nabla F(\mathbf{x}_k) \tag{10}$$

where x_k is the current parameter value, η_k is the learning rate, also called as step size and $\nabla F(x_k)$ is the gradient of the function at point x_k. For every iteration the algorithm moves towards the minimum of the function, reducing the functions value at each step.

The learning rate η_k is crucial in the gradient descent process. η_k controls how big step you take in the direction of the negative gradient. If the learning rate η_k is too small, the algorithm converges very slowly while if it is too large, the method may fail to converge or behave erratically.

For each iteration, the algorithm updates the parameters to move closer to a minimum, effectively decreasing the functions value at each step.

While Gradient Descent (GD) is powerful, it has several limitations. Ideally, GD aims to reach the global minimum, but in non-convex functions, it can get stuck in local minima. Its performance is also highly sensitive to the initial starting point and learning rate. If η is too small, it leads to slow convergence, while a large one may cause divergence. Additionally, computing gradients for large datasets is costly, and the method struggles with saddle points, slowing down the optimization process.

2.3.1 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) is a variant of Gradient Descent that updates the models parameters more frequently by using just one data point, or a small batch at a time, instead of the entire dataset [6]. In SGD, the models parameters are updated after calculating the gradient for each individual point, rather than waiting for the entire dataset. This means the parameters are updated right away after each training example, instead of waiting until the entire dataset has been processed. The main idea of SGD is that the cost function we want to minimize can be expressed as the sum of smaller parts, each part corresponding to a data point as in Equation 11.

$$C(\beta) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i, \beta) \tag{11}$$

This also means the gradient can be computed as a sum of the gradients for each data point as in Equation 12.

$$\nabla_\beta C(\beta) = \sum_{i=1}^{n} \nabla_\beta c_i(\boldsymbol{x}_i, \beta) \tag{12}$$

This lets SGD calculate and update the gradient more often by using each data point individually, rather than waiting for the whole dataset.

Since SGD updates the model parameters frequently, it allows for quicker and more refined adjustments. However, this frequent updating also have noise, which can cause the parameter updates to fluctuate. While these fluctuations might slow down the convergence, they are also beneficial by adding randomness that prevents the model from getting stuck in local minima, and can increase the chances of finding a global minimum.

2.3.2 Batches and mini-batches

Batches computes the total error across the entire training set before updating the model's parameters [4]. While this approach is stable and efficient for smaller datasets, it becomes slow and memory-intensive for large datasets since all data must be processed and stored at once. It produces smooth convergence but can sometimes get stuck in local minima.

Mini-Batches offers a middle ground between batch and SGD by splitting the dataset into smaller batches [4]. After each mini-batch is processed, the model updates its parameters. This approach combines the stability of batches with the frequent updates of SGD. This approach is more efficient and makes it well suited for large datasets. Typically, each batch contains a few hundred to a few thousand data points, allowing the model to learn efficiently without processing the entire dataset at once.

2.3.3 Momentum parameter

Momentum is a technique used in Gradient Descent to speed up convergence by using the information from previous iterations. Instead of just using the current gradient, momentum also uses parts of the previous update to smooth out changes and help the optimization move faster in steady directions. This speed, the momentum update, can be describes as:

$$v_t = \rho v_{t-1} + \eta \nabla_\theta E(\theta_t), \tag{13}$$

where v_t is the momentum term, ρ is the momentum parameter, ranging between 0 and 1. The momentum term controls how much of the past velocity is kept. η is the learning rate and $\nabla_\theta E(\theta_t)$ is the gradient of the error function with respect to the current parameters.

Once the velocity v_t is calculated, the parameter update is done as follows:

$$\theta_{t+1} = \theta_t - v_t \tag{14}$$

Momentum helps the algorithm continue moving in the same direction, making it more efficient and faster in finding the minimum, especially in places where the cost function is relatively flat. The momentum parameter ρ and learning rate η can be defined as:

$$\rho = \frac{m}{m + \mu \Delta t}, \tag{15}$$

and

$$\eta = \frac{(\Delta t)^2}{m + \mu \Delta t}, \tag{16}$$

where m is the mass, μ is the viscous drag and Δt is the time step.

Here, the parameter m works like the mass of a particle, giving it inertia. This means it helps the optimization process keep moving in the same direction, even when conditions change, resulting in smoother and more efficient movement through flat regions of the cost function. Similarly, the viscous drag μ works like a resistance and slows down the movement.

Overall, adding momentum makes the optimization process smoother and more efficient, even in areas that are noisy or have steep gradients, leading to faster convergence and better results.

2.3.4 AdaGrad

Adaptive Gradient Algorithm (AdaGrad) is an optimization technique used in machine learning to adjust the learning rate for each parameter during training, making it particularly effective [13]. AdaGrad automatically adapts the learning rate for each parameter based on past updates, leading to more efficient convergence [13]. AdaGrad works by scaling the learning rate for each parameter based on the sum of its past squared gradients.

2.4 Adam Optimizer & RMSprop

Second-order moment methods like Adam and RMSprop adaptively adjust the learning rate based on both the gradient and the curvature of the error surface. This allows them to better navigate complex error landscapes. Adam combines the running averages of both gradients and their squares to adapt the learning rate dynamically. This makes it particularly effective for handling noisy data and sparse gradients.

RMSprop focuses on adjusting the learning rate by keeping a moving average of the squared gradients [6]. This helps stabilize and speed up convergence, especially in scenarios with uneven error landscapes.

These adaptive techniques make both optimizers well-suited for efficiently training deep learning models.

2.5 Neural networks

Neural Networks are computational models inspired by biological neural networks in the human brain. They learn from examples without being explicitly programmed with task-specific rules. This is usually done by using training and test data. Neural networks are built from layers of connected neurons. The concept is that each neuron takes in multiple inputs, combines them using weights, applies a function to decide its output, and then sends this output to the next layer. The strength of the connections between neurons is represented by weights. By adjusting these weights, the network learns to recognize patterns and improve its performance on a specific task. This can be mathematically expressed as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(z), \tag{17}$$

Where y is the output, w_i are the weights, x_i are the input signals, and $f(z)$ is the activation function.

2.5.1 Feed-Forward Neural Network (FFNN)

This is the simplest type of Neural Network, where the input only flows in one direction from input to output, without looping back [7]. Each layers nodes are connected to all nodes in the next layer, which makes the network fully connected.

The model have three types of layers: the input layer, hidden layers and output layer. Each neuron takes its inputs, calculates a weighted sum, applies an activation function, and then sends the result to the next layer, just like in a neural network. This can be mathematically expressed as:

$$z = \sum_{i=1}^{n} w_i x_i, \tag{18}$$

Where z is the weighted sum, w_i are the weights, a_i are the inputs to this neuron. The neurons output y is calculated by applying an activation function f to the input z.

$$y = f(z) \tag{19}$$

In a FFNN, using this function creates a perceptron. It is when it takes in one or more inputs, applies weights to them, and then gives a result of either 1 or 0. It works well for binary classification tasks, where it can be used in the output layer to separate two classes.

In a neural network, each layer receives input signals either from the original input data or from the outputs of neurons in the previous layer. To make the model more flexible, a bias term is added to each neurons weighted sum. This allows the network to have a threshold that can shift, making the model more capable of fitting a wider range of functions.

The input to the activation function for neuron j in a layer can therefore be expressed as in Equation 20.

$$z_j = \sum_{i=1}^{n} w_{ij} a_i + b_j, \tag{20}$$

where z_j is the pre-activation output of neuron j, w_{ij} is the weight associated with the connection between input i and neuron j.a_i is the input to the neuron, coming from the previous layer or the input layer and b_j is the bias term for neuron j.

The bias b_j allows the neuron to produce an output even if the sum of weighted inputs is zero. This flexibility is crucial as it helps the neuron shift the activation threshold, making it easier for the neuron to activate under different circumstances, depending on the problem being solved.

Thus, the final output of a neuron is calculated by applying an activation function f to z_j, allowing the network to introduce non-linearity and better capture complex patterns in the data. This is a core mechanism that helps the neural network approximate complex functions and relationships in the data.

2.5.2 Activation Functions

Activation functions play a crucial role in enabling neural networks to learn complex patterns by introducing non-linearity [7]. Without non-linear activation functions, neural networks would be limited to performing simple linear transformations, regardless of the number of layers, and unable to model complex data effectively. We will be focusing on three different activation functions during this project. The sigmoid funtion, Rectified Linear Unit (ReLU) and Leaky ReLU.

The sigmoid, also called as logistic function is commonly used for binary classification and maps input values to a range between 0 and 1. This is given as Equation 21.

$$f(x) = \sigma(x) = \frac{1}{1 + e^{-x}} \tag{21}$$

The sigmoid is often used in the output layer of binary classification models to represent probabilities.

The Rectified Linear Unit (ReLU) function is the most widely used activation function in deep learning due to its simplicity and efficiency. It outputs the input directly if it is positive, otherwise it returns zero. This is given as Equation 22

$$R(z) = \begin{cases} 0 & \text{if } z \leq 0 \\ z & \text{if } z > 0 \end{cases} \tag{22}$$

However, ReLU suffers from the dying ReLU problem, where neurons can stop learning entirely by outputting zero for all inputs. This occurs when the weights are adjusted so that the inputs turn negative, effectively "killing" the neuron.

To reduce the this issue, Leaky ReLU allows a small, non-zero output for negative input values, keeping neurons active even when they receive negative inputs. This is given as:

$$f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ \alpha x, & \text{if } x < 0 \end{cases} \tag{23}$$

Where α is typically a small value like 0.01.

Choosing the right activation function is crucial for neural network performance. The sigmoid function is effective for binary classification, but it can suffer from vanishing gradients, making it less suitable for deep networks. ReLU, on the other hand, is widely used in hidden layers due to its simplicity and computational efficiency, which generally leads to faster convergence. Leaky ReLU builds on this by addressing the dying ReLU problem, helping to maintain active neurons during training and making it a more robust alternative in some scenarios.

3 Implementation

3.1 Stochastic Gradient Descent

We generated sample data using $n = 100$ observations of x, with x values randomly sampled from a uniform distribution. To keep the model straightforward, we defined the target variable y using a simple quadratic function:

$$y = 1 + 2x + 3x^2 + \epsilon \tag{24}$$

where ϵ represents noise sampled from a standard normal distribution. We then implemented and analyzed gradient descent techniques, each offering different approaches to optimize model convergence. These methods included standard gradient descent, modification with momentum, mini-batch Stochastic Gradient Descent, and adaptive learning rate optimizers. Below is a summary of each technique and its role in enhancing model training efficiency.

3.1.1 Plain Gradient Descent

The `PlainGradientDescent` function minimizes Mean Squared Error (MSE) by optimizing model weights (β) through gradient descent, with optional momentum and L2 regularization. In each iteration, it calculates the gradient of the MSE with respect to β, adding a ridge penalty term if ridge parameter (λ) is provided. If a nonzero momentum parameter (ρ) is set, the function applies momentum by incorporating a fraction of the previous update to smooth and speed up convergence, helping the algorithm move past areas where progress is slow. The testing included Ordinary Least Squares (OLS) regression with and without momentum, and Ridge Regression with different values of λ and learning rates (η). Results were visualized with line plots showing MSE of iterations, comparing momentum and non-momentum versions. These results will be presented in the next section. Additional heatmaps displayed MSE in Ridge regression as function of the ridge parameter λ and η, highlighting the effects of these parameters on model accuracy and stability.

3.1.2 Stochastic Gradient Descent

The `StochasticGradient`function performs Stochastic Gradient Descent (SGD) with mini-batches, optional momentum, and L2 regularization for optimizing linear regression models. Each epoch, shuffles the data and divides it into mini-batches of size M, computing the gradient of the Mean Squared Error (MSE) for each batch. With Ridges regularization term λ, a penalty is added, and if momentum ρ is specified, the function applies a fraction of the previous velocity to smooth and accelerate convergence. This function was tested on Ordinary Least Squares (OLS) and Ridge regression, both with and without momentum, and varying regularization parameters. Results included MSE line plots and heatmaps displaying the effects of batch size, learning rate (η), and ridge penalty (λ) on convergence and model performance.

3.1.3 Adagrad

The `Adagrad_plain` function applies the Adagrad optimization method, where gradient updates are adjusted based on the squared gradients, making the learning rates apdaptive for each parameter. Optional momentum (η) is included to smooth updated and accelerate convergence, while ridge regularization term (λ) can be added to penalize large weights, helping to control overfitting. At each iteration, MSE is recorded to track model performance over each epoch. MSE was recorded across epochs for each iteration to compare how adaptive learning rates, momentum, and regularization impacted model performance.

The `Adagrad_StochasticGradient` function combines Adagrad with mini-batch stochastic gradient descent, shuffling the data at the each epoch and dividing it into mini-batches for efficient processing. The adaptive learning rate scales mini-batch update, and optional momentum and L2 regularization further enhance model stability and generalization. Testing the function for ρ ¿ 0, $\rho = 0$, and regularization demonstrates how these adjustments impact stability, and overall model accuracy, giving insight into which settings best optimize training.

3.1.4 RMSprop and Adam Optimizers

The `RMSprop` function implements the RMSprop optimization algorithm, maintaining a running average squared gradients (`G`) to adaptively adjust the learning rate. A decay factor (`rho`) helps stabilize convergence, especially in the presence of noisy gradients. The `Adam` function applies the Adam optimizer, combining momentum (first momentum estimate,`m`) with adaptive learning rates (second moment estimates, `v`), and includes bias correction to ensure stable and efficient updates. Both functions have been added to the library for potential future use but have not been tested or applied to any models.

3.2 Neural Network

The neural network, defined in `FeedForwardNN.py`, is designed for flexible application in both regression and binary classification tasks. Key parameters include network architecture, learning rate η, momentum parameter ρ, and regularization term λ, with weights initialized using He initialization to promote stable gradient flow and biases set to zero for simplicity [3].Even though the He intialization is more favorable for ReLU, we chose sigmoid as the final activation function, affect the outcomes of the results. The reason for He Intialization is to prevent gradient explosion [3]. A learning schedule gradually reduce the learning rate, preventing training instability [9]. Forward and backward propagation methods calculate activations and gradients across layers, essential for efficient weight updates and optimization. The model is trained via `fit()` function, which employs Stochastic Gradient Descent algorithms such employing mini-batch with data shuffling to improve convergence and applies momentum to accelerate learning. Cost functions used are Binary Cross-Entropy(BCE)/log loss for classification and Mean Squared Error for regression, with optional regularization λ to control overfitting. Evaluation metrics include accuracy for classification and MSE or R^2 for regression, providing insight into model performance on new data.

In `FFNN_evaluation.ipynb`, we assessed the performance of the model using synthetic polynomial regression data to test accuracy on non-linear pattern. Standardizing the input data to zero mean and unit variance ensures consistent and stable training. The FFNN model is trained and compared with Linear Regression, Ridge Regression, and `MLPregressor` from Scikit-learn [10], with MSE calculated for performance comparison. Further analysis on test data uses MSE and R^2 scores to evaluate FFNN performance relative to other models.

3.3 Activation Functions

When testing the Activation Functions, we implemented the custom Feed-Forward Neural Network (FFNN), trained using three activation functions- `Sigmoid, ReLU, Leaky ReLU`-to analyze their effects on model performance. Each model configuration was trained over 50 epochs with η=0.02, and $\rho = 0.1$, using a batch size of 10 for each activation functions. Following training, the code plots the Mean Squared Error (MSE) over epochs for each activation function. This visualization enables comparative analysis of convergence rates and training efficiency across activation functions, providing insights into their distinct impacts on model performance and error minimization.

3.4 Classification

This implementation trains a Feed-Forward Neural Network (FFNN) for binary classification on **the breast cancer dataset** from Scikit-learn [10], using `.load_breast_cancer` method, which includes to differentiate malignant from bening tumors [15]. Binary Cross-Entropy, also known as Log loss, is used as the cost function, ideal for classification as it penalizes prediction errors in probability-based outputs [8]. The data is split into 80/20 training and testing sets, then standardized with `StandardScaler` to achieve zero mean and unit variance [10], promoting stable and faster convergence. The FFNN model, with a single hidden layer of 10 nodes, is trained using Sigmoid activation, Momentum and Ridge regularization. Training and validation metrics, including accuracy and loss, are recorded across epochs and visualized with smoothed plots for clarity. Accuracy is calculated using the `accuracy_score` method from `sklearn.metrics` [10]. The performance of the model is further analyzed with different λ and η through heatmaps. Finally, the FFNN results are compared with `MLPClassifier` from Scikit-Learn [10], to highlight differences in training loss and accuracy under various parameter settings.

3.5 Logistic Regression

The `LogisticRegression` function uses Stochastic Gradient Descent (SGD) with mini-batches, momentum (ρ), and L2 regularization (λ) to optimize the weights (`Beta`). The `Sigmoid` function transforms continuous values into a range of (0,1), ensuring that input the next layer remains within a fixed range, which stabilizes weight updates and enhances model convergence [11]. The `binary_cross_entropy_loss`, using the `log_loss` method imported from `sklearn.metrics`[10], calculates the difference between predicted probabilities and true labels. This loss function is ideal for logistic regression, as it heavily

penalizes misclassifications, encouraging the model to minimize classification errors [2]. Additionally, a learning schedule (`LearningSchedule`) adjusts the learning rate, improving the stability as the model converges. A grid search is conducted over λ and initial learning rate η values, systematically evaluating their effects on model loss which is visualized in final heatmaps.

4 Results

4.1 Stochastic Gradient Descent

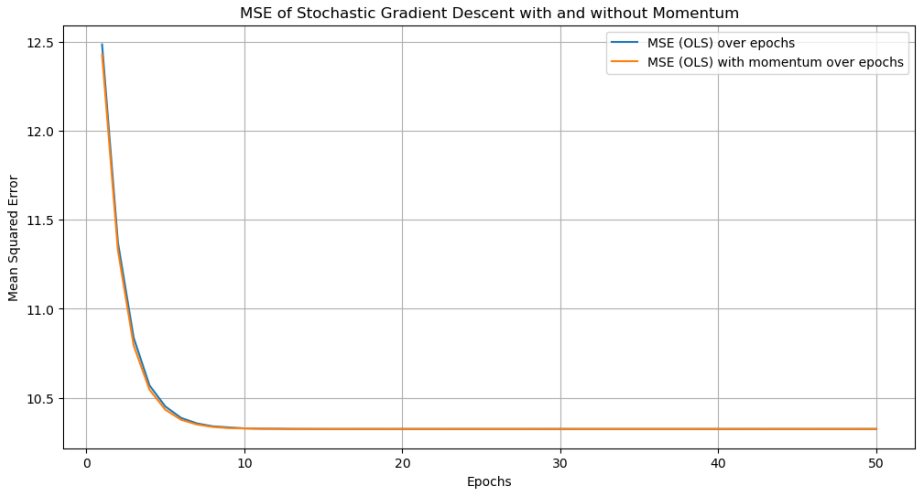

Figure 1: MSE of Stochastic Gradient Descent with and without Momentum

Figure 1 illustrates the Mean Squared Error (MSE) over epochs for Stochastic Gradient Descent (SGD) with and without momentum, highlighting momentum's impact on optimization. Both methods initially reduce error rapidly, but the model with momentum converges slightly faster and stabilizes at a marginally lower MSE. Momentum smooths out fluctuations caused by noisy updates in SGD by incorporating information from previous updates, allowing the optimization process to progress more stably and efficiently. By the end of 50 epochs, both approaches reach similar MSE levels.

The heatmap, in Figure 2, shows the MSE for Ordinary Least Squares (OLS) across different batch sizes and epochs, illustrating how batch size impacts the training dynamics. Smaller batch sizes introduce more noise, which helps avoid local minima but causes fluctuations in MSE, while larger batches provide smoother, more stable convergence but may get stuck in suboptimal solutions. Mini-batch sizes achieve a balance, combining frequent updates with stability, leading to efficient learning across epochs. Overall, as epochs increase, MSE mostly decreases across batch sizes, reflecting improved model performance.

Figure 3 illustrates MSE over epochs for Ridge Regression with Stochastic Gradient Descent (SGD) using various regularization term (λ) and the option of momentum. Lower λ values like 1×10^{-3} lead to faster convergence and lower MSE, while higher values like 3.16×10^{1} and 1×10^{3} cause slower convergence and stabilize at a higher MSE due to

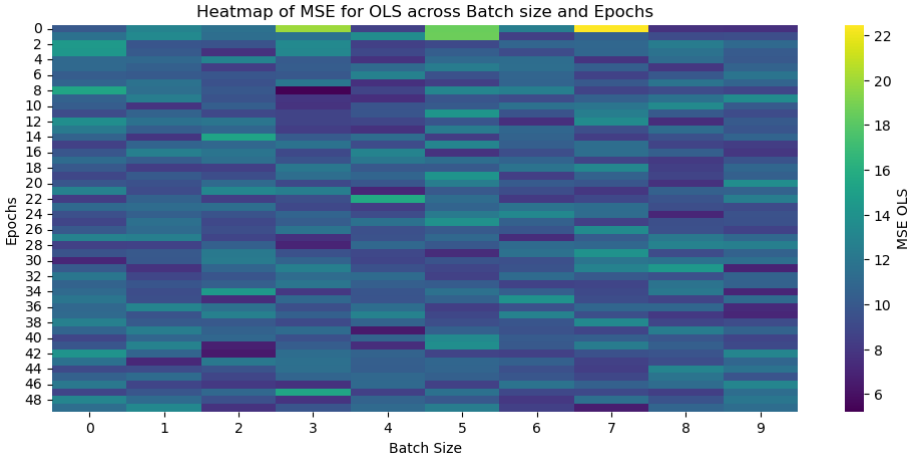

Figure 2: Heatmap of MSE for OLS across Batch size and Epochs

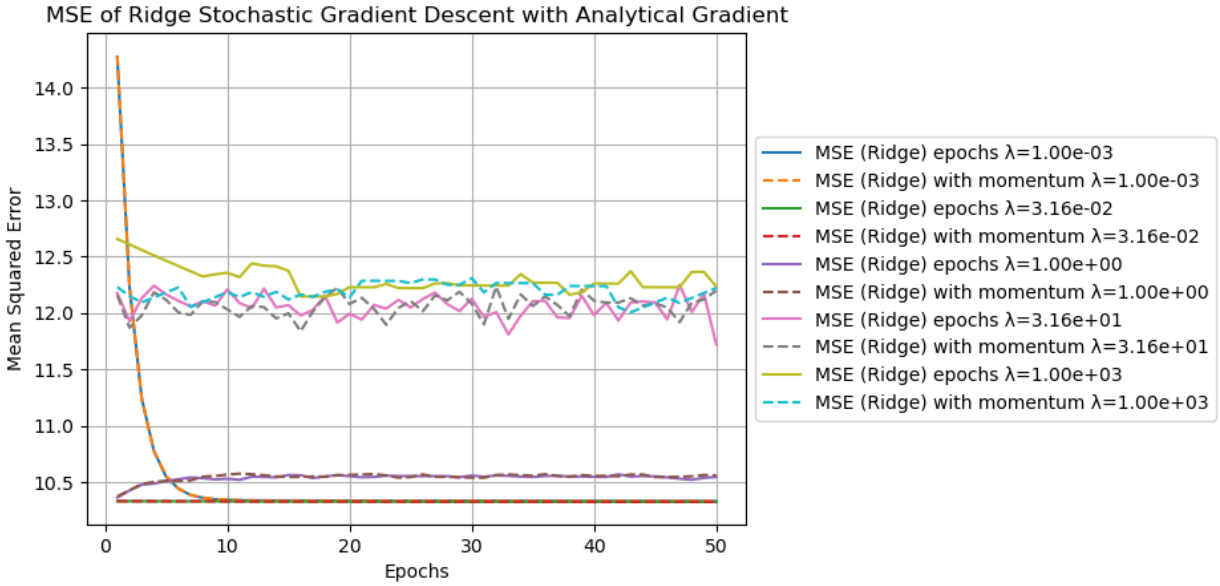

Figure 3: MSE of Ridge Stochastic Gradient Descent with Analytical Gradient

stronger regularization. Intermediate values like 3.16×10^{-2} and 1×10^{0} balance error reduction with stability. Momentum smooths the convergence path, reducing fluctuations and providing more stability, particularly for higher λ values. Overall, this demonstrates that balancing λ and using momentum can optimize both the stability and speed of the learning process.

Figure 4 compares the Mean Squared Error (MSE) for Ridge Regression across different combinations of regularization terms (λ) and learning rates (η), both with and without the use of momentum. In the MSE without momentum heatmap, the lowest MSE is achieved with a learning rate of 9.26×10^{-3} and low values of λ, which shows that without momentum, a moderate learning rate and low regularization yield the best results. In contrast, the MSE with momentum heatmap reveals that the lowest MSE occurs when the learning rate is 8.47×10^{-3} and λ is between 3.16×10^{-2} and 3.16×10. However, overall, the MSE values in the "without momentum" scenario are generally

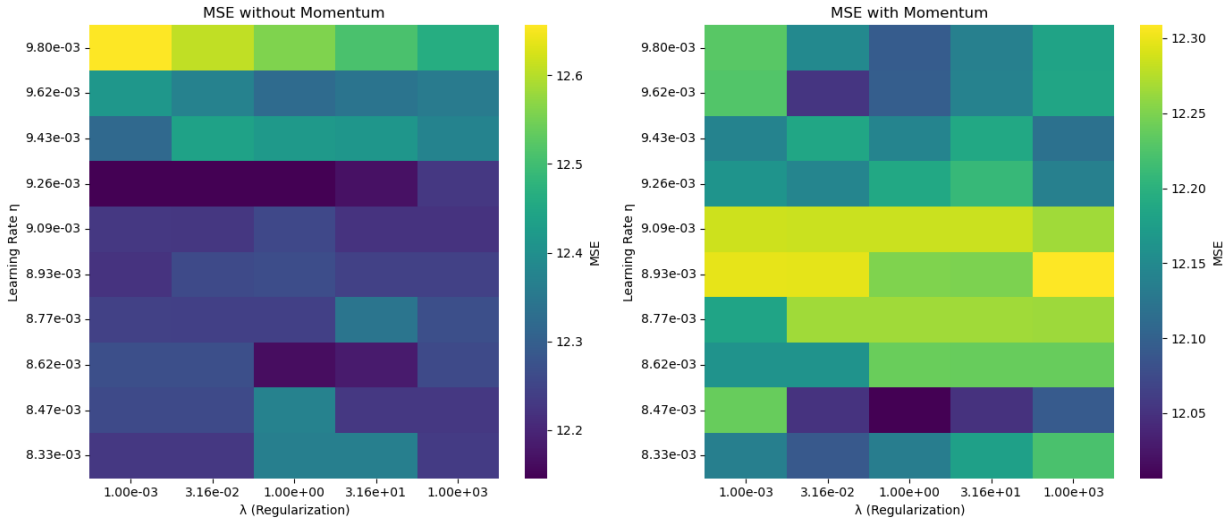

Figure 4: Heatmaps of MSE with and without momentum across λ and η

lower, indicating that in this case, the addition of momentum did not improve model performance and led to higher MSE values across most parameter combinations. This suggests that, for this specific task, using Ridge Regression without momentum provided more favorable results in terms of minimizing prediction error.

4.2 Neural Network

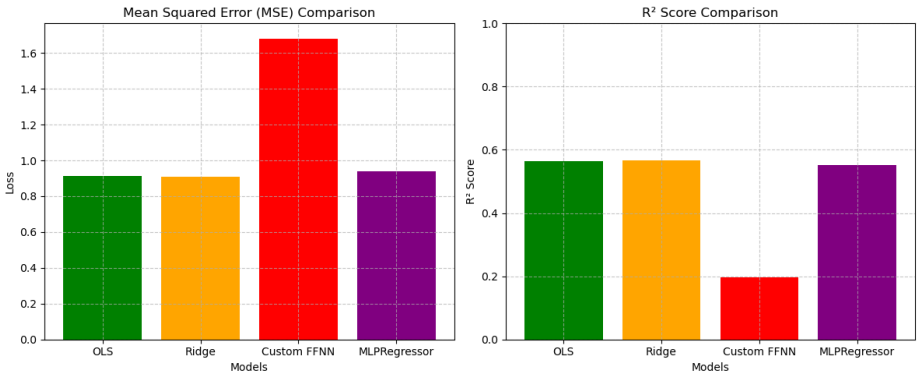

Figure 5: Model Prediction on Training Data

```
Custom FFNN MSE (Scaled): 1.6807, R²: 0.1978
OLS MSE (Scaled): 0.9116, R²: 0.5649
Ridge MSE (Scaled): 0.9088, R²: 0.5662
Scikit-Learn MLPRegressor MSE (Scaled): 0.9386, R²: 0.5520
```

Figure 6: Model Performance Comparison: Mean Squared Error (MSE) and R^2 Scores for Various Regression Models on Scaled Data

Figure 5 compares the performance of four models for the MSE and R^2 score: Ordinary Least Squares (OLS), Ridge Regression, a custom Feed-Forward Neural Network (FFNN), and Scikit-Learn's MLPRegressore. OLS and Ridge Regression both achieve

almost identical MSE and R^2 scores as shown in Figure 6, showing a stable and solid performance in capturing the data trends. Interestingly, although the MLPRegressor is generally well-suited for capturing non-linear relationships, its MSE is slightly higher than OLS and Ridge, indicating that, in this particular case, it did not outperform the simpler models. The R^2 score of the MLPRegressor, however, is almost the same as OLS and Ridge, reflecting similar capabilities in explaining data variance. Ridge Regression typically benefits from its regularization term, which adds a penalty to large coefficients, generally making it more effective at preventing overfitting. The custom FFNN model displayed the highest MSE and lowest R^2 score, suggesting challenges in fitting the data, likely due to suboptimal parameter tuning or insufficient model complexity. Overall, while the MLPRegressor should be capable of handling non-linear patterns effectively, OLS and Ridge performed slightly better in this scenario, with Ridge having an edge over OLS due to its regularization feature.

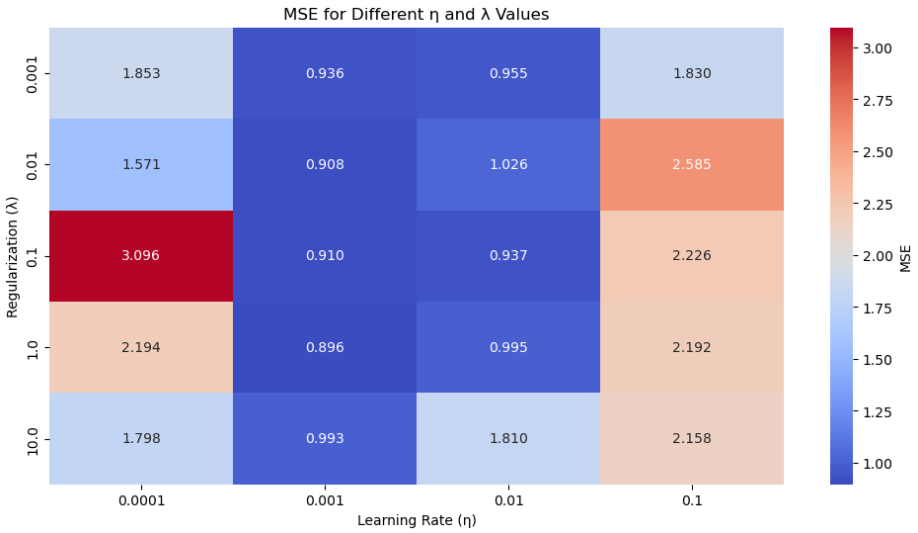

Figure 7: MSE for Different η and λ

The heatmap in Figure 7 shows the MSE for different combinations of learning rates (η) and regularization term (λ), illustrating the conditions that yield the best and worst performances. The results indicate that the best MSE values are achieved when learning rate is either 0.001 or 0.01. Specifically, with learning rate of 0.001, the λ values ranging from 0.001 to 10.0 provide consistently low MSE, indicating strong model performance. Similarly, with a learning rate of 0.01, λ values between 0.001 and 1.0 also produce low MSE values, showing a favorable balance of learning rate and regularization. Conversely, the worst MSE is observed when $\lambda = 0.1$ and the learning rate is 0.0001, which is somewhat surprising given the generally stabilizing influence of a low learning rate. These findings highlight that while careful tuning of both η and λ is crucial, certain combinations like those with moderate regularization and appropriate learning rates, lead to more favorable results while others can unexpectedly degrade performance.

The heatmap in Figure 8 shows R^2 scores for various combinations of learning rates (η) and regularization terms (λ), revealing both effective and poor parameter settings. The best R^2 values are achieved when $\eta = 0.001$, regardless of λ, indicating stable and consistent model performance across different λ values. Similarly, when $\eta = 0.01$, the R^2 scores are also good for λ values ranging from 0.001 to 1.0. In contrast, the worst

Figure 8: R^2 score across different λ and η values

performance occurs when $\eta = 0.0001$ and $\lambda = 0.1$, as well as when $\eta = 0.1$ and $\lambda = 0.01$. These settings result in negative R^2 scores, indicating that the model is either underfitting or struggling to converge properly. This highlights the importance of careful hyperparameter tuning. Choosing the wrong learning rates or regularization values can make the model perform poorly. Proper tuning helps find the right balance between learning effectively, being stable, and making good predictions on new data.

4.3 Activation Function

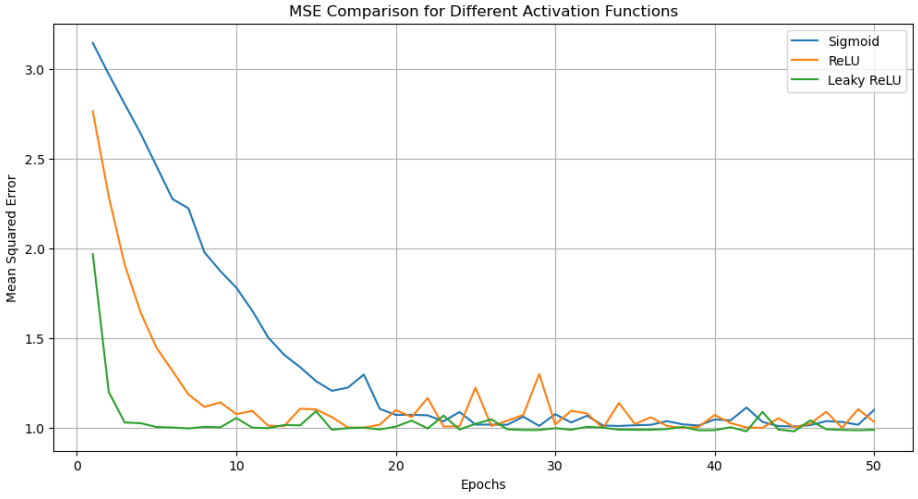

Figure 9: MSE Comparison for Different Activation Functions

The plot in Figure 9 shows the MSE over 50 epochs for Sigmoid, ReLU, and Leaky ReLU activation functions. Each activation function starts at a different MSE level. Sigmoid starts the highest, while Leaky ReLU starts the lowest. As training goes on, Sigmoids MSE decreases steadily but slowly, which could be due to the vanishing gradient problem. ReLU drops the MSE quickly at first but has several spikes along the way,

showing instability, possibly because some neurons stopped updating. There are times when Sigmoids MSE even becomes lower than ReLUs. However, by the end, Sigmoid ends up with the highest MSE, while Leaky ReLU ends with the lowest. This means that Leaky ReLU not only starts better but also keeps reducing the error smoothly, making it the most effective choice overall.

4.4 Classification

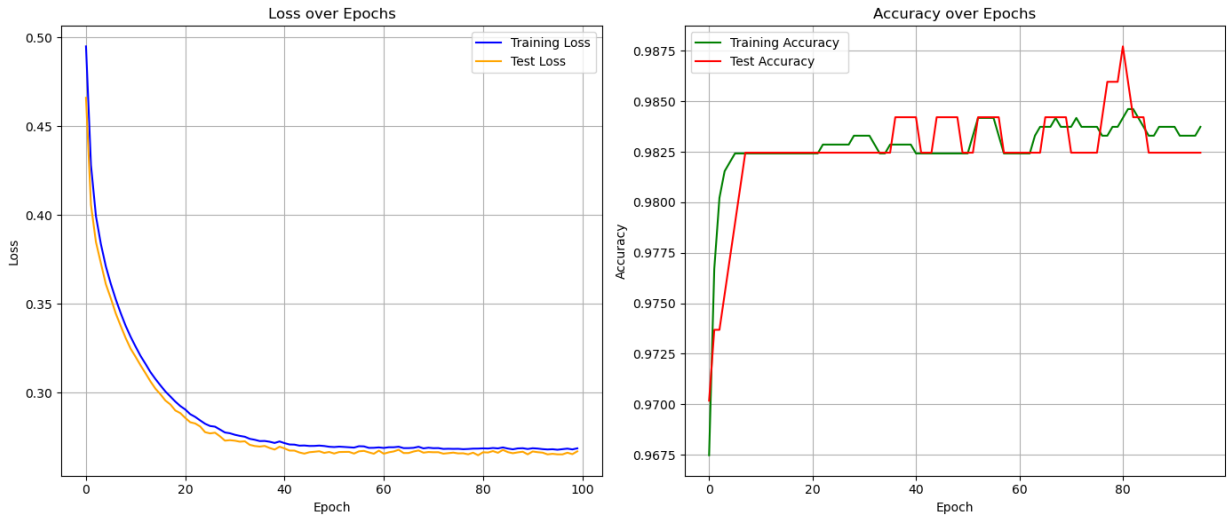

Figure 10: Loss and Accuracy over Epochs for Training and Test Data

Figure 10 presents two plots: The first shows the loss over epochs, and the second displays accuracy over epochs for both training and test datasets. In the loss plot, the training (blue line) and test (orange line) are almost identical throughout the epochs, indicating good consistency and generalization between the training and test data. However, in the accuracy plot, while the training accuracy (green line) remains relatively stable around 98%, the test accuracy (red line) exhibits some peaks and fluctuations. These peaks indicate sensitivity to the test data, suggesting that while the model is learning effectively, it may occasionally overreact to the variations in the test set. This is why the accuracy plot is less stable compared to the smooth decline seen in the loss plot. These fluctuations could be mitigated by using additional regularization or further hyperparameter tuning to enhance stability, especially in test scenarios.

Figure 11 illustrates three plots: Training loss, Training accuracy, and Test accuracy for different combinations of learning rates (η) and regularization values (λ). These plots show how these hyperparameters impact the models performance, helping us understand which combinations yield the best results for both training and testing data. From the training loss heatmap, we see that $\eta = 0.0063$ and $\lambda = 0.1$ provide a low training loss, which corresponds to a high training accuracy of 0.96 and a decent test accuracy of 0.60. Another good combination is when $\eta = 0.0063$ and $\lambda = 1.0$, where training accuracy reaches 0.97, showing effective learning. However, our goal is to achieve the best possible test accuracy, and this occurs with $\eta = 0.1$ and $\lambda = 10$, giving a value of 0.97 for test accuracy. Although the training loss for this combination is 0.30, which is acceptable,

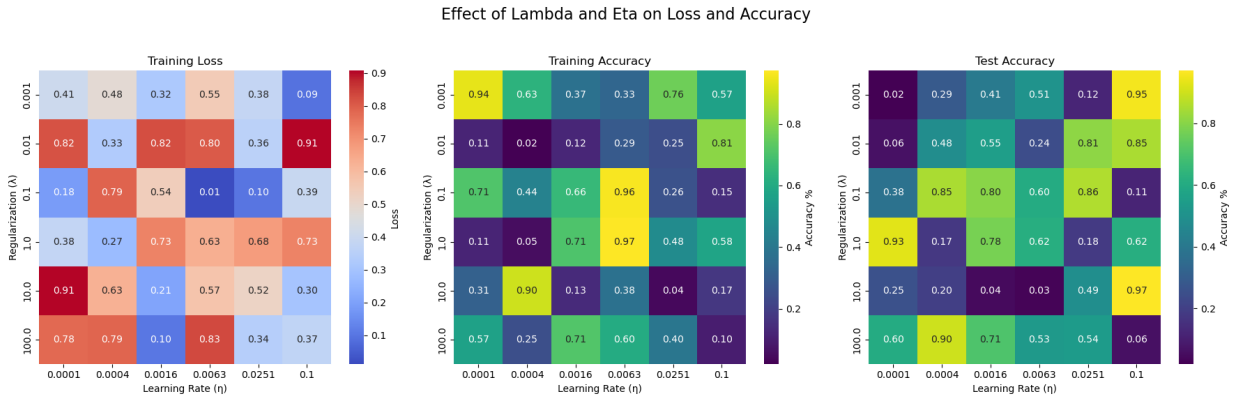

Figure 11: Effect of Lamda and Learning rate on Loss and Accuracy. (This figure has also been included individually in our GitHub repository to provide a clearer view of the values on the heatmap)

the training accuracy is only 0.17, which is quite poor, indicating underfitting during training. Additionally, when $\eta = 0.0016$ or $\eta = 0.0063$ with $\lambda = 0.1$ or 1.0, these combinations also give decent results, with very high training accuracy and acceptable training loss, though the test accuracy remains moderate. These results highlight the importance of balancing hyperparameters to ensure good training performance while also achieving strong generalization on test data.

4.5 Logistic Regression

Figure 12: Test Accuracy for Different learning rates and lamda values for own Logistic Regression and Scikit-Learn Logistic Regression.

Figures 11 and 12 illustrate the effect of different learning rates (η) and L2 regularization values (λ) on test accuracy for three models: the FFNN (Figure 11), and the Custom and Scikit-Learn Logistic Regression models (Figure 12). For the FFNN, the best test accuracy (0.97) is achieved with $\eta = 0.1$ and $\lambda = 10$, though training accuracy is low, indicating underfitting. Other combinations, such as $\eta = 0.0063$ with $\lambda = 0.1$ or 1.0, yield high training accuracy and moderate test accuracy, underscoring the need for balanced hyperparameter tuning for optimal results.

For the Custom Logistic Regression model, results are best when the learning rate is high, generally performing well for all regularization values (λ) when the learning rate (η) ranges from 0.0006 to 0.1. This indicates that performance is more dependent on the learning rate than on λ. In contrast, for Scikit-Learn's logistic regression, performance is more dependent on the regularization value, achieving the best results when $\lambda = 10$ across all learning rates. Overall, the Custom Logistic Regression demonstrates a strong reliance on learning rate adjustments, while Scikit-Learn's logistic regression exhibits stability, particularly with moderate to high regularization values.

5 Discussion

In this study, we explored various algorithms, each demonstrating specific strengths and weaknesses that made them suitable for different parts of the project.

5.1 Pros and Cons of the algorithms

Linear models, such as OLS and Ridge Regression, offered simplicity, capturing general trends effectively in linear data. Ridge, with its regularization, proved particularly useful in controlling noisy data, although too much regularization risked underfitting in some scenarios. FFNN proved effective at capturing non-linear relationships, especially when using activation functions like Leaky ReLU, which mitigated issues like vanishing gradients and dying neurons.

SGD and its variations, such as mini-batch and momentum, brought a significant advantage in terms of optimization speed. However, they introduced instability during learning, especially in scenarios with noisy gradient updates. For example, Figure 1 illustrates how, although momentum led to faster convergence, its benefits over plain SGD were not as substantial. FFNN is generally effective for non-linear relationships, also showed certain limitations. Figure 9 reveals that activation functions like ReLU experienced spikes in MSE, likely due to the "dying ReLU" problem, where neurons stop updating entirely. Even the better performing Leaky ReLU showed sensitivity to learning rates and required careful hyperparameter tuning, as seen in Figures 11 and 12, where imbalanced choices of learning rates and λ values led to poor model performance and underfitting in several cases. These results underscore the delicate trade-off between parameter tuning and model stability, particularly for more complex and adaptive neural architectures.

5.2 Regression

The best-performing algorithm for regression in this study was Ridge Regression. As shown in Figure 5, Ridge achieved the lowest MSE and highest R^2 scores, unexpectedly outperforming the MLPRegressor, which is typically more suited to non-linear relationships. This advantage is likely due to Ridge's regularization term (λ), which helps stabilize the model by penalizing large coefficients, enhancing its ability to generalize.

The tuning of hyperparameters, specifically learning rate (η) and regularization (λ),

proved crucial in optimizing Ridge's performance. From Figures 7 and 8, the lowest MSE (0.896) and the highest R^2 score (0.572) were achieved with $\lambda = 1.0$ and $\eta = 0.001$. These values surpass those seen in other models, confirming that, in this specific context, Ridge offered the best balance between minimizing error and explaining data variance, even though MLPRegressor might typically be expected to perform better in capturing non-linear patterns.

5.3 Classification

Based on the results, the best algorithm for classification was the Scikit-Learn Logistic Regression model. As shown in Figure 12, this model achieved consistently high test accuracy across a range of learning rates and regularization values (λ), demonstrating both stability and adaptability. Specifically, the Scikit-Learn Logistic Regression model performed best with $\lambda = 10$, achieving the highest accuracy, which indicates that its performance was more dependent on regularization than on the learning rate.

In comparison, the custom Logistic Regression model was more sensitive to learning rate adjustments and performed well with higher learning rates across multiple values of λ. However, it did not consistently achieve the same high accuracy as the Scikit-Learn implementation. Additionally, the Feed-Forward Neural Network (FFNN) achieved a high test accuracy of 0.97 with $\eta = 0.1$ and $\lambda = 10$, but this setting led to low training accuracy, indicating possible underfitting. Overall, Scikit-Learn's Logistic Regression model demonstrated the most reliable performance for classification, with robust generalization across the test data.

6 Conclusion

This study demonstrated that Ridge Regression performed best for regression tasks, benefiting from its regularization term to achieve the lowest MSE and highest R^2 scores, even surpassing the MLPRegressor. Key parameters, $\lambda = 1.0$ and $\eta = 0.001$, were essential in minimizing error and enhancing data variance explanation, underscoring the importance of tuning for optimal results. In classification, Scikit-Learn's Logistic Regression showed consistently high accuracy, especially at $\lambda = 10$, proving to be both stable and adaptable compared to the more sensitive custom Logistic Regression model and the Feed-Forward Neural Network (FFNN).

Each algorithm had specific strengths and limitations. Linear models like OLS offered simplicity and good generalization for basic trends but struggled with more complex, non-linear patterns. Techniques like mini-batch and momentum in SGD helped speed up optimization, though they ultimately produced nearly the same results. Future work could focus on deeper neural networks, robust optimization methods, and automated hyperparameter tuning using gridsearchCV, to enhance stability and performance, especially for tasks with complex, non-linear data relationships.

7 Acknowledgment

During this project, we gathered essential information for both the code and theoretical concepts from Professor Morten Hjorth-Jensen's GitHub page and the extra materials and videos he provided.

The ChatGPT tool has been valuable a lot during this project. It has been used to:

- Provided guidance on adapting and modifying code when encountering issues, especially with generating plots, and for the feed-forward neural network.
- Review and refined the grammar, enhancing the overall clarity and language of the project.

References

[1] Amine Charot. *TensorFlow Neural Network Regression: Predicting Real Estate Prices*. Accessed: 2024-11-04. 2024. URL: https://charotamine.medium.com/tensorflow-neural-network-regression-predicting-real-estate-prices-25a02976736d.

[2] DataScience-ProF. *Understanding Log Loss: A Comprehensive Guide with Code Examples*. https://medium.com/@TheDataScience-ProF/understanding-log-loss-a-comprehensive-guide-with-code-examples-c79cf5411426. Accessed: 28/10/2024. 2024.

[3] GeekforGeeks. *Kaiming Initialization in Deep Learning*. https://www.geeksforgeeks.org/kaiming-initialization-in-deep-learning/. Accessed: 29/10/2024. 2023.

[4] IBM. *Gradient Descent*. https://www.ibm.com/topics/gradient-descent. Accessed: 2024-11-04.

[5] IBM. *What is Linear Regression?* URL: https://www.ibm.com/topics/linear-regression.

[6] Morten-Hjorth Jensen. *Week 39: Optimization and Gradient Methods*. https://github.com/CompPhysics/Mach
Accessed: 2024-11-04. 2024.

[7] Morten-Hjroth Jensen. *Data Analysis and Machine Learning: Neural networks, from the simple perceptron to deep learning*. https://github.com/CompPhysics/MachineLearning/blob/master/doc/
Accessed: 2024-11-04. 2019.

[8] Igal Leikin. *Understanding Binary Cross-Entropy and Log Loss for Effective Model Monitoring*. https://www.aporia.com/learn/understanding-binary-cross-entropy-and-log-loss-for-effective-model-monitoring/. Accessed: 2024-11-04. 2024.

[9] AI Maverick. *Schedule learning rate in deep-learning*. https://samanemami.medium.com/schedule-learning-rate-in-deep-learning-bf4875f4f90f. Accessed: 28/10/2024. 2023.

[10] Fabian Pedregosa et al. "Scikit-learn: Machine learning in Python". In: *Journal of machine learning research* 12.Oct (2011).

[11] Asifur Rahman. *Logistic Regression is nothing but Linear Regression with a Non-Linear Activation Function*. https://medium.com/@asifurrahmanaust/lesson-4-logistic-regression-is-nothing-but-linear-regression-with-non-linear-activation-function-464b91a7a077. Accessed: 28/10/2024. 2023.

[12] Week 38: Logistic Regression and Optimization. *Morten-Hjorth Jensen*. https://github.com/CompPhysics/Ma Accessed: 2024-11-04. 2024.

[13] Brijesh Soni. *Understanding the Adagrad Optimization Algorithm: An Adaptive Learning Rate Approach*. https://medium.com/@brijesh_soni/understanding-the-adagrad-optimization-algorithm-an-adaptive-learning-rate-approach-9dfaae2077bb. Accessed: 2024-11-04. 2023.

[14] Wikipedia contributors. *Logistic regression*. https://en.wikipedia.org/wiki/Logistic$_r egression$. Accessed: 2024-11-04. 2024.

[15] Wolberg, William, Mangasarian, Olvi, Street, Nick, and Street, W. *Breast Cancer Wisconsin (Diagnostic)*. UCI Machine Learning Repository. 1993.

8 Appendix

https://github.uio.no/arangans/Project2_0.git