Learning From Differences

Oliver Linger

April 10, 2024

Final-Year Project - B.Sc. in Computer Science

Supervisor: Dr. Derek Bridge

Department of Computer Science

University College Cork

Abstract

This study presents an innovative exploration into the realm of machine learning, specifically focusing on enhancing model interpretability and performance through a novel ensemble of model-based and instance-based learning approaches. Grounded in the principles of Case-Based Reasoning (CBR) and the Learning From Differences methodology, we introduce the `LingerRegressor` and `LingerClassifier` models. These models aim to leverage the strengths of both learning approaches to address the limitations present in traditional machine learning algorithms, particularly in the contexts of regression and classification tasks.

Our methodology involves a detailed examination of the algorithms' design and implementation, highlighting the integration of novel variations such as context inclusion, duplication based on distance, and the addition of a distance column to further refine the learning process. Through extensive experiments, we demonstrate the models' effectiveness in achieving higher interpretability and robust performance across diverse datasets. The `LingerRegressor` and `LingerClassifier` models not only offer new avenues for academic research but also practical implications for developing more transparent and efficient machine learning systems.

The findings underscore the potential of blending model-based and instance-based learning to enhance machine learning models' adaptability and interpretability. Future work will focus on addressing the identified limitations and exploring the applicability of our approach to broader machine learning challenges. This study contributes to the ongoing dialogue on improving machine learning methodologies and paves the way for further innovation in the field.

Declaration of Originality

In signing this declaration, you are confirming, in writing that the submitted work is entirely your own original work, except where it clearly attributed otherwise, and that it has not been submitted partly or wholly for any other educational award.

1. This body of work is all my own, unless clearly stated otherwise, with the full and proper accreditation;
2. With respect to my work: none of it has been submitted to any educational institution contributing in any way to an educational award.
3. With respect to anyone else's work: all diagrams, text, code, or ideas have been duly attributed to the source in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or print.

Signed

Date: April 10, 2024

Acknowledgements

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning encompasses various paradigms and methodologies aimed at enabling computational systems to learn from data and make predictions or decisions without explicit programming. Among the diverse approaches in machine learning, model-based and instance-based learning represents two fundamental strategies, each with distinct advantages and disadvantages.

This dissertation presents a comprehensive study on blending model-based and instance-based learning methodologies to address their individual limitations and harness their strengths, leading to the development of robust, interpretable, and efficient machine learning models suitable for a wide range of applications. Below, we outline the roadmap of this dissertation and summarize our key contributions.

1.1 Report Roadmap

- **Chapter 1** introduces the motivations behind exploring blended learning methodologies, highlighting the potential benefits of integrating model-based and instance-based learning approaches.
- **Chapter 2** delves into the theoretical underpinnings of model-based and instance-based learning, providing a detailed review of the literature and identifying gaps that our research aims to fill.
- **Chapter 3** describes the methodology adopted in this study, including the development, and implementation of new models based ont the new learning approaches designed to address specific challenges. That integrates seamlessly with scikit-learn's code base.

- **Chapter 4** presents the results of extensive experiments conducted to evaluate the performance of the proposed models against traditional approaches, showcasing their advantages or disadvantages in various scenarios.
- **Chapter 5, 5**, synthesizes the findings, discusses the implications of this research, and outlines potential directions for future exploration.

1.2 Contributions of This Dissertation

This dissertation makes the following significant contributions to the field of machine learning:

1. The development of a novel blended learning framework that combines the strengths of model-based and instance-based learning approaches to improve prediction accuracy and model interpretability, with an adaptable and extendable code base, based on sckit-learn.
2. An extensive evaluation of the proposed models using diverse datasets, demonstrating their performance in terms of accuracy, robustness, and efficiency compared to traditional learning models.
3. The introduction of an innovative approach to handling categorical features in blended learning models, enhancing their applicability to a wider range of problems.
4. A detailed exploration into the application of the learning from differences methodology into image based classification and regression problems.
5. The proposition of several areas for future work, including the exploration of other categorical similarity measures, variations in rule generation, scalability improvements, and a diversification of datasets to further enhance the effectiveness and efficiency of blended difference learning models.

1.3 Model-Based Learning

Model-based learning involves the construction of explicit representations of the underlying relationships within the data. These representations, often referred to as models, capture the patterns and structures present in the dataset and can be used to

make predictions or infer insights from new, unseen data points. For instance, artificial neural networks (ANNs) exemplify model-based learning by discerning trends and patterns in data.

1.3.1 Advantages of Model-Based Learning:

1. **Generalization:** Well-constructed models have the ability to generalize patterns learned from the training data to unseen instances, thereby making accurate predictions on new data samples. This generalization capability is essential for robust performance across diverse datasets and real-world scenarios. For example, a well-trained regression model can accurately predict housing prices in different regions based on historical data. The book [4] provides insights into the generalization capabilities of model based learning.

2. **Robustness:** Models learn underlying patterns and assumptions within data, making them more robust to noisy outliers. By capturing trends within data, models can effectively mitigate noise, enhancing their reliability in real-world scenarios. The book [4] also shows the ability of models to learn underlying patterns within data, making them far more robust to noisy data.

3. **Efficiency:** Once trained, model-based systems can quickly return predictions, making them suitable for real-time applications where rapid decision-making is essential. The journal paper [8] offers an insight how the efficiency of model based learning can be implemented to make predictions on big data.

1.3.2 Disadvantages of Model-Based Learning:

1. **Sensitivity to Assumptions:** Models often rely on simplifying assumptions about data distribution and relationships. Deviations from these assumptions can significantly degrade performance, leading to inaccurate predictions or biased outcomes. For example, a model trained to predict house prices based on 2017 data may inaccurately predict house prices for 2023. The book "Machine Learning: A Bayesian and Optimization Perspective" by Sergios Theodoridis and Konstantinos Koutroumbas emphasizes the crucial role assumptions play in model sensitivity, stressing their significance in machine learning applications [16].

2. **Interpretability and Decision-Making:** Certain models, like Neural Networks, lack transparency in decision-making despite their high accuracy. In critical domains like healthcare and finance, interpretability is crucial for validating

decisions and ensuring trust in the system. This disadvantage of lack of transparency is depicted well in the journal paper [14].

3. **Limited Flexibility:** Traditional model-based algorithms struggle to capture complex, non-linear relationships in high-dimensional or unstructured data, limiting their performance in tasks with intricate patterns or diverse feature interactions.

1.4 Instance-Based Learning

Instance-based learning, also known as memory-based learning or lazy learning, eschews explicit model construction in favor of storing and manipulating instances or examples from the training data. Instead of deriving general rules or representations, instance-based methods make predictions based on the similarity between new instances and those observed in the training set. A classic example of instance-based learning is the k-nearest neighbors (KNN) algorithm. The advantages and disadvantages discussed with instance based learning are seen in the paper [2].

1.4.1 Advantages of Instance-Based Learning:

1. **Flexibility:** Instance-based methods adapt to the characteristics of the training data, making them suitable for tasks with complex, non-parametric relationships or evolving patterns. They can capture nuances and irregularities without imposing rigid assumptions.

2. **Explainability:** The transparency of instance-based learning enables analysts to trace predictions back to specific examples in the training set, facilitating interpretability and trust in the system. The journal paper [14] highlights this issue for black box model based learning such as neural nets.

1.4.2 Disadvantages of Instance-Based Learning:

The disadvantages discussed with instance based learning are discussed in the paper [2].

1. **Computational Complexity:** Storing and manipulating the entire training dataset can lead to high memory consumption and computational overhead during prediction, limiting scalability.

2. **Susceptibility to Noise and Redundancy:** Instance-based approaches are sensitive to noisy or irrelevant features, which can lead to suboptimal performance or overfitting. Redundant instances or outliers may distort similarity measures, compromising model robustness.

1.4.3 Motivation for Blended Learning

The advantages and disadvantages of both model-based and instance-based learning highlight the need for a versatile and robust approach to machine learning. While model-based learning excels in generalization and robustness, it lacks transparency and struggles with complex relationships. Instance-based learning offers flexibility and explainability but suffers from computational complexity and susceptibility to noise.

The motivation for blended learning arises from leveraging the strengths of both approaches while mitigating their weaknesses. Blended methods aim to capitalize on the advantages of model-based and instance-based learning, circumventing their limitations. By integrating the interpretability of instance-based learning with the accuracy of model-based systems, blended learning offers a promising solution for critical domains like healthcare and finance. The journal paper [14] provides further evidence that it is important to move towards interpretability for our models going forward, so that critical systems are interpretable and repairable by humans going forward.

In recent years, neural networks have gained significant attention in regression tasks [12]. Inspired by case-based reasoning principles [13], researchers are exploring predictive ensemble models based on these principles [17]. Deviating from conventional approaches, some researchers advocate for training neural networks on differences between sets of features to enhance efficiency and transparency [7].

By adopting a blended approach, machine learning systems can achieve greater interpretability, reliability, and performance across diverse applications.

Chapter 2

Literature Review

2.1 Introduction

The exploration of alternative methodologies for enhancing the performance of neural networks in regression tasks has garnered significant attention within the machine learning community in recent years. One innovative approach, inspired by the principles of case-based reasoning (CBR), involves training neural networks to predict differences between problems based on disparities between features rather than the features themselves. This departure from the conventional supervised learning paradigm aims to leverage the inherent advantages of learning from differences, potentially leading to more interpretable and efficient models.

The study under consideration [7] conducts a systematic factorial study to investigate the efficacy of this approach across various datasets and experimental conditions. The findings suggest that neural networks trained on differences demonstrate comparable or even superior performance to those trained using traditional methods, while requiring significantly fewer epochs for convergence. In this section of the literature review, we consider the utilization of case-based reasoning for regression tasks, with a specific focus on the implementation of KNN (K-Nearest Neighbors) and ANN (Artificial Neural Network) frameworks.

As we delve further into the literature review, we extend our exploration of case-based reasoning beyond regression tasks to include classification challenges. The study by Ye et al. [18], titled "Learning Adaptations for Case-based Classification," offers insights into the application of CBR in classification tasks, shedding light on recent advancements and methodologies. Through a detailed analysis of various adaptation

strategies and neural network-based approaches, we aim to uncover the effectiveness of CBR in solving classification problems, thereby providing a comprehensive understanding of its utility across different machine learning tasks.

2.2 KNN + ANN for Regression, Utilizing CBR

The exploration of alternative methodologies for enhancing the performance of neural networks in regression tasks has attracted considerable attention within the machine learning community in recent years. One innovative approach, inspired by the principles of case-based reasoning (CBR), involves training neural networks to predict differences between problems based on disparities between features rather than the features themselves. This departure from the conventional supervised learning paradigm aims to leverage the inherent advantages of learning from differences, potentially leading to more interpretable and efficient models. The study under consideration [7] conducts a systematic factorial study to investigate the efficacy of this approach across various datasets and experimental conditions. The findings suggest that neural networks trained on differences demonstrate comparable or even superior performance to those trained using traditional methods, while requiring significantly fewer epochs for convergence. In this section of the literature review, we consider the utilization of case-based reasoning for regression tasks, with a specific focus on the implementation of KNN (K-Nearest Neighbors) and ANN (Artificial Neural Network) frameworks.

2.2.1 Learning From Differences Paper

The paper by [7] introduces the innovative concept of training a neural network based on the disparities between a case and its closest neighbors. Titled "A Factorial Study of Neural Network Learning from Differences for Regression" [7], the study aims to present novel learning methodologies geared towards enhancing performance and fostering a more interpretable learning process. The research evaluates three model variations: a benchmark neural network, a learning from differences model, and an augmented learning from differences model incorporating the original context. The context here refers to the base case, from which differences with its nearest neighbors are calculated.

According to [7], the study's key findings underscore a significant increase in accuracy when incorporating contextual information in the learning process. Moreover, comparable or superior results were achieved with fewer training epochs compared to the basic neural network. This enhancement is attributed partly to the broader training

dataset that encompasses neighbors greater than 1. Notably, the paper advocates for a hybrid approach, combining learning from differences with direct feature-based learning, which yields superior results.

The related work for the paper [7] has addressed the use of neural networks in the case-based reasoning process. These papers have focused on exploiting the base case using the idea that adaptation of information is acquirable from the differences between pairs [9]. Since then, various different approaches have been used. More recently, in relation to using neural networks to predict the differences between problems, a few interesting points have been investigated. These points show the ability of a neural network to correct the solution of the most similar retrieved case. These preliminary works expressed in the paper showed that the use of differences plus context yielded superior results [6]. The previous studies examined in the paper [7] did not examine the impact of different parameters during the CBR (case-based reasoning) experiments. The paper [7] made an effort to explore the effect of epochs in the learning process.

The graphical representations in the paper highlight a distinct trend: rapid attainment of high accuracy levels with the learning from differences method, particularly when contextual information is included. Conversely, the basic neural network requires a substantially larger number of epochs to achieve comparable accuracy, if not worse. This disparity suggests the efficiency of the learning from differences approach.

The paper suggests a marginal performance enhancement and a substantial reduction in training epochs through the adoption of the differences method and inclusion of contextual information. However, it is crucial to acknowledge the time overhead associated with retrieving and training using similar cases.

2.2.2 Algorithms

The training algorithm for learning from differences is described in Algorithm 1, while the prediction algorithm is outlined in Algorithm 2. Additionally, a variant of the training algorithm with context included is presented in Algorithm 3, and the corresponding prediction algorithm is provided in Algorithm 4.

Context is utilized in 3, and 4. Context is the case used to retrieve (Nearest Neighbors). In the training algorithm it is X_i^{train}, and in prediction it is X_j^{test}. These are included in the training, and prediction in Variant 2.

The paper [7] has an associated library where they have implemented a regression-based machine learning model.

Algorithm 1 Training Algorithm for Learning from Differences

Input: Dataset D, neural network model, number of neighbors n, number of epochs E

Split D into training set D^{train} and test set D^{test} with a 80%-20% split
Initialize ΔD^{train} as an empty list
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta D_{ij}^{train} = (X_i^{train} - X_j^{train}, y_i^{train} - y_j^{train})$
    end for
end for
Train the MLPRegressor neural network model using ΔD^{train} over a few epochs for later predictions.
Return Trained neural network model $TrainedNN()$

Algorithm 2 Prediction Algorithm for Learning from Differences

Input: Test dataset D^{test}, trained neural network model $TrainedNN()$, number of neighbors n

for each case $X_j^{test} \in D^{test}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors
    Initialize empty list $y^{pred}predictions$
    for each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta X_{ij}^{test} = X_j^{test} - X_i^{train}$
        Use the trained neural network to predict: $TrainedNN(\Delta X_{ij}^{test}) = \Delta X_{ij}^{pred}$
        Adapt neighbor's y values: $y^{pred}predictions_j = X_i^{train} + \Delta X_{ij}^{pred}$
    end for
    Average the adapted y values: $y_j^{pred} = \frac{1}{n} \sum_{i=1}^{n} y^{pred}predictions_j$
    return y^{pred}
end for

Algorithm 3 Training Algorithm for Learning from Differences with context

Input: Dataset D, neural network model, number of neighbors n, number of epochs E

Split the dataset D into training set D^{train} and test set D^{test} using an 80%-20% split.
Initialize ΔD^{train} as an empty list.
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors.
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        Compute the differences, with concatenated context : $\Delta D_{ij}^{train} = ((X_i^{train} - X_j^{train}) : X_i^{train}, y_i^{train} - y_j^{train})$.
        Append ΔD_{ij}^{train} to ΔD^{train}.
    end for
end for
Train the `MLPRegressor` neural network model using ΔD^{train}.
return Trained neural network model $TrainedNN()$.

Algorithm 4 Prediction Algorithm for Learning from Differences with context

Input: Test dataset D^{test}, trained neural network model $TrainedNN()$, number of neighbors n

for each case $X_j^{test} \in D^{test}$ **do**
  Retrieve n similar cases from D^{train} using nearest neighbors
  Initialize y_j^{pred} as an empty list
  for each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do** ‘
    Initialize empty list $y^{pred}predictions$
    Compute the differences: $\Delta X_{ij}^{test} = ((X_j^{test} - X_i^{train}) : X_j^{test})$
    Use the trained neural network to predict: $\Delta X_{ij}^{pred} = TrainedNN(\Delta X_{ij}^{test})$
    Adapt neighbor's y values: $y^{pred}predictions_j = y_i^{train} + \Delta X_{ij}^{pred}$
  end for
  Average the adapted y values: $y_j^{pred} = \frac{1}{n} \sum_{i=1}^{n} y_i^{pred}$
  return y^{pred}
end for

The new model proposed in the paper is tested using the factorial study methodology. This process involved finding a network structure through experimental means for each dataset. It entailed training and testing using a wide range of hyperparameters to determine the optimal settings.

Despite its contributions, the paper falls short by confining its attention to regression tasks and not considering classification tasks and exploring additional variations of the learning from differences model beyond context inclusion. Incorporating original values for the base case in the generated differences dataset provides a valuable framework for understanding the observed disparities. While the model with contextual inclusion demonstrates slight improvements over the base model in learning from differences, a more comprehensive examination of the methodology is warranted for further refinement and enhancement.

The paper tested three models using the same factorial methodology: a base version, which is a basic neural network trained empirically; a differences model employing the (CBH) Case-based Heuristic for predictions, (CBH) guides how cases are retrieved with respect to the base case, and how the retrieved cases are adapted to produce a prediction, (CBH) relies on the adaptation of previous problems to solve the current issue; and a differences model with its base context included. The results yielded interesting insights, with similar performance for all three models and a slight improvement in the differences + context model. Notably, the differences and differences + context models required significantly fewer epochs to achieve accuracy compared to the basic neural network, which achieved similar accuracy after more epochs.

Another consideration in the paper [7] was the number of neighbors for both training

and prediction. The study revealed no consistent trend across datasets, suggesting dataset-specific requirements.

In conclusion, the learning from differences method achieved equal or higher accuracy compared to the base neural network model.

2.3 Learning Adaptations for Case-based classification

The paper by Ye et al. [18], titled "Learning Adaptations for Case-based Classification," investigates the application of Case-Based Reasoning (CBR) for classification tasks. It explores recent advancements in CBR, particularly focusing on replacing traditional rule-based learning with Case-Based Heuristic (CBH) network models for adaptation. The study introduces a novel model comprising three variations: segmentation of adaptation knowledge based on the classes of the source cases, training a single neural network on differences between problem solutions, and adapting from an ensemble of source cases followed by a majority vote.

The "NN-CDH" method, as proposed in the paper, represents a novel classification approach that operates on pairs of cases, where one serves as the source and the other as the target. These pairs form the basis of adaptation knowledge, which is learned through neural network models. The methodology adopted in [18] emphasizes learning adaptation knowledge through case pairs processed by neural networks.

The study introduces the first neural network-based approach to classification, termed "C-NN-CDH" or Classification with Neural Network-based Case Difference Heuristic. This approach employs neural networks to learn adaptation knowledge from pairs of cases. According to the findings, this methodology outperforms traditional statistical models, as evidenced in [10].

The first approach involves segmenting the dataset by class and training a neural network for all segmented differences set between base cases. This method offers faster training but slightly less accuracy, according to the paper. The use of these CBR methods provides at least two benefits: inertia-free lazy learning and the ability to assess different cases when running classification. Inertial free means that the model does not make assumptions based on the underlying structure of the data, and lazy learning pertains to an algorithm that will postpone generalization until it is certain a new instance needs to be classified. It will only classify as the data comes; it won't try and learn or adapt preemptively.

In the process of CBR, adaptation is often treated as the most difficult. Various methods have been developed by academics over the years to tackle the issue. For example, Leake et al. [11] extrapolate a method in which the case base adaptation cases are populated from previously successful adaptations. Another approach is to have a stored base case and query and retrieve the case most similar to it, and adapting the retrieved case to the base case [6].

In the paper by Ye et al. [18], several design questions had to be addressed. How to approach case differences and how to select case pairs for training. The paper in question [18] considers standard pair selection methods as well as a new training per selection approach based on class-class classification.

The paper [18] also addresses the application of (CDH) method for classification. It discusses the value distance metric, which is a probabilistic method to measure the similarity that allows the comparison of nominal values in a single-dimensional space.

2.3.1 An NN-CDH Approach for Classification

To achieve adaptation, a neural network is integrated into the process. The source and target cases become case pairs, and their adaptation is learned through a neural network. The neural network computes the difference between the source and target pairs. This difference calculation is a crucial step in the Case-Based Reasoning (CBR) system. Once the difference is computed, the predicted result obtained from the neural network is applied to the source solution. This adapted solution is then considered as the final result of the classification process.

The NN-CDH approach represents a fusion of neural network techniques with the principles of Case-Based Reasoning, offering a dynamic and adaptable method for classification tasks. When calculating the difference between the problem and the solution values, it is important to implement a difference function. This presents an interesting challenge for nominal values. The method presented in [18] uses an implicit calculation through machine learning techniques. This replaces traditional (CDH) case difference heuristic approaches for difference calculations. Through the use of a neural network, the paper's method hopes to include the base context in the differences' calculation. This method for generating differences has been aptly named ("C-CDH") case difference heuristic approach for classification.

When calculating the difference, an issue arrives in that we cannot explicitly calculate differences through subtraction, which was the case in the previous paper by Learning

From Differences [7]. An implicit way of learning from differences is required. In the paper by Ye et al. [18], this is called "C-CDH".

This presents an interesting challenge for nominal values. The method presented in [18] uses an implicit calculation through machine learning techniques. When discussing the Neural Network-based Case-Driven Hierarchical (NN-CDH) Approach described in [18], notation will be homogenized with the algorithms from the previous paper.

For clarification when going through the algorithms, Retrieved cases = Source Cases, Target case = The case which we are attempting to predict its y label values.

Multiple variants were created with the paper by Ye et al. [18].

2.3.2 Variant 0: Non-Network C-CDH

Variant 0, as described in [18], serves as a foundational test bed implementation for classification tasks. The algorithms presented here outline the process of building and utilizing a dictionary of case pairs, where each pair consists of a source case and its corresponding target case. Through the use of nearest neighbors, the algorithm identifies similar cases within the training dataset and organizes them based on their source solution. Subsequently, during prediction, the algorithm retrieves the nearest similar cases for each test case and utilizes the corresponding target solutions for classification. This approach embodies the essence of case-based reasoning, where solutions are adapted from similar past cases to classify new instances effectively. The following algorithms detail the steps involved in training and predicting using Variant 0's case-based heuristic approach.

Algorithm 5 Variant 0, Classification Using Case Based Heuristic, Training

Input: Dataset D, number of neighbors 1

Split D into training set D^{train} and test set D^{test} with an 80%-20% split
Create $CasePairs()$ dictionary.
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve 1 similar cases from D^{train} using nearest neighbors.
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        $CasePairs(y_j^{train}).append([X_j^{train} : X_i^{train}, y_i^{train}])$
    end for
end for

Algorithm 6 Variant 0, Classification Using Case Based Heuristic, Prediction

Input: Test dataset D^{test}, $CasePairs()$

Initialize Empty list y^{pred}
for each case $X_j^{test} \in D^{test}$ **do**
    Retrieve 1 similar cases from D^{train} using nearest neighbors
    Initialize y_j^{pred} as an empty list
    for each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do**
        Retrieve $r = 1NearestNeighbor(CasePairs(y_i^{train}), [X_i^{train} : X_j^{test}])$
        $y_j^{pred} = r$
    end for
end for
return y^{pred}

2.3.3 Variant 1 - 2: C-NN-CDH

Variants 1 - 2, known as C-NN-CDH, represent a pivotal advancement in classification methodologies by integrating neural networks into the case-based heuristic framework. In this variant, the traditional case-based heuristic approach is augmented with the power of neural networks to adapt solutions from similar cases. The algorithms presented here delineate the training and prediction processes, where a neural network model learns adaptation knowledge from case pairs derived from the training dataset. Unlike Variant 0, which relies solely on case similarity for adaptation, C-NN-CDH leverages the expressive capacity of neural networks to capture intricate relationships between cases and their solutions. Through the fusion of case-based reasoning principles with neural network techniques, Variant 1 - 2 offers a dynamic and adaptable approach to classification tasks, capable of learning complex adaptation patterns and improving classification accuracy.

2.3.4 Variant 3 - 5: C-NN-CDH

Variants 3 - 5 of the classification neural network using case-based heuristics introduce a novel approach to classification by grouping case pairs based on their source solutions. In this framework, multiple specialized adaptation neural networks are employed, each trained to adapt cases of specific solutions to all solutions. Inspired by the ensemble of adaptations for classification approach, these variants share a common grouping and training method but differ in their testing strategies. The algorithms delineate the training process, where case pairs are organized and used to train specialized neural networks for adaptation. The variation arises in the prediction algorithm, where different strategies are employed to classify test instances based on the learned adaptation knowledge. The following section details the nuances of each variant's

Algorithm 7 Variant 1 - 2: Classification Using Neural Net Case Based Heuristic (Training)

Input: Dataset D, neural network model, number of neighbors n, number of epochs E

Split D into training set D^{train} and test set D^{test} with an 80%-20% split
Create $CasePairs$ list.
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
  Retrieve 1 similar case from D^{train} using nearest neighbors.
  for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
    if Variant 1 **then**
      $CasePairs$.append($[X_j^{train} : X_i^{train}, y_i^{train}]$)
    else
      if Variant 2 **then**
        $CasePairs$.append($[X_j^{train} : X_i^{train} : y_j^{train}, y_i^{train}]$)
      end if
    end if
  end for
end for
Train the neural network model using $CasePairs()$
return Trained neural network model $TrainedNN()$.

Algorithm 8 Variant 1 - 2: Classification Using Neural Net Case Based Heuristic (Prediction)

Input: Test dataset D^{test}, trained neural network model $TrainedNN()$

1: Initialize Empty list y^{pred}
2: **for** each case $X_j^{test} \in D^{test}$ **do**
3: Retrieve 1 similar case from D^{train} using nearest neighbors
4: **for** each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do**
5: Use the trained neural network to predict: $y_j^{pred} = TrainedNN([X_i^{train} : X_j^{test} : y_i^{train}])$
6: **end for**
7: **end for**
8: **return** y^{pred}

prediction methodology and discusses their implications for classification accuracy and adaptability.

Algorithm 9 Variant 3 - 5: Classification Neural Network Using Case Based Heuristic (Training)

Input: Dataset D, number of neighbors 1

Split D into training set D^{train} and test set D^{test} with an 80%-20% split
Create $CasePairs()$ dictionary.
Create $AdaptNN()$, a dictionary to store each Neural Network
List of unique Classes $UniqueClasses$
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve 1 similar case from D^{train} using nearest neighbors.
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        $CasePairs(y_j^{train}).append([X_j^{train} : X_i^{train}, y_i^{train}])$
    end for
end for
for each unique $Class$ in $UniqueClasses$ **do**
    Create a new Classification Neural Network, trained on Case Pairs
    $AdaptNN(Class).fit(CasePairs(Class))$
end for
return Trained neural network models dictionary $AdaptNN()$.

In variant (5), the prediction method utilizes a Class-to-Class approach for retrieving neighbors. This method retrieves the most similar neighbor for each class from the training dataset. Once the nearest neighbors for each class are identified, the corresponding specialized neural networks associated with those classes are employed to generate predictions for the target instance. Subsequently, a majority vote is conducted among the predictions from the specialized neural networks to determine the final classification outcome.

It is worth considering re-implementing the algorithm to allow for the retrieval of multiple neighbors (N) from each class during the prediction stage. This adjustment could potentially lead to improved results, particularly when dealing with large and varied datasets.

Variant (5) stands out as the most successful approach in the study, primarily due to its Class-to-Class retrieval of neighbors during the prediction stage. By employing multiple specialized neural networks, the variant reduces training time while effectively leveraging implicit learning through differences for classification tasks. This method demonstrates its efficacy in adapting to diverse datasets and achieving robust classification performance.

Algorithm 10 Variant 3 - 5: Classification Using Neural Net Case-Based Heuristic (Prediction)

Input: D^{test}, $AdaptNN()$, $UniqueClasses$, and $CasePairs()$

Initialize an empty list y^{pred}
for each case $X_j^{test} \in D^{test}$ **do**
  if Variant 3 **then**
    Retrieve 1 similar case from D^{train} using nearest neighbors
    for each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do**
      $y_j^{pred} = AdaptNN(y_i^{train}).predict([X_i^{train} : X_j^{test}])$
    end for
  else
    if Variant 4 **then**
      Create an empty list *predictions*
      Retrieve N similar cases from D^{train} using nearest neighbors
      for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        $predictions$.append($Adapt(y_i^{train}).predict([X_i^{train} : X_j^{test}])$)
      end for
      Perform a majority vote on *predictions*
      $y_j^{pred} = MajorityVote(predictions)$
    else
      if Variant 5 **then**
        Create an empty list *preds*
        for each unique class $Class$ in $UniqueClasses$ **do**
          Retrieve 1 similar case from $CasePairs(Class)$
          for each similar case $(X_i^{CasePair}, y_i^{CasePair}) \in CasePairs(Class)$ **do**
            for each $(X_z^{train}, X_q^{train}) \in X_i^{CasePair}$ **do**
              $preds.append(AdaptNN(Class).predict(X_z^{train} : X_j^{test}))$
            end for
          end for
          Perform a majority vote on *preds*
          $y_j^{pred} = MajorityVote(preds)$
        end for
      end if
    end if
  end if
end for
return y^{pred}

2.3.5 Evaluation Summary

Within the paper [18] the experiments conducted on two different datasets provided answers to several key questions.

The effectiveness of neural network learning adaptation knowledge was demonstrated through consistent performance of one or more C-NN-CDHs, which often outperformed the neural network classifier. Despite the possibility of neural networks learning to discard the source problem and rely solely on the target problem, the non-zero weights associated with the source problem and significant performance differences between C-NN-CDHs and the neural network indicate that C-NN-CDHs effectively learn adaptation knowledge. This suggests that if a neural network can handle the classification task directly, it can also learn adaptation knowledge or the relation between pairs of cases in the task domain.

Surprisingly, variant (1) performed almost identically to variant (2), which also considers the source solution in adaptation. This suggests that the source solution, heavily coupled with the source problem, may not provide additional useful information for adaptation.

In terms of segmenting case pairs by source case solution, variants (1, 2) generally outperformed variants (3, 4) in accuracy on most datasets. This could be because a single adaptation neural network in (1, 2) is well-trained with all pairs of cases, while a specialized adaptation neural network in (3, 4) is trained with segmented examples. However, variants (3, 4) showed faster convergence during training due to training on segmented examples.

The impact of the ensemble of adaptations was investigated, with EAC-NN-CDH (variant (4)) performing similarly to its counterpart variant (3) without ensemble. This suggests that the generalization power of C-NN-CDH produces stable predictions that an ensemble version does not significantly alter.

The usefulness of the class-to-class approach for adaptation was demonstrated by C2C-NN-CDH (variant (5)), which performed differently from and often better than the other C-NN-CDHs. C2C-NN-CDH reaches its prediction by collecting evidence from diverse source cases from all classes, which can provide more global support, especially when there are multiple classes. Moreover, the C2C approach offers the possibility of explanation with contrastive evidence.

2.3.6 Conclusions on Learning adaptations for case based classification: A Neural Network approach

The conclusion of the paper [18] is that (variant (5)) is worth investigating using its C2C-NN-CDH approach to classification. As it yielded the most promising results when compared to all the other variants tested in the paper. The grouping of the dataset based on class outcome and training a specialized neural network, has its benefits in terms of speedup. However, the retrieval of N cases across all classes during testing, has shown to yield better results, in a way that allows an interpretation of the results.

Chapter 3

Design and Implementation

Objectives and Contributions

The design chapter of this paper serves as a comprehensive guide, presenting various algorithms and their implementations in the context of our novel framework. Our primary objectives are to elucidate the intricacies of these algorithms, introduce innovative variations, and showcase the integration of our framework within the broader machine learning landscape.

The tables provided below offer a structured roadmap through our design chapter, delineating where and how different algorithms are described. Notably, the algorithms discussed originate from seminal works such as [7] and [18]. However, in addition to presenting established methodologies, we introduce novel variations that represent fresh approaches to classification and regression tasks.

Key innovations include Variant 2, which introduces duplication based on distances, and Variant 3, incorporating a column indicating each neighbor's distance to the base case. Notably, we employ a new type of classifier, distinct from those mentioned in [7]. Variants 1, 2, and 3 exemplify our endeavor to explore uncharted territory within the learning from differences methodology, offering novel perspectives on classification.

Furthermore, our introduction of the `LingerImplicitRegressor` presents a groundbreaking approach to regression tasks. Leveraging methodologies outlined in [18], this novel regressor extends the learning from differences framework, offering a fresh perspective on regression problems.

By presenting a cohesive framework that integrates established algorithms, innovative

variations, and our novel contributions, these tables provide a clear overview of the design landscape. Our aim is to facilitate understanding and navigation through the complexities of machine learning methodologies, empowering researchers and practitioners to leverage our framework effectively in their endeavors.

3.1 Design and Implementation Guide

3.1.1 LingerRegressor and LingerClassifier Models

The following models are based on the algorithms outlined in [7]. Variations such as Variant 2 and Variant 3 introduce innovative adaptations to the base models, providing enhanced functionality and performance.

`LingerRegressor`		
Full Name	Acronym	Description
`LingerRegressor` Section: 3.3	LR	Base Linger regressor, no variations
`LingerRegressor` Variant 1 Section: 3.5.1	LRV1	Linger regressor with the addition of context, where context is the original base case.
`LingerRegressor` Variant 2 Section: 3.5.2	LRV2	Linger regressor integrated with duplication based on distance, where differences are duplicated based on the neighbors' proximity to the base case.
`LingerRegressor` Variant 3 Section: 3.5.3	LRV3	Linger regressor integrated with a distance column indicating the distance of each neighbor from the base case.

Table 3.1: Regression Models based on the algorithms in [7]

`LingerClassifier`		
Full Name	Acronym	Description
`LingerClassifier` Section: 3.4	LC	Base Linger classifier, no variations
`LingerClassifier` Variant 1 Section: 3.5.1	LCV1	Linger classifier with the addition of context, where context is the original base case.
`LingerClassifier` Variant 2 Section: 3.5.2	LCV2	Linger classifier integrated with duplication based on distance, where differences are duplicated based on the neighbors' proximity to the base case.
`LingerClassifier` Variant 3 Section: 3.5.3	LCV3	Linger classifier integrated with a distance column indicating the distance of each neighbor from the base case.

Table 3.2: Classifier models based on the algorithms in [7]

3.1.2 `LingerImplicitClassifier` and `LingerImplicitRegressor` Models

The classifier models presented here are based on algorithms from [18], with Variations 1 and 2 also originating from the same source.

The regressor models in this section extend the learning from differences framework outlined in [18] to regression tasks. This innovative approach maintains the core principles of the methodology while introducing adaptations tailored to address the unique challenges of regression problems.

`LingerImplicitClassifier`		
Full Name	Acronym	Description
`LingerImplicitClassifier` Section: 3.8	LIC	Base Linger Implicit Classifier selects the N nearest neighbors from each class
`LingerImplicitClassifier` Variant 1 Section: 3.8.5	LICV1	Selects N random items from each class
`LingerImplicitClassifier` Variant 2 Section: 3.8.5	LICV2	Selects the single nearest neighbor from each class

Table 3.3: LingerImplicitClassifier models based on the algorithms in [18]

.

`LingerImplicitRegressor`		
Full Name	Acronym	Description
`LingerImplicitRegressor` Section: 3.3	LIR	Base Linger Implicit Regressor selects the N nearest neighbors from each class during prediction
`LingerImplicitRegressor` Variant 1 Section: 3.9.5	LIRV1	Selects N random items from each class
`LingerImplicitRegressor` Variant 2 Section: 3.9.5	LIRV2	Selects the single nearest neighbor from each class
Regression to Classification Converter		
Full Name	Acronym	Description
Regression to Classification Converter Section: 3.9.7	RTCC	Converts regression problems into classification problems

Table 3.4: LingerImplicitRegressor models based on the algorithms in [18]

3.1.3 `LingerImageClassifier` and `LingerImageRegressor`

`LingerImageClassifier`		
Full Name	Acronym	Description
`LingerImageClassifier` Section: 3.11	(LIMC)	Image Classifier that learns from differences

Table 3.5: Image Classifier Using learning from difference methodologies seen in [7]. Innovative in its application to classification image data sets.

`LingerImageRegressor`		
Full Name	Acronym	Description
`LingerImageRegressor` Section: 3.12	(LIMR)	Image Regressor that learns from differences

Table 3.6: Image Regressor Using learning from difference methodologies seen in [7]. Innovative in its application to classification image data sets.

3.2 Learning From Differences

3.2.1 Introduction and Motivation

Our design process is anchored in the seminal work of [7], which laid the groundwork for the learning from differences method. Building upon this foundation, our primary objective is to enhance and extend these algorithms, creating a more dynamic and adaptable framework.

Inspired by Scikit-learn's modular design philosophy, we aim to develop models that not only match but surpass its rigorous hyperparameter testing capabilities. Our goal is to introduce multiple new hyperparameters accessible through familiar pipelines, thereby enhancing flexibility and functionality across diverse machine learning tasks and datasets.

As we progress from implementing base models to introducing variations, our focus shifts to optimizing the performance of both the `LingerRegressor` and `LingerClassifier`. These variations are meticulously crafted to improve training efficiency and predictive accuracy, ultimately enhancing overall model performance.

Named `LingerRegressor` and `LingerClassifier`, our custom models are designed to leverage the training from differences approach, distinguishing them within the machine learning landscape.

Central to our efforts is ensuring the seamless testability of introduced variations, akin to Scikit-learn's comprehensive hyperparameter testing framework. This rigorous analysis empowers stakeholders to make informed decisions about the efficacy and applicability of our models to specific tasks.

However, we encountered the critical challenge of compatibility during model development. Our codebase is structured as a downloadable package, seamlessly integrating with Scikit-learn's ecosystem to ensure interoperability across various datasets and environments. This compatibility enhances accessibility and usability for practitioners in diverse machine learning endeavors.

3.3 Linger Regression Base Model: `LingerRegressor` (LR)

3.3.1 Objective

The Linger Regression Base Model integrates principles of Case-Based Reasoning (CBR) into regression analysis to enhance interpretability and create a performative model. The objectives include:

- Developing a dynamic regression model using differences between a case and its nearest neighbors, adapting retrieved solutions to the target solution.
- Ensuring compatibility with Scikit-learn for accessibility, ease of implementation, and extensive hyperparameter testing.
- Enabling effective interpretation of predictions for users.
- Implementing the learning from differences algorithm with enhancements for improved performance.
- Creating an easily extendable code base to allow for variations.

Its intended use case is for regression problems that could benefit from a more transparent learning process.

3.3.2 Architecture

The `LingerRegressor` combines K-nearest neighbors (KNN) and `MLPRegressor` from Scikit-learn for flexibility and accuracy. A K-nearest neighbors (KNN) is used in conjunction with a `MLPRegressor` in the fit phase of the machine. During the predict phase, a K-nearest neighbors (KNN) is used once again with the now trained `MLPRegressor` to get the differences' prediction or K neighbors.

3.3.3 Components

- KNN module: Computes nearest neighbors and extracts differences during both training and prediction phases.

- `MLPRegressor`: Offers flexibility in activation functions, solvers, and hyperparameters for regression.
- Supplementary Features: Weighted KNN, Context Addition, Additional Distance Column, and Duplication Based on Distance. Each supplementary feature can be activated through a hyperparameter.

3.3.4 Parameterization

Extensive parameterization allows fine-tuning of KNN, `MLPRegressor`, and additional features. All the same hyperparameters are available as in a conventional (KNN) and neural network, with additional hyperparameters for `LingerRegressor`.

3.3.5 Training and Prediction Design

Training Design

Given dataset D, split it into D^{train} and D^{test} with an 80%-20% split through random selection. For each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$, retrieve n similar cases from D^{train}, $(X_j^{train}, y_j^{train}) \in D^{train}$ using Scikit-learn's neighbors module. Compute the differences between C_i and its neighbors and append them to ΔD^{train} as $(X_i^{train} - X_j^{train}, y_i^{train} - y_j^{train})$.

Algorithm 11 Training Algorithm for Learning from Differences

Split D into training set D^{train} and test set D^{test} with an 80%-20% split
Initialize ΔD^{train} as an empty list
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta D_{ij}^{train} = (X_i^{train} - X_j^{train}, y_i^{train} - y_j^{train})$
        Append ΔD_{ij}^{train} to ΔD^{train}
    end for
end for
Train the MLPRegressor neural network model using ΔD^{train} over a few epochs for later predictions.
Return Trained neural network model $TrainedNN()$

Prediction Design

During prediction, for each case X_j^{test}, retrieve n similar cases from D^{train}. Compute the differences $\Delta X_{ij}^{test} = X_j^{test} - X_i^{train}$ for each neighbor. Use the trained neural network to predict ΔX_{ij}^{test}, resulting in ΔX_{ij}^{pred}. For each set of retrieved neighbors from D^{train}, calculate $y_i^{train} + \Delta X_{ij}^{pred}$ and add each n neighbors adapted y values. Once summed, divide by n to obtain y_j^{pred}. Compare y_j^{pred} to y_j^{test} to evaluate accuracy.

Algorithm 12 Prediction Algorithm for Learning from Differences

Input: Test dataset D^{test}, trained neural network model $TrainedNN()$, number of neighbors n

```
Initialize empty list y^pred
for each case X_j^test ∈ D^test do
    Initialize empty list y_j^pred
    Retrieve n similar cases from D^train using nearest neighbors
    Initialize empty list y^predpredictions
    for each similar case (X_i^train, y_i^train) ∈ D^train do
        Compute the differences: ΔX_ji^test = X_j^test − X_i^train
        Use the trained neural network to predict: TrainedNN(ΔX_ji^test) = ΔX_ji^pred
        Adapt neighbor's y values: y_ji^pred = y_i^train + ΔX_ji^pred
    end for
    Average the adapted y values: y_j^pred = (1/n) Σ_{i=1}^{n} y_ji^pred
end for
return y^pred
```

3.3.6 Training and Prediction Implementation

Training Implementation

During training, the algorithm fits the base Linger Regressor to the input dataset X and the corresponding label vector y. It computes the nearest neighbors for each sample in X and constructs the input features and target labels for the regressor by calculating the differences between each sample and its neighbors. These differences are then used to train the regressor, and the trained model along with the training data are stored for later use. It makes use of scikit-learns [15] NearestNeighbors, numpy and MLPRegressor.

Algorithm 13 Training Implementation Algorithm for Base Linger Regressor

Input: Dataset, X, label vector y, number of neighbors $n\_neighbors\_1$

function FIT(X, y)
    Increment $n\_neighbours\_1$ ▷ Fit scikit-learns NearestNeighbors function to X
    $neighbors = NearestNeighbors(n\_neighbors = n\_neighbors\_1).fit(X)$
    Compute nearest neighbors for each sample in X
    Compute the indexes and distances, of nearest neighbors in X
    distances, indices $= neighbors.kneighbors(X)$
    $differences\_X = []$
    $differences\_y = []$
    if $variant == 1$ (LRV1) **then**
        Call $addition\_of\_context\_func\_fit()$
        $differences\_X, differences\_y =$
     $addition\_of\_context\_func\_fit(X, y, indices, differences\_X, differences\_y)$
    else if $variant == 2$ (LRV2) **then**
        Call $duplicate\_difference\_based\_on\_distance()$
        $differences\_X, differences\_y =$
     $duplicate\_difference\_based\_on\_distance(differences\_X, differences\_y, distances)$
    else if $variant == 3$ (LRV3) **then**
        Call $add\_additional\_distance\_column(differences\_X, distances)$
        $differences\_X = add\_additional\_distance\_column(differences\_X, distances)$
    else
        for each sample i in nearest neighbors indices **do**
            $base \leftarrow$ Index of base case
            for each neighbor n of i **do**
                Append $(X[base] - X[n])$ to $differences\_X$
                Append $(y[base] - y[n])$ to $differences\_y$
            end for
        end for
    end if
    Fit `MLPRegressor` with $differences\_X$ and $differences\_y$
    Set $train\_X = X$
    Set $train\_y = y$
    return Fitted `LingerRegressor`
end function

Prediction Implementation

During prediction, the algorithm utilizes the fitted Nearest Neighbors model to identify the nearest neighbors for each sample in the input data X. It then computes the differences between each sample in X and its corresponding neighbors, preparing the input features for the regressor. After predicting the differences using the trained regressor, the algorithm aggregates the predictions to obtain the final predicted target values for each sample in X. If weighted KNN is enabled, the predictions are weighted based on the distances to the neighbors, providing more influence to closer neighbors in the prediction process. This offers a different form of nearest neighbors. Where values are weighted based on their proximity to the base case. It makes use of scikit-learns [15] NearestNeighbors, and numpy.

Algorithm 14 Prediction Implementation Algorithm for Base Linger Regressor

Input: Input data, X, number of neighbors $n\_neighbors\_2$

function PREDICT(X)
    Fit Nearest Neighbors model with $n\_neighbours\_2$ using $train\_X$
    Compute nearest neighbors for each sample in X,
    Compute the indexes and distances, of nearest neighbors in X
    Initialize empty list $differences\_test\_X$
    $is\_sparse\_X \leftarrow$ Check if X is sparse
    if is_sparse_X **then**
        Convert X to dense array
    end if
    for each sample $test$ in X **do**
        for each nearest neighbor $nn\_in\_training$ in $indices[test]$ **do**
            Append $(X[test,:] - train\_X[indices[test][nn\_in\_training],:])$ to $differences\_test\_X$
        end for
    end for
    if $variant == 1$ (LRV1) **then**
        Call $addition\_of\_context\_func\_pred(differences\_test\_X, X)$
        $differences\_test\_X_{=}addition\_of\_context\_func\_pred(differences\_test\_X, X)$
    else if $variant == 3$ (LRV3) **then**
        Call $add\_additional\_distance\_column(differences\_test\_X, distances)$
        $differences\_X = add\_additional\_distance\_column(differences\_test\_X, distances)$
    end if
    Continue predict with next function.
end function

Algorithm 15 Prediction Implementation Algorithm for Base Linger Regressor (Part 2)

Input: Continuation from Part 1

function PREDICT_CONTINUE
    $predictions \leftarrow$ Predict using `LingerRegressor` with $differences\_test\_X$
    Split $predictions$ into batches of size $n\_neighbours\_2$
    Initialise y_pred as an empty list
    $y\_pred = []$
    if Weighted KNN is enabled **then**
        for each $indexes$, $differences$, $dist$ in $zip(indices, predictions, distances)$ **do**
            Compute weights for each neighbor
            $weights = 1 \div (dist + 1e - 8)$
            Compute weighted sum of target values
            $weighted\_sum = sum(w \times (train\_y[i] + d)$ for i, d, w in zip($indexes$, $differences$, $weights$))
            Compute weighted average
            $weighted\_avg = weighted\_sum \div sum(weights)$
            Append $weighted\_avg$ to y_pred
        end for
    else
        for each $indexes$, $differences$ in $zip(indices, predictions)$ **do**
            Compute sum of target values
            $sum\_result = sum((train\_y[i] + d) for i, d in zip(indexes, differences))$
            Compute average
            $avg\_res = sum\_result \div n\_neighbours\_2$
            Append avg_res to y_pred
        end for
    end if
    return y_pred
end function

3.3.7 Evaluation

Evaluation metrics include Mean Squared Error (MSE), Root Mean Squared Error (RMSE), and R-squared (R^2) to assess model accuracy and performance.

3.3.8 Conclusion

In conclusion, the `LingerRegressor` model presents a novel approach to regression analysis by integrating principles of Case-Based Reasoning (CBR) and the learning from differences methodology. Through the combination of K-nearest neighbors (KNN) and `MLPRegressor`, the model offers flexibility, accuracy, and interpretability in regression tasks. The implementation of supplementary features further enhances its adaptability and performance. Despite its limitations, the model shows promise in addressing the challenges of transparent and dynamic regression modeling. Future research can focus on refining the model's robustness to noise and improving its interpretability for complex datasets. Overall, the `LingerRegressor` model contributes to the advancement of machine learning methodologies, providing a valuable tool for regression analysis in diverse domains where a more interpretable model is needed.

3.4 Linger Classification Base Model: `LingerClassifier` (LC)

3.4.1 Objective

The objective of the Linger Classification Base Model (`LingerClassifier`) is to extend the learning from differences methodology to classification tasks. The primary goals include:

- Developing a versatile classification model that utilizes differences between cases for prediction.
- Ensuring compatibility with Scikit-learn for seamless integration and extensive hyperparameter testing.
- Providing interpretable predictions to facilitate user understanding.

- Implementing enhancements to the learning from differences algorithm for improved classification performance.
- Designing a modular and extensible codebase to support future variations and enhancements.

The `LingerClassifier` is tailored for classification problems where interpretability and adaptability are critical.

3.4.2 Architecture

Similar to the regression counterpart, the `LingerClassifier` integrates K-nearest neighbors (KNN) and Scikit-learn's `MLPClassifier` for classification tasks. During training, KNN is used to compute nearest neighbors, and during prediction, the differences are used to adapt the class labels.

3.4.3 Components

The key components of the `LingerClassifier` include:

- KNN Module: Computes nearest neighbors and extracts differences during training and prediction phases.
- `MLPClassifier`: Offers flexibility in activation functions, solvers, and hyperparameters for classification.
- Additional Features: Similar to the regression model, features like weighted KNN, context addition, additional results' column, and duplication based on distance can be activated through hyperparameters.

3.4.4 Parameterization

The model provides extensive parameterization options for fine-tuning KNN, `MLPClassifier`, and additional features. Hyperparameters for the classification model mirror those available in standard KNN and neural networks, with additional parameters specific to `LingerClassifier`.

3.4.5 Training and Prediction Design

The training and prediction algorithms for the `LingerClassifier` are similar to those of the `LingerRegressor`, with adjustments made to accommodate classification tasks.

Training Algorithm Design

During training, the algorithm computes differences between cases and their nearest neighbors, similar to the regression model. However, the adaptation of class labels is based on the majority class among the nearest neighbors.

Algorithm 16 Training Algorithm for `LingerClassifier`

Split the dataset D into training set D^{train} and test set D^{test} using an 80%-20% split.
Initialize ΔD^{train} as an empty list.
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors.
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta D_{ij}^{train} = (X_i^{train} - X_j^{train}, y_i^{train} - y_j^{train})$.
        Append ΔD_{ij}^{train} to ΔD^{train}.
    end for
end for
Train the `MLPClassifier` neural network model using ΔD^{train}.
return Trained neural network model $TrainedNN()$.

Prediction Algorithm Design

During prediction, the algorithm adapts class labels based on the majority class among the nearest neighbors' labels.

3.4.6 Training and Prediction Implementation

Training Algorithm Implementation

The provided algorithm outlines the prediction process for the Linger Classifier implemented in Python. It begins by fitting a Nearest Neighbors model with a specified number of neighbors using the training data. Next, it computes the nearest neighbors for each sample in the input data and initializes an empty list to store the differences

Algorithm 17 Prediction Algorithm for Learning from Differences for Classification

Input: Test dataset D^{test}, trained neural network model $TrainedNN()$, number of neighbors n

Initialize empty list y^{pred}
for each case $X_j^{test} \in D^{test}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors
    Initialize empty list y_j^{pred}
    for each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta X_{ji}^{test} = X_j^{test} - X_i^{train}$
        Use the trained neural network to predict: $TrainedNN(\Delta X_{ji}^{test}) = \Delta X_{ji}^{pred}$
        Adapt neighbor's y value: $y_{ji}^{pred} = y_i^{train} + \Delta X_{ji}^{pred}$
    end for
    Perform majority voting on $y^{pred} predictions$ to obtain y_j^{pred}
end for
return y^{pred}

between the test samples and their corresponding neighbors. If the input data is sparse, it is converted to a dense array for compatibility. The algorithm then iterates through each test sample and its nearest neighbors to calculate the feature differences. After computing the differences, it proceeds to make predictions using the trained classifier. If weighted KNN is enabled, it calculates the weighted results based on the differences and distances to determine the item with the highest weighted sum as the prediction. Otherwise, it finds the most common item among the predictions. Finally, the predicted target values are aggregated and returned as the output of the algorithm.

Algorithm 18 Training Implementation Algorithm for Base Linger Classifier

Input: Dataset, X, label vector y, number of neighbors $n\_neighbors\_1$

function FIT(X, y)
    Increment $n\_neighbors\_1$
    $neighbors \leftarrow$ Fit scikit-learn's `NearestNeighbors` function to X
    Compute nearest neighbors for each sample in X
    distances, indices $\leftarrow$ neighbors.kneighbors(X)
    $differences\_X \leftarrow []$
    $differences\_y \leftarrow []$
    if $variant == 1$ (LCV1) **then**
        Call $addition\_of\_context\_func\_fit()$
        $differences\_X, differences\_y = addition\_of\_context\_func\_fit(X, y, indices, differences\_X, differences\_y)$
    else if $variant == 2$ (LCV2) **then**
        Call $duplicate\_difference\_based\_on\_distance()$
        $differences\_X, differences\_y = duplicate\_difference\_based\_on\_distance(differences\_X, differences\_y, distances)$
    else if $variant == 3$ (LCV3) **then**
        Call $add\_additional\_distance\_column(differences\_X, distances)$
        $differences\_X = add\_additional\_distance\_column(differences\_X, distances)$
    else
        for each sample i in nearest neighbors indices **do**
            $base \leftarrow$ Index of base case
            for each neighbor n of i **do**
                Append $(X[base] - X[n])$ to $differences\_X$
                Append $(y[base] - y[n])$ to $differences\_y$
            end for
        end for
    end if
    Fit `MLPClassifier` with $differences\_X$ and $differences\_y$
    Set $train\_X = X$
    Set $train\_y = y$
    return Fitted `LingerClassifier`
end function

Prediction Algorithm Implementation

The prediction algorithm for the Linger Classifier, implemented in Python, outlines the process of predicting target values for the input data. Initially, it fits a Nearest Neighbors model with a specified number of neighbors using the training data. Then, it computes the nearest neighbors for each sample in the input data and initializes an empty list to store the feature differences between the test samples and their corresponding neighbors. If the input data is sparse, it is converted to a dense array for compatibility. The algorithm iterates through each test sample and its `NearestNeighbors` to calculate the feature differences. After computing the differences, it proceeds to make predictions using the trained `LingerClassifier`. If weighted KNN is enabled, it calculates the weighted sum of target values for each neighbor based on the differences and distances, and selects the item with the highest weighted sum as the prediction. Otherwise, it finds the most common item among the predictions. Finally, the predicted target values are aggregated and returned as the output of the algorithm.

Algorithm 19 Prediction Algorithm for Linger Classifier (Part 1)

Input: Input data, X, number of neighbors $n\_neighbours\_2$

function PREDICT(X)
    Fit Nearest Neighbors model with $n\_neighbours\_2$ using $train\_X$
    Compute nearest neighbors for each sample in X
    Initialize empty list $differences\_test\_X$
    $is\_sparse\_X \leftarrow$ Check if X is sparse
    if is_sparse_X **then**
        Convert X to dense array
    end if
    for each sample $test$ in X **do**
        for each nearest neighbor $nn\_in\_training$ in $indices[test]$ **do**
            Append $(X[test, :] - \text{train\_X}[indices[test][nn\_in\_training], :])$ to $differences\_test\_X$
        end for
    end for
    if $variant == 1$ (LCV1) **then**
        Call $addition\_of\_context\_func\_pred(differences\_test\_X, X)$
        $differences\_test\_X$=$addition\_of\_context\_func\_pred(differences\_test\_X, X)$
    else if $variant == 3$ (LCV3) **then**
        Call $add\_additional\_distance\_column(differences\_test\_X, distances)$
        $differences\_X = add\_additional\_distance\_column(differences\_test\_X, distances)$
    end if
end function

Algorithm 20 Prediction Algorithm for Linger Classifier (Part 2)

Input: Continuation from Part 1

function PREDICT_CONTINUE
    $predictions \leftarrow$ Predict using `LingerClassifier` with $differences\_test\_X$
    Split $predictions$ into batches of size $n\_neighbours\_2$
    Initialise y_pred as an empty list
    $y\_pred = []$
    if Weighted KNN is enabled **then**
        for each $indexes, differences, distance$ in $zip(indices, differences\_test\_X, distances)$ **do**
            Compute weights for each neighbor
            $weights = 1 \div (dist + 1e - 8)$
            Compute weighted sum of target values,
            $weighted\_sum = []$
            weighted_sum = $[w \times (train\_y[i] + d)]$ for i, d, w in zip(indexes, differences, weights)
            Find the item with the highest weighted sum
            $max\_sum = max(weighted\_sum)$
            Append max_sum to y_pred
        end for
    else
        for each $indexes$, $differences$ in $zip(indices, predictions)$ **do**
            $results = [train_y[i] + dfori, dinzip(indexes, differences)]$
            Find the most common item in $results$
            $result = most_common(results)$
            Append the $result$ to y_pred
        end for
    end if
    return y_pred
end function

3.4.7 Evaluation

Evaluation metrics for the `LingerClassifier` include accuracy, precision, recall, F1-score, and confusion matrix analysis. These metrics provide insights into the model's performance and its ability to classify instances accurately.

3.4.8 Conclusion

The `LingerClassifier` offers a novel approach to classification tasks by integrating the learning from differences methodology. Its modular design and extensible architecture make it well-suited for a wide range of classification problems, particularly those where interpretability and adaptability are paramount.

For more details on the `LingerClassifier`, and `LingerRegressor` variations, refer to Table 3.7.

3.5 Variations For `LingerRegressor` (LR) and `LingerClassifier` (LC)

3.5.1 Variation 1: Addition of Context Design and Implementation (LRV1, LCV1)

Objective

The objective of adding context is to incorporate the original attribute values of the source cases into the generated differences array. This ensures that the new datasets used for fitting and prediction contain both the original attribute values of the source cases and the differences between them and their nearest neighbors. This may aid in the learning process and improve the model. Its design is the same for both the `LingerRegressor` and `LingerClassifier`.

Training Design

Algorithm 21 Training Algorithm for Learning from Differences, with context included in `LingerClassifier` and `LingerRegressor`

Split the dataset D into training set D^{train} and test set D^{test} using an 80%-20% split.
Initialize ΔD^{train} as an empty list.
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors.
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        Compute the differences, with concatenated context:
        $\Delta D_{ij}^{train} = ((X_i^{train} - X_j^{train}) : X_i^{train}, y_i^{train} - y_j^{train})$.
        Append ΔD_{ij}^{train} to ΔD^{train}.
    end for
end for
Train the `MLPClassifier` or `MLPRegressor` neural network model using ΔD^{train}.
return Trained neural network model $TrainedNN()$.

Prediction Design

Algorithm 22 Prediction Algorithm for Variant 1, context included.

Input: Test dataset D^{test}, trained neural network model $TrainedNN()$, number of neighbors n

Initialize y^{pred} as an empty list.
for each case X_j^{test} in D^{test} **do**
    Retrieve n similar cases from D^{train} using nearest neighbors.
    Initialize y_j^{pred} as an empty list.
    for each similar case $(X_i^{train}, y_i^{train})$ in D^{train} **do**
        Compute the differences: $\Delta X_{ji}^{test} = ((X_j^{test} - X_i^{train}) : X_j^{test})$.
        Use the trained neural network to predict: Trained NN$(\Delta X_{ji}^{test}) = \Delta X_{ji}^{pred}$.
        Adapt neighbor's y values: $y_{ji}^{pred} = y_i^{train} + \Delta X_{ji}^{pred}$.
    end for
    if Regression **then**
        Average the adapted y values: $y_j^{pred} = \frac{1}{n} \sum_{i=1}^{n} y_j^{pred}$.
    else if Classification **then**
        Perform majority voting on y_j^{pred} to obtain y_j^{pred}.
    end if
end for
return y^{pred}

Training Implementation

This algorithm is called during the Fit stage of both the `LingerClassifier`, and `LingerRegressor`.

Algorithm 23 Context Addition Function for Base Linger Classifier

Input: Dataset X, label vector y, nearest neighbor indices $indices$, differences in X $differences\_X$, differences in y $differences\_y$ **Output:** Updated differences in X $differences\_X$, updated differences in y $differences\_y$

function $addition\_of\_context\_func\_fit(X, y, indices, differences\_X, differences\_y)$
  for i in $indices$ **do**
    $base \leftarrow i[0]$
    $neighbors \leftarrow i[1:]$
    for n in $neighbors$ **do**
      $base\_case \leftarrow X[base]$
      $diff \leftarrow X[base] - X[n]$
      $combined\_list \leftarrow []$
      for $pair$ in $zip(diff, base\_case)$ **do**
        for $item$ in $pair$ **do**
          Append $item$ to $combined\_list$
        end for
      end for
      Append np.array($combined\_list$) to $differences\_X$
      Append $y[base] - y[n]$ to $differences\_y$
    end for
  end for
  return $differences\_X$, $differences\_y$
end function

Prediction Implementation

This algorithm is called during the predict algorithm for both the `LingerClassifier`, and `LingerRegressor`.

Algorithm 24 Context Addition Function for Base Linger Classifier during Prediction

Input: Differences in test data $differences\_test\_X$, input data X **Output:** Combined differences in test data $combined\_differences\_test\_X$

```
function addition_of_context_func_pred(differences_test_X, X)
    combined_differences_test_X ← []
    duplicated_test_data ← [item for item in X for _ in range(n_neighbours_2)]
    num_items ← length of duplicated_test_data
    for item_pos in range(num_items) do
        combined_list ← []
        for pair in zip(differences_test_X[item_pos], duplicated_test_data[item_pos]) do
            for item in pair do
                Append item to combined_list
            end for
        end for
        Append np.array(combined_list) to combined_differences_test_X
    end for
    return combined_differences_test_X
end function
```

3.5.2 Variation 2: Duplication Based on Distance Design and Implementation (LRV2, LCV2)

Objective

A novel variation introduced in this study involves duplicating differences based on distance, aiming to amplify the influence of neighbors closely resembling the base case. This innovative approach represents a departure from conventional methods in the learning from differences paradigm. By magnifying the impact of differences that exhibit stronger correlation with the base case, this technique effectively mitigates the influence of outliers, leading to more robust model performance.

This novel approach is specifically integrated into the fitting process of both the `LingerClassifier` and `LingerRegressor` models. By emphasizing the significance of closely resembling neighbors, this technique enhances the model's ability to discern meaningful patterns in the data, thereby improving its predictive accuracy and generalization capabilities.

1. **Normalize Distances**: Normalize the distances from the base case to its nearest neighbors represented by $Distances_{ij}$. We apply a logarithmic transformation to the distances: $Distances_{ij} = \log_e(Distances_{ij} + 1)$, with the addition of 1 to prevent a logarithm of 0.

2. **Determine Maximum and Minimum Distances**: Determine the maximum distance $MaxDistance$ and minimum distance $MinDistance$ in $Distances_{ij}$.

3. **Compress Distances**: Compress the distances within $Distances_{ij}$ to a range between 0 and 10 using the formula:

$$Distances_{ij} = \text{round}\left(10 - \frac{(Distances_{ij} - MinDistance)}{(MaxDistance - MinDistance) \times 10}\right)$$

4. **Duplicate Items**: Duplicate each item ΔD_{ij}^{train} by the corresponding distance in $Distances_{ij}$. It's important to note that this duplication process can significantly increase the size of the dataset and, consequently, the training time overhead. The process of duplication is only used in the training process of the algorithm.

Training Design

Algorithm 25 Duplication Based on Distance in `LingerClassifier` and `LingerRegressor`

Input: Distances $Distances_{ij}$, Base dataset ΔD^{train}

Normalize Distances:
for each distance $Distances_{ij}$ **do**
    $Distances_{ij} \leftarrow \log_e(Distances_{ij} + 1)$ ▷ Apply logarithmic transformation
end for
Determine Maximum and Minimum Distances:
$MaxDistance \leftarrow$ maximum value of $Distances_{ij}$
$MinDistance \leftarrow$ minimum value of $Distances_{ij}$
Compress Distances:
for each distance $Distances_{ij}$ **do**
    $Distances_{ij} \leftarrow \text{round}\left(10 - \frac{(Distances_{ij} - MinDistance)}{(MaxDistance - MinDistance) \times 10}\right)$
end for
Duplicate Items:
for each item ΔD_{ij}^{train} in ΔD^{train} **do**
    Duplicate ΔD_{ij}^{train} by the corresponding distance in $Distances_{ij}$
end for

Training Implementation

Algorithm 26 Difference Duplication Based on Distance (Part 1)

```
function duplicate_difference_based_on_distance(differences_X, differences_y, distances)
    distances_X ← []
    for d in distances do
        neighbors ← d[1 :]
        for n in neighbors do
            append n to distances_X
        end for
    end for
    distances_array ← np.array(distances_X)
    log_values ← np.log(distances_array + 1)
    min_val ← np.min(log_values)
    max_val ← np.max(log_values)
    if min_val == max_val then
        scaled_distances_rounded ← np.ones_like(log_values).astype(int)
    else
        scaled_distances ← 10 − ((log_values − min_val)/(max_val − min_val) × 10)
        scaled_distances_rounded ← np.round(scaled_distances).astype(int)
    end if
    Algorithm continues
end function
```

Algorithm 27 Difference Duplication Based on Distance (Part 2)

```
1: function DUPLICATE_DIFF_CONTD(filtered_data_X, filtered_data_y,
   scaled_distances_rounded)
2:     duplicated_list_X ← []
3:     duplicated_list_y ← []
4:     for each item, count in zip(filtered_data_X, scaled_distances_rounded) do
5:         for _ ← 1 to count do
6:             append item to duplicated_list_X
7:         end for
8:     end for
9:     for each item, count in zip(filtered_data_y, scaled_distances_rounded) do
10:        for _ ← 1 to count do
11:            append item to duplicated_list_y
12:        end for
13:    end for
14:    return duplicated_list_X, duplicated_list_y
15: end function
```

3.5.3 Variation 3: Addition of Distance Column Design and Implementation (LRV3, LCV3)

Objective

Variation 3 introduces a novel approach by incorporating the distance between each base case and its nearest neighbor into both the training and prediction phases of the neural network. This innovative method aims to leverage the proximity information to enhance the learning process and predictive performance of the model.

By integrating distance information, the neural network gains insight into the spatial relationships between data points. This allows the model to potentially learn associations between differences and their corresponding distances, leading to improved accuracy and effectiveness in both training and prediction tasks.

The motivation behind this variation lies in its potential to provide the model with a deeper understanding of the underlying data structure. By considering distance as a factor in the learning process, the model can better capture nuanced patterns and relationships, resulting in more robust and reliable predictions.

Design for Training

In the fit phase, the distances between each base case D_i^{train} and its nearest neighbor in D^{train} are computed and stored in $Distances_{ij}$. To incorporate these distances into the training data, each distance in $Distances_{ij}$ is concatenated with the corresponding difference ΔD^{train}. This results in a new dataset where each difference is paired with its corresponding distance, ensuring a one-to-one ratio between distances and differences.

Algorithm 28 Training Algorithm for Learning from Differences, with Distance Column

Input: Dataset D, Neural network model, Number of neighbors n, Number of epochs E

Split D into training set D^{train} and test set D^{test} with an 80%-20% split
Initialize ΔD^{train} and $Distances$ as empty lists
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta D_{ij}^{train} = (X_i^{train} - X_j^{train}, y_i^{train} - y_j^{train})$
        Compute the distance: $Distances_{ij} = ||X_i^{train}, X_j^{train}||$ (Euclidean distance)
        Concatenate $Distances_{ij}$ as a new attribute to ΔD_{ij}^{train}
    end for
end for
Train the MLPRegressor neural network model using ΔD^{train} over E epochs
return Trained neural network model $TrainedNN()$.

Design for Predict

In the prediction phase, distances between each test case X_j^{test} and its nearest neighbors in D^{train} are stored in $Distances_{ji}$. These distances are concatenated with the corresponding differences ΔX^{test}, ensuring a one-to-one ratio between distances and differences in the prediction dataset.

Algorithm 29 Prediction Algorithm for Learning from Differences with Additional Distance Column

Input: Test dataset D^{test}, Trained neural network model, Number of neighbors n, $TrainedNN()$

Initialize ΔD^{train} and $Distances$ as empty lists
for each case $X_j^{test} \in D^{test}$ **do**
    Retrieve n similar cases from D^{train} using nearest neighbors
    for each similar case $(X_i^{train}, y_i^{train}) \in D^{train}$ **do**
        Compute the differences: $\Delta X_{ij}^{test} = (X_j^{test} - X_i^{train})$
        Compute the distance: $Distances_{ji} = ||X_j^{test}, X_i^{train}||$ (Euclidean distance)
        Add $Distances_{ji}$ as a new attribute to ΔX_{ji}^{test}
        Use the trained neural network to predict: $TrainedNN(\Delta X_{ji}^{test}) = \Delta X_{ji}^{pred}$
        Adapt neighbor's y values: $y_j^{pred} = y_i^{train} + \Delta X_{ji}^{pred}$
    end for
    if Regression **then**
        Average the adapted y values: $y_j^{pred} = \frac{1}{n} \sum_{i=1}^{n} y_j^{pred}$
    else if Classification **then**
        Perform majority voting on y_j^{pred} to obtain y_j^{pred}
    end if
    return y^{pred}
end for

Implementation for Training, and Prediction

The same function is called during both predict and training.

Algorithm 30 Add Additional Distance Column

Input: Differences dataset $differences\_X$, Distances of the nearest neighbors $distances$

1: **function** $add\_additional\_distance\_column(differences\_X, distances)$
2: Adds a column with distance values associated with each $differences\_X$ during fitting.
3: **Output:** differences_X with an additional column.
4: **for** i **in** range($len(distances)$) **do**
5: $diff\_res \leftarrow$ Convert $distances[i]$ to numpy array
6: Append $diff\_res$ to $differences\_X[i]$
7: **end for**
8: **return** differences_X
9: **end function**

3.6 Acronym Guide

Table 3.7: Variations for `LingerRegressor` and `LingerClassifier`

Variation	Acronym
Addition of Context	LRV1, LCV1
Duplication Based on Distance	LRV2, LCV2
Addition of Distance Column	LRV3, LCV3

3.7 Learning From differences Implicitly

3.7.1 Introduction and Motivation

This design segment aims to implement and extend the algorithms proposed by Ye et al. [18] for both classification and regression tasks. The primary objective is to develop a unified framework where a single set of algorithms can be employed for both classification and regression tasks, leveraging implicit learning from differences within case pairs.

As outlined in the literature review (see Section 2), the algorithms (Algorithm 9, Algorithm 10) proposed by Ye et al. were chosen as the basis for our model

implementation. Variant 5, as demonstrated in the paper, exhibited superior performance among the variants explored, prompting its adoption in our implementation.

Moreover, while the paper focuses on classification tasks, it overlooks the potential application of implicit difference learning within case pairs for regression tasks. Thus, our secondary motivation is to explore the adaptability of Variant 5 to regression models. We seek to investigate whether the framework outlined in Algorithm 9 and Algorithm 10 can be extended and optimized for regression tasks effectively.

By extending the application of Variant 5 to regression tasks, we aim to provide a comprehensive and versatile framework for both classification and regression tasks, harnessing the power of implicit learning from differences within case pairs.

3.8 Linger Implicit Classification Model: `LingerImplicitClassifier` (LIC)

3.8.1 Objective

The objective of the Linger Implicit Classification Base Model (`LingerImplicitClassifier`) is to implement the learning from differences methodology seen in [18] for classification tasks. The primary goals include:

- Developing a versatile classification model that learns through implicit differences in case pairs.
- Ensuring compatibility with Scikit-learn for seamless integration and extensive hyperparameter testing.
- Providing interpretable predictions to facilitate user understanding, for a variety of tasks.
- Implementing enhancements to the learning from differences algorithm for improved classification performance through hyperparameter testing.
- Designing a modular and extensible codebase to support future variations and enhancements.

The `LingerImplicitClassifier` is tailored for classification problems where interpretability and adaptability are critical, Where explicit differences may not be easily calculable. It adds another layer of interpretability, as both the target case and retrieved case are visible within the case pair which the Neural Nets are trained on.

3.8.2 Architecture

The `LingerImplicitClassifier` leverages a hybrid architecture incorporating K-nearest neighbors (KNN) and Scikit-learn's `MLPClassifier` to tackle classification tasks. In the training phase, KNN plays a pivotal role in computing nearest neighbors, thereby facilitating the creation of case pairs. During the prediction phase, KNN is again utilized to retrieve the most similar cases from each class.

The process of generating case pairs for predictions involves KNN's functionality to identify the nearest neighbors. These neighbors serve as crucial elements in forming case pairs, enabling the classifier to make informed predictions based on similarities observed within the dataset.

The integration of KNN with the `MLPClassifier` offers a comprehensive approach to classification tasks, combining the strengths of both algorithms. While KNN excels in identifying similarities and forming case pairs, the `MLPClassifier` contributes by providing a powerful neural network-based classification model capable of learning intricate patterns and relationships within the data.

By harnessing the complementary capabilities of KNN and the `MLPClassifier`, the `LingerImplicitClassifier` aims to achieve robust and accurate classification results across diverse datasets, making it a versatile tool for various machine learning applications.

3.8.3 Components

The key components of the `LingerClassifier` include:

- KNN Module: Computes nearest neighbors and extracts differences during training and prediction phases.
- `MLPClassifier`: Offers flexibility in activation functions, solvers, and hyperparameters for classification.

- Additional Features: Class to class pairs, random N selected neighbors, and N nearest neighbors.

3.8.4 Parameterization

The model provides extensive parameterization options for fine-tuning KNN, `MLPClassifier`, and additional features. Hyperparameters for the classification model mirror those available in standard KNN and neural networks, with additional parameters specific to `LingerClassifier`.

3.8.5 Training and Prediction Design

The training and prediction algorithms for the `LingerImplicitClassifier`.

Training Algorithm Design

Algorithm 31 Training Algorithm for `LingerImplicitClassifier`

Input: Dataset D, number of neighbors 1

Split D into training set D^{train} and test set D^{test} with an 80%-20% split
Create $CasePairs()$ dictionary.
Create $AdaptNN()$, a dictionary to store each Neural Network
List of unique Classes $UniqueClasses$
for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**
    Retrieve 1 similar case from D^{train} using nearest neighbors.
    for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**
        $CasePairs(y_j^{train}).append([X_j^{train} : X_i^{train}, y_i^{train}])$
    end for
end for
for each unique $Class$ in $UniqueClasses$ **do**
    Create a new Classification Neural Network, trained on Case Pairs
    $AdaptNN(Class).fit(CasePairs(Class))$
end for
return Trained neural network models dictionary $AdaptNN()$.

Prediction Algorithm Design

During the prediction phase, various methods exist for retrieving neighbors within each class. These methods can be categorized into three variants:

1. **Variant 1:** (LICV1) Selecting N random items from each class.
2. **Variant 2:** (LICV2) Selecting the single nearest neighbor from each class.

Each variant may be suitable depending on the specific characteristics of the dataset and the objectives of the prediction task.

Algorithm 32 Prediction Algorithm for `LingerImplicitClassifier`

Input: Dataset D^{test}, Dictionary $AdaptNN()$, Set $UniqueClasses$, and Dictionary $CasePairs()$

```
Initialize an empty list y^pred
for each case X_j^test ∈ D^test do
    Create an empty list preds
    for each unique class Class in UniqueClasses do
        if Base version: (LIC) then
            Retrieve N nearest neighbors from CasePairs(Class)
        else if variant 1: (LICV1) then
            Select N random items from CasePairs(Class)
        else if variant 2: (LICV2) then
            Select the single nearest neighbor from CasePairs(Class)
        end if
        for each selected case (X_i^CasePair, y_i^CasePair) do
            for each (X_z^train, X_q^train) ∈ X_i^CasePair do
                preds.append(AdaptNN(Class).predict(X_z^train : X_j^test))
            end for
        end for
        Perform a majority vote on preds
        y_j^pred = MajorityVote(preds)
    end for
end for
return y^pred
```

3.8.6 Training and Prediction Implementation

Training Algorithm Implementation

The `fit` function of the `LingerImplicitClassifier` model trains the classifier using the provided training data X and target labels y. It begins by handling input data, converting sparse matrices to dense arrays if necessary. Next, it initializes structures to store trained adaptation neural networks and unique classes. The algorithm then computes the nearest neighbors for each sample in X, generates pairs of cases (consisting of base and retrieved cases), and organizes the data based on class labels. Finally, it trains adaptation neural networks for each class label using the generated case pairs and target labels. Overall, this process enables the `LingerImplicitClassifier` model to adapt to specific instances retrieved during inference, enhancing its performance on the task.

Algorithm 33 Fit Function for LingerImplicitClassifier

Input: Training data X, target labels y

function FIT(X, y)
    $is\_sparse\_X \leftarrow$ Check if X is sparse
    $is\_sparse\_y \leftarrow$ Check if y is sparse
    if is_sparse_X **then**
        Convert X to dense array
    end if
    if is_sparse_y **then**
        Convert y to dense array
    end if
    $classes\_ \leftarrow$ Get unique classes from y
    $adaptation\_networks\_ \leftarrow \{\}$
    $neighbours \leftarrow$ Fit Nearest Neighbors with X
    $distances, indices \leftarrow$ Compute nearest neighbors for each sample in X
    Initialize $case\_pairs\_X$ and $case\_pairs\_y$
    for each i in $indices$ **do**
        $base \leftarrow i[0]$
        $neighbors \leftarrow i[1:]$
        for each n in $neighbors$ **do**
            $case\_x \leftarrow [X[base], X[n]]$
            $case\_y \leftarrow [y[base], y[n]]$
            Append $case\_x$ to $case\_pairs\_X$
            Append $case\_y$ to $case\_pairs\_y$
        end for
    end for
    Initialize dictionaries: $case\_pairs\_by\_class$, $cases\_without\_concatenation$, $case\_pairs\_y$
    for each $(source, target)$ in $zip(case\_pairs\_X, case\_pairs\_y)$ **do**
        Append $source[0]$ to $cases\_without\_concatenation[target[1]]$
        $problem\_query \leftarrow$ Concatenate $source[0]$ and $source[1]$
        Append $problem\_query$ to $case\_pairs\_by\_class[target[1]]$
        Append $target[0]$ to $case\_pairs\_y[target[1]]$
    end for
    for each $class\_label, cases$ in $case\_pairs\_by\_class$ **do**
        Initialize MLPClassifier with parameters
        Fit MLPClassifier with $cases$ and $case\_pairs\_y[class\_label]$
        Store the fitted MLPClassifier in $adaptation\_networks\_[class\_label]$
    end for
end function

Prediction Algorithm Implementation

The `predict` function of the `LingerImplicitClassifier` model predicts class labels for the input samples. It creates nearest neighbor finders for each class, then iterates through the test samples to predict class labels. Depending on the chosen variant, it either selects nearest neighbors using (LICV1) or a single nearest neighbor using (LICV2). Finally, it performs majority voting to decide the final classification.

Algorithm 34 Predict Function for LingerImplicitClassifier

Input: Test data X

function PREDICT(X)
    $y^{pred} \leftarrow []vector$
    $is\_sparse\_X \leftarrow$ Check if X is sparse
    if is_sparse_X **then**
        Convert X to dense array
    end if
    $nbrs\_by\_class \leftarrow$ Create nearest neighbor finders for each class
    for each *sample* in X **do**
        if *variant* == 2 (LICV2) **then**
            class_predictions ← Predict class labels for *sample* using 1 `NearestNeighbor`
        else
            class_predictions ← Predict class labels for *sample* using N `NearestNeighbors`
        end if
        majority_prediction ← Perform majority voting on *class_predictions*
        Append *majority_prediction* to y^{pred}
    end for
    return y^{pred}
end function

Algorithm 35 Helper: Create Nearest Neighbor Finders

function _CREATE_NBRS_BY_CLASS
    $nbrs\_by\_class \leftarrow \{\}$
    for each $class\_label$ in $classes\_$ **do**
        $n\_neighbors \leftarrow 1$ if $variant == 2$ (LICV2) else $n\_neighbours\_2$
        $nbrs \leftarrow$ `NearestNeighbors`$(n\_neighbors).fit(cases\_without\_concatenation[class\_label])$
        $nbrs\_by\_class[class\_label] \leftarrow nbrs$
    end for
    return $nbrs\_by\_class$
end function

Algorithm 36 Helper: Predict Class Labels

function $\_predict\_class(sample, nbrs\_by\_class)$
    $class\_predictions \leftarrow$ {class_label : [] for class_label in $classes\_$}
    for each $class\_label$, $nbrs$ in $nbrs\_by\_class.items()$ **do**
        if $variant == 1$ (LICV1) **then**
            $indices \leftarrow$ Randomly select $n\_neighbours\_2$ indices from $range$(len($cases\_without\_concatenation[class\_label]$))
        else
            $distances, indices \leftarrow nbrs.kneighbors([sample])$
        end if
        $nearest\_neighbors \leftarrow indices[0]$
        for each $neighbor\_index$ in $nearest\_neighbors$ **do**
            $neighbor\_sample \leftarrow cases\_without\_concatenation[class\_label][neighbor\_index]$
            $merged\_case\_pair \leftarrow$ Concatenate $neighbor\_sample$ and $sample$
            $adapted\_prediction \leftarrow$ Predict with $adaptation\_networks\_[class\_label]$ using $merged\_case\_pair$
            Append $adapted\_prediction[0]$ to $class\_predictions[class\_label]$
        end for
    end for
    return $class\_predictions$
end function

Algorithm 37 Helper: Perform Majority Voting

```
function _majority_vote(class_predictions)
    all_predictions ← [prediction for predictions in class_predictions.values() for prediction in p
    prediction_counter ← Counter(all_predictions)
    most_common_predictions ← prediction_counter.most_common(1)
    If the case occurs where there is no distinct most common.
    if len(most_common_predictions) > 1 then
        majority_prediction ← Randomly select from most_common_predictions
    else
        majority_prediction ← most_common_predictions
    end if
    return majority_prediction
end function
```

3.8.7 Conclusion

The `LingerImplicitClassifier` offers a novel approach to classification tasks by integrating the learning from differences implicitly through case pairs. It performs well for a variety of classification tasks. That could benefit from a more interpretable learning methodology

3.9 Linger Implicit Regression Model: `LingerImplicitRegressor`

3.9.1 Objective

The objective of the `LingerImplicitRegressor` is to pioneer a learning methodology seen in [18] tailored specifically for regression tasks, known as Learning from Differences. The primary objectives encompass:

- Creation of a flexible regression model capable of learning implicitly from differences observed in pairs of cases.
- Adaptation of regression tasks into the Learning from Differences framework to unlock its potential.

- Seamless integration with Scikit-learn to ensure compatibility and enable extensive hyperparameter optimization.
- Provision of interpretable predictions crucial for user comprehension across a wide spectrum of regression scenarios.
- Continuous enhancement of the Learning from Differences algorithm through rigorous hyperparameter testing to achieve superior regression performance.
- Designing a modular and extensible codebase to accommodate future variations and improvements.

The `LingerImplicitRegressor` is designed to address regression challenges where explicit differences may be elusive, prioritizing interpretability and adaptability. It implements the classification methodology described in [18] but facilitates its use for regression tasks. This is a novel approach. It provides added transparency by exposing both the target case and the retrieved case within the case pair, facilitating a deeper understanding during model training.

3.9.2 Architecture

The `LingerImplicitRegressor` utilizes a hybrid architecture integrating K-nearest neighbors (KNN) with Scikit-learn's `MLPClassifier`. In the training phase, KNN plays a pivotal role in computing nearest neighbors, thereby facilitating the creation of case pairs. During the prediction phase, KNN is again utilized to retrieve the most similar cases from each class.

The process of generating case pairs for predictions involves KNN's functionality to identify the nearest neighbors. These neighbors serve as crucial elements in forming case pairs, enabling the regressor to make informed predictions based on similarities observed within the dataset.

The integration of KNN with the `MLPClassifier` offers an interesting approach to regression tasks, combining the strengths of both algorithms. It places regression values within specified ranges, offering a new type of solution to classic regression issues. While KNN excels in identifying similarities and forming case pairs, the `MLPClassifier` contributes by providing a powerful neural network-based classification model capable of learning intricate patterns and relationships within the data. By placing continuous values, into classes, we essentially construct a classification problem out of a regression issue.

By harnessing the complementary capabilities of KNN and the `MLPClassifier`, the `LingerImplicitRegressor` aims to achieve robust and accurate regression results across diverse datasets while using a classifier, making it a versatile tool for various machine learning applications. This new approach to regression tasks that allows for categorization of regression results. It processes a regression task and converts it to a classification task, this could be useful for problems where ranges of values are highly important. This would make certain regression tasks more interpretable.

3.9.3 Components

The key components of the `LingerRegressor` include:

- KNN Module: Computes nearest neighbors and extracts differences during training and prediction phases.
- `MLPClassifier`: Offers flexibility in activation functions, solvers, and hyperparameters for classification.
- Additional Features: Class to class pairs, random N selected neighbors, and N nearest neighbors.

3.9.4 Parameterization

The model provides extensive parameterization options for fine-tuning KNN, `MLPClassifier`, and additional features. Hyperparameters for the classifier model mirror those available in standard KNN and neural networks, with additional parameters specific to `LingerRegressor`.

3.9.5 Training and Prediction Design

The training and prediction algorithms for the `LingerImplicitRegressor`.

Training Algorithm Design

Algorithm 38 Training Algorithm for `LingerImplicitRegressor`

Input: Dataset D, number of neighbors 1, N Classes

Split D into training set D^{train} and test set D^{test} with an 80%-20% split

Create $CasePairs()$ dictionary.

Create $AdaptNN()$, a dictionary to store each Neural Network

Convert training data y values into classes, using the algorithm 39

$D^{train}y$, $unique\_ranges$, $y_{\text{train}}^{\max}$, $y_{\text{train}}^{\min} = RegToClassConverterFunction(D^{train}, N)$

Convert test y values based on the train y value range. So that a new range is the same for both.

$D^{test}y$, $unique\_ranges$, $y_{\text{test}}^{\max}$, $y_{\text{test}}^{\min} = RegToClassConverterFunction(D^{train}, N, y_{\text{train}}^{\max}, y_{\text{train}}^{\min})$

List of unique Classes $UniqueClasses$

for each case $C_i = (X_i^{train}, y_i^{train}) \in D^{train}$ **do**

  Retrieve 1 similar case from D^{train} using nearest neighbors.

  for each similar case $(X_j^{train}, y_j^{train}) \in D^{train}$ **do**

    $CasePairs(y_j^{train}).append([X_j^{train} : X_i^{train}, y_i^{train}])$

  end for

end for

for each unique $Class$ in $UniqueClasses$ **do**

  Create a new Classification Neural Network, trained on Case Pairs

  $AdaptNN(Class).fit(CasePairs(Class))$

end for

return Trained neural network models dictionary $AdaptNN()$.

Prediction Algorithm Design

In the prediction phase for regression, several methods exist for retrieving neighbors within each class. These methods can be categorized into three variants:

1. **Variant 1:** (LIRV1) Selecting N random items from each class.
2. **Variant 2:** (LIRV2) Selecting the single nearest neighbor from each class.

The prediction algorithm mirrors Algorithm 32. The key distinction lies in the preprocessing step, where continuous y values are converted into classes to enable accuracy testing.

3.9.6 Training and Prediction Implementation

Both the LingerImplicitClassifier and LingerImplicitRegressor share the same training and prediction implementations. This shared functionality is encapsulated in a base class, allowing for code reuse and maintenance. The algorithm 39 found in the 3.9.7 section is used during the data preprocessing stage in the same way as `sklearn LabelEncoder`. To convert the regression problem into a classification problem.

Training Implementation

The training algorithm for `LingerImplicitRegressor` is outlined in 3.8.6. The function 3.9.7 is called before training to convert y values into class labels for both training and test datasets

Prediction Implementation

The prediction algorithm for `LingerImplicitRegressor` is outlined in 3.8.6.

3.9.7 Regression to Classification Converter Design and Implementation

(RTCC) This function serves to categorize continuous y target values for training, validation, and testing. It returns a human-readable range with their corresponding

numeric prediction. This ensures that the results are interpretable to humans at the end of the process. Central to its operation is the parameter equal division, which indicates whether samples should be equally distributed into N classes. This feature enables uniform representation across different segments of the target variable, promoting fairness and consistency in the categorization process.

Regression to Classification Converter Design

The "Regression to Classification Converter" algorithm (Algorithm 39) is designed to transform continuous regression target values into categorical classes. It takes as input a dataset D, the desired number of segments N, and optional parameters such as $y_{\max}$, $y_{\min}$, and Equal division.

If Equal division is set, the algorithm divides the data into N segments based on nearly equal counts. Otherwise, it calculates the range for each segment based on the provided or inferred maximum and minimum values of y. It then iterates through each value in y, determining the corresponding category based on the segment ranges. The algorithm returns the transformed y values, along with the unique ranges, minimum and maximum values of y, providing a categorical representation suitable for classification tasks.

Algorithm 39 Converter Algorithm: Regression to Classification (Part 1)

Input: Dataset D, number of segments N, optional: $y_{\max}$, $y_{\min}$, Equal division

$(X, y) \in D$
Get the minimum and maximum values of the labels
Ensure that the max and min are used from the test set so that y values are classified the same in both fit and predict.
if Equal division **then**
    Divide the data into N segments based on nearly equal counts
    $quantiles \leftarrow$ np.linspace$(0, 1, N + 1)$
    if $unique\_ranges$ is None **then**
        $unique\_ranges \leftarrow$ np.quantile$(y, quantiles)$
    end if
    $categories \leftarrow$ np.digitize$(y, unique_ranges[: -1])$
    $category\_counts \leftarrow$ Counter$(categories)$
    Return $categories$, $unique\_ranges$, $y_{\min}$, $y_{\max}$
end if

Algorithm 40 Converter Algorithm: Regression to Classification (Part 2)

Input: Continuation from Part 1

if $y_{\max}$ and $y_{\min}$ are provided **then**
    Leave $y_{\max}$ and $y_{\min}$ as is
else
    $y_{\max} \leftarrow \text{MAX}(y)$
    $y_{\min} \leftarrow \text{MIN}(y)$
end if
Calculate the range for each segment for y values
segmentRange $\leftarrow \frac{y_{\max} - y_{\min}}{N}$
Initialize an empty list: *categories*
Initialize an empty dictionary: *unique_ranges*
current_index $\leftarrow 0$
for each value *val* in y **do**
    for do$i$ in range(N)
        if $i < N - 1$ **then**
            if $y_{\min} + i \times \text{segmentRange} \leq \text{val} < y_{\min} + (i+1) \times \text{segmentRange}$ **then**
                category $\leftarrow (y_{\min} + i \times \text{segmentRange}) - (y_{\min} + (i+1) \times \text{segmentRange})$
                if category not in *unique_ranges* **then**
                    unique_ranges[*category*] $\leftarrow$ *current_index*
                    current_index $\leftarrow$ *current_index* $+ 1$
                end if
                Append *unique_ranges*[*category*] to *categories*
                break
            end if
        else
            if $y_{\min} + i \times \text{segmentRange} \leq \text{val} \leq y_{\max}$ **then**
                category $\leftarrow (y_{\min} + i \times \text{segmentRange}) - (y_{\max})$
                if category not in *unique_ranges* **then**
                    unique_ranges[*category*] $\leftarrow$ *current_index*
                    current_index $\leftarrow$ *current_index* $+ 1$
                end if
                Append *unique_ranges*[*category*] to *categories*
                break
            end if
        end if
    end for
end for
Converted y values: $y \leftarrow$ *categories*
Return y, *unique_ranges*, $y_{\min}$, $y_{\max}$

Regression to Classification Converter Implementation

This function is encapsulated within a class called `RegressionToClassificationConverter`. The `transform` function carries out the conversion of regression values to classification values.

Algorithm 41 Transform function Initialised

function TRANSFORMINIT($y, min\_val = None, max\_val = None, unique\_ranges = None$)
    Input:
    y (list or array-like): The continuous target variable.
    min_val (float): Minimum value of the target variable.
    max_val (float): Maximum value of the target variable.
    $unique\_ranges$ (array-like): Predefined unique ranges for segments.
    Output:
    $categories$ (list): A list of categories (segment indices) corresponding to each value in y.
    $unique\_ranges$ (dict or array-like): Unique ranges for segments.
    min_val (float): Minimum value of the target variable.
    max_val (float): Maximum value of the target variable.
    if min_val is None and max_val is None **then**
        $min\_val \leftarrow \min(y)$
        $max\_val \leftarrow \max(y)$
    end if
end function

Algorithm 42 Transform function (Part 1)

```
1: All variables from transformInit() are available.
2: function TRANSFORMPARTONE(y, min_val = None, max_val = None, unique_ranges = None)
3:     if equal_division is None then
4:         segment_range ← (max_val − min_val)/n_segments
5:         categories ← []
6:         unique_ranges ← {}
7:         current_index ← 0
8:         for val in y do
9:             for i in range(n_segments) do
10:                if i < n_segments − 1 then
11:                    if min_val + i × segment_range ≤ val < min_val + (i + 1) × segment_range then
12:                        category ← int(min_val + i × segment_range)
13:                        int((min_val + (i + 1)) × segment_range)
14:                        if category not in unique_ranges then
15:                            unique_ranges[category] ← current_index
16:                            current_index ← current_index + 1
17:                        end if
18:                        categories.append(unique_ranges[category])
19:                        break
20:                    end if
21:                else
22:                    if min_val + i × segment_range ≤ val ≤ max_val then
23:                        category ← int(min_val + i × segment_range)
24:                        int(max_val)
25:                        if category not in unique_ranges then
26:                            unique_ranges[category] ← current_index
27:                            current_index ← current_index + 1
28:                        end if
29:                        categories.append(unique_ranges[category])
30:                        break
31:                    end if
32:                end if
33:            end for
34:        end for
35:        return categories, unique_ranges, min_val, max_val
36:    else
37:        Call transformPartTwo() function
38:    end if
39: end function
```

Algorithm 43 Transform Function (Part 2): Equal Division

```
function transformPartTwo(y, min_val, max_val, unique_ranges, n_segments)
    quantiles ← np.linspace(0, 1, n_segments + 1)
    if unique_ranges is None then
        unique_ranges ← np.quantile(y, quantiles)
    end if
    categories ← np.digitize(y, unique_ranges[: −1])
    return categories, unique_ranges, min_val, max_val
end function
```

3.9.8 Conclusion

The `LingerImplicitClassifier` offers a novel approach to regression tasks by integrating the learning from differences implicitly through case pairs. It performs well for a variety of regression tasks, while utilizing a learning from differences classification model. The number of segments greatly affects the outcome of this model. One of the limitations observed the `LingerImplicitClassifier` is its sensitivity to noise, particularly when labeling values based on the maximum and minimum. However, opting to evenly distribute samples into classes can mitigate this sensitivity.

3.10 Learning from Differences For Images

3.10.1 Introduction and Motivation

The introduction of the `LingerImageClassifier` and `LingerImageRegressor` represents a novel approach to image classification and regression tasks. Motivated by the innovative methodology outlined in "Learning from Differences 2022," these tools are specifically designed to enhance interpretability and explainability in processing image data. The aim was to ensure these models not only integrate smoothly into the existing software framework but also support the customization of Keras convolutional neural networks. This strategy diverges from traditional methods, such as those employed by Scikit-learn's `MLPClassifier` and `MLPRegressor`, offering a fresh perspective on addressing image-centric challenges.

3.11 Linger Image Classification Model `LingerImageClassifier` (LIMC)

3.11.1 Objective

The motivation behind the development of (LIMC) stems from the growing demand for more sophisticated image classification tools capable of handling the complexities and nuances of real-world data. Traditional models often fall short in terms of explainability, and transparency. (LIMC) aims to address these gaps by:

- Incorporating state-of-the-art convolutional neural network (CNN) designs that

improve upon the limitations of existing models in capturing intricate patterns in image data.

- Utilizing a novel training methodology seen in [7] that enhances model learning from image differences, inspired by recent research findings.
- Aiming to demystify the black-box nature of deep neural networks by providing insights into the decision-making process, making the model more trustworthy and applicable in sensitive areas such as medical diagnosis and autonomous driving.

3.11.2 Architecture

The `LingerImageClassifier` employs a sophisticated, modular architecture designed to leverage the nuances of image data, enhancing both predictive accuracy and interpretability. This architecture consists of three integral components:

- **Custom Fit Method:** Developed to integrate with the scikit-learn ecosystem, this Fit method is central to the classifier's functionality. It generates a unique dataset of image differences, effectively capturing variations between image pairs. This method not only emphasizes the importance of contrastive learning but also enriches the model's understanding of nuanced features within the data, such as changes in texture, lighting, and spatial relationships. The process involves computing the differential aspects of images, focusing on attributes that significantly influence the y label values, thereby enabling a more focused and effective learning strategy. It also guarantees that we know what it is precisely learning from, its visual difference to its neighbor.
- **Convolutional Neural Network (CNN):** At the heart of the classifier lies a customizable Convolutional Neural Network, tailored from Keras's comprehensive library. This CNN is specifically designed to process and predict the differences highlighted by the custom Fit method. The architecture features a combination of convolutional layers, activation functions, and pooling layers, strategically organized to maximize feature extraction and minimize information loss. Special attention is given to the network's depth, filter sizes, and the incorporation of advanced techniques such as batch normalization and dropout for regularization. This careful construction ensures optimal performance in discerning subtle differences in image data, thus significantly contributing to the precision of subsequent predictions.

- **Custom Predict Method:** Complementing the custom Fit method, the Predict function stands as a testament to the model's innovative approach to classification. This component takes the CNN's predictions on image differences and synthesizes them with the original data to formulate the final classification outcomes. The method employs an algorithm that accurately applies predicted differences to their corresponding instance, accounting for the model's understanding of contrastive features within the dataset. This not only enhances the model's accuracy but also its adaptability to various image classification scenarios, ensuring that the final predictions are both precise and contextually grounded.

Integration and Workflow: The synergy between these components forms the cornerstone of the `LingerImageClassifier`'s architecture. Initially, the custom Fit method preprocesses the input images to create a differential dataset, which is then fed into the CNN for feature extraction and difference prediction. The custom Predict method subsequently interprets these predictions, applying them to derive the final classification results. This integrated workflow ensures a coherent and efficient processing pipeline, from initial data preparation to final outcome generation.

Innovations and Contributions: The architecture of the `LingerImageClassifier` introduces several key innovations to the field of image classification. By focusing on image differences, the model uncovers deeper insights into the data, facilitating a more nuanced understanding and interpretation of the factors driving classification decisions. This approach not only improves model performance across diverse datasets but also enhances the explainability of its predictions, addressing a critical need within the domain of deep learning. Visually we could Identify exactly what the image is learning from based on the difference image generated.

3.11.3 Components

The key components of the `LingerImageClassifier` include:

- KNN Module: Computes nearest neighbors used to extract differences during training phase and prediction phase.
-
- **Convolutional Neural Network (CNN):** The heart of the classification process is a Convolutional Neural Network (CNN), meticulously designed to analyze and interpret the differences identified by the KNN module. This CNN is

notable for its flexibility, allowing for the customization of activation functions, solvers, and an extensive range of hyperparameters to optimize classification performance. The adaptability of the CNN component ensures that the model can be fine-tuned to suit various datasets and classification challenges. It excels in processing the contrastive features extracted by the KNN module, effectively learning from these differences to make accurate predictions. The CNN's architecture, which can include layers specifically tailored to enhance feature extraction and reduce overfitting, such as dropout and batch normalization layers, is instrumental in capturing the complex patterns inherent in image data.

3.11.4 Parameterization

The `LingerImageClassifier` is designed with flexibility and adaptability at its core, offering a comprehensive suite of parameters that can be fine-tuned to optimize performance across a wide range of image classification tasks. The use can vary the design of the (CNN) convolutional Neural Network, as well as vary the number of neighbors used in the fit and prediction methods.

3.11.5 Training and Prediction Design

Training Design

The Image Classification Training Algorithm outlines the steps involved in training a convolutional neural network (CNN) for image classification tasks using the differences between nearest neighbor images.

Algorithm 44 Image Classification Training Algorithm

function TrainImageClassifier(X, y)
  Input: Dataset of images X, Target labels y
  Output: Trained CNN model
  Generate the image difference datasets.
  $differences\_X, differences\_y \leftarrow$ GenerateDifferences(X, y)
  Initialize (CNN) model.
  Train CNN model using $differences\_X$ and $differences\_y$
  return Trained CNN model
end function

The ‘GenerateDifferences‘ 45 algorithm encapsulates the process of computing differences between nearest neighbor images and their base image, preparing the data for training the k-nearest neighbors classifier. It iterates through each image in the training dataset, identifies its nearest neighbors, and calculates the absolute differences between their pixel values. These differences, along with the target value differences, are stored in lists for subsequent use in the training process.

Algorithm 45 Generate Differences Algorithm

function GENERATEDIFFERENCES(X, y)
Input: Training images X, Target values y
Output: List of arrays representing differences $differences_X$, List of target differences $differences_y$
if Shape of X is More than 2-dimensional **then**
Reshape X to 2-dimensional array
else
Keep X as it is
end if
Increment $n\_neighbours\_1$ by 1
Initialize empty list $differences\_X$
Initialize empty list $differences\_y$
Fit nearest neighbors to find indices of nearest neighbors
Compute nearest neighbor indices for each image in X
for each image I in X **do**
Get nearest neighbors indices for I
for each neighbor index N **do**
Get neighbor image and target value
Compute absolute difference between I and neighbor image
Append difference to $differences\_X$
Append target value difference to $differences\_y$
end for
end for
return $differences\_X$, $differences\_y$
end function

Prediction Design

This algorithm, named "Image Classification Prediction Algorithm," takes a dataset of images X and a trained convolutional neural network (CNN) model $trainedCNN$ as input. It utilizes the trained CNN model to predict the differences between images in the dataset. The predicted differences are then processed further using the FinalPrediction algorithm (referenced as Algorithm 47) to compute the final predictions for the images. Finally, the algorithm returns the predicted target values $y_predictions$.

Algorithm 46 Image Classification Prediction Algorithm

1: **function** PREDICTIMAGECLASSIFIER(X, $trainedCNN$)
2: **Input:** Dataset of images X, Trained CNN model $trainedCNN$
3: **Output:** Predictions
4: Use trained CNN on X to predict differences
5: $difference\_pred \leftarrow trainedCNN(X)$
6: Compute final predictions using Algorithm: $y^{pred} \leftarrow$ FinalPrediction($difference\_pred$)
7: **return** y^{pred}
8: **end function**

This algorithm, titled "FinalPrediction," takes input data X, a trained CNN model $model$, and the input shape $input\_shape$. It computes the predicted target values y^{pred} based on the differences between the input data and the training images using the trained CNN model.

First, it fits nearest neighbors to the training data and then computes the absolute differences between each test image and its nearest neighbors from the training set. These differences are reshaped to match the input shape and used to make predictions using the trained CNN model.

Finally, it computes the predicted results for each nearest neighbor and determines the most common item among them to form the final prediction y^{pred}. If no items are found, it handles this case by appending "None" to y^{pred}.

Algorithm 47 Image Classification Prediction Algorithm

function FinalPredictionClassification(X, *model*, *input_shape*)
    Input: Input data X, Trained CNN model *model*, Input shape *input_shape*
    Output: Predicted target values *y_pred*
    Fit nearest neighbors to the training data
    Initialize empty list $differences\_test\_X$
    for *test* in range(length of X) **do** ▷ Compute differences between test and training images
        for *nn_in_training* in range(length of nearest neighbors) **do**
            Compute difference between test and nearest neighbor:
            $diff \leftarrow$ absolute difference between $X[test]$ and $self.train\_X[indices[test][nn\_in\_training]]$
            Append $diff$ to $differences\_test\_X$
        end for
    end for
    Reshape $differences\_test\_X$ to match input shape
    Make predictions using the trained CNN model: $predictions \leftarrow model.predict(differences\_test\_X)$
    Initialize empty list *y_pred*
    for *indexes*, *differences* in zip(*indices*, *predictions*) **do**
        Compute predicted results for each nearest neighbor:
        $results \leftarrow$ [self.train_y[i][0] + d for i, d in zip(*indexes*, *differences*)]
        Compute counts of each result: *counts* $\leftarrow$ count of each value in *results*
        Retrieve most common item: *most_common_item* $\leftarrow$ most common value in *results*
        if *most_common_item* exists **then**
            Append *most_common_item* to y^{pred}
        else
            Append *None* to y^{pred} ▷ Handle cases with no items
        end if
    end for
    return y^{pred}
end function

3.11.6 Training and Prediction Implementation

Training Implementation

Algorithm 48 Initialize `LingerImageClassifier`

1: Number of nearest neighbors for difference computation
2: **Input:** $n\_neighbours\_1, n\_neighbours\_2$
3: **procedure** InitializeClassifier($n\_neighbours\_1, n\_neighbours\_2$)
4: $self.n\_neighbours\_1 \leftarrow n\_neighbours\_1$
5: $self.n\_neighbours\_2 \leftarrow n\_neighbours\_2$
6: **return** self
7: **end procedure**

Algorithm 49 Generate Differences

```
procedure GenerateDifferences(X, y, n_neighbours)
    Input: X - dataset of images, y - target labels, n_neighbours - number of neighbors
    Differences in images and labels
    Output: differences_X, differences_y
    Initialize differences_X and differences_y as empty lists
    neigh ← Fit NearestNeighbors on X with n_neighbours
    for each image I in X do
        Find n_neighbours nearest neighbors for I
        for each neighbor N do
            Compute absolute difference between I and N
            Append difference to differences_X
            Compute difference in target values and append to differences_y
        end for
    end for
    return differences_X, differences_y
end procedure
```

Algorithm 50 Train with `LingerImageClassifier`

1: **procedure** TRAIN `LINGERIMAGECLASSIFIER`($X_{train}, y_{train}, X_{test}$)
2: Load dataset into $X_{train}, y_{train}, X_{test}, y_{test}$
3: Initialize `LingerImageClassifier` with $n\_neighbours\_1$ and $n\_neighbours\_2$
4: $differences\_X, differences\_y \leftarrow$ GenerateDifferences(X_{train}, y_{train})
5: Train CNN model on $differences\_X, differences\_y$
6: **end procedure**

Prediction Implementation

Algorithm 51 Prediction Implementation with LingerImageClassifier

X = test data set, $model$ = Trained CNN model, $input\_shape$ = image shape
procedure PREDICT($X, model, input\_shape$) ▷ Pythonic steps for predicting target values using image differences.
    Input: X - Input data, $model$ - Trained CNN, $input\_shape$ - Shape for CNN input.
    Output: y_pred - Predicted targets.
    $nbrs \leftarrow$ NearestNeighbors($n\_neighbors = self.n\_neighbours\_2$).fit($self.train\_X$.reshape($len$
    $indices \leftarrow nbrs.kneighbors(X.reshape(len(X), -1), return\_distance = False)$
    $differences\_test\_X \leftarrow$ [np.abs($X[test].flatten() - self.train\_X[indices[test][nn]].flatten()$) for $test$ in range($len(X)$) for nn in range($len(indices[0])$)]
    $differences\_test\_X \leftarrow np.array(differences\_test\_X).reshape(-1, *input\_shape)$
    $predictions \leftarrow model.predict(differences\_test\_X)$
    $y^{pred} \leftarrow []vector$
    for $indexes, differences$ in zip($indices, predictions$) **do**
        $results \leftarrow [self.train\_y[i][0] + d$ for i, d in zip($indexes, differences$)]
        $counts \leftarrow$ Counter($results$)
        $most\_common \leftarrow counts.most\_common(1)$
        if $most\_common$ **then**
            $y^{pred}.append(most\_common[0][0])$
        else
            $y^{pred}.append(None)$
        end if
    end for
    return y^{pred}
end procedure

3.12 Linger Image Regressor Model `LingerImageRegressor` (LIMR)

3.12.1 Objective

The motivation behind the development of (LIMR) stems from the growing demand for more sophisticated image regression tools capable of image data and solving regression issues. Traditional models often fall short in terms of explainability, and transparency. (LIMR) aims to address these gaps by:

- Incorporating state-of-the-art convolutional neural network (CNN) designs that

improve upon the limitations of existing models in capturing intricate patterns in image data.

- Utilizing a novel training methodology seen in [7] that enhances model learning from image differences, inspired by recent research findings.
- Aiming to demystify the black-box nature of deep neural networks by providing insights into the decision-making process, making the model more trustworthy and applicable in sensitive areas such as medical diagnosis and autonomous driving.

3.12.2 Architecture

Its architecture is similar to the classifier, only that it uses a convolutional neural net to predict continuous values rather than classes. The method of applying the generated differences to the data during prediction varies slightly. The structure of the `LingerImageClassifier`, and `LingerImageRegressor` is identical, the functionality of thier predict methods vary. See section 3.11.2 for a more indepth analysis.

3.12.3 Components

The key component within the `LingerImageRegressor` are identical to those in the `LingerImageClassifier`. For more details see section 3.11.3.

3.12.4 Parameterization

The `LingerImageRegressor` is designed with flexibility and adaptability at its core, offering a comprehensive suite of parameters that can be fine-tuned to optimize performance across a wide range of image classification tasks. The use can vary the design of the (CNN) convolutional Neural Network, as well as vary the number of neighbors used in the fit and prediction methods.

3.12.5 Training and Prediction Design

Training Design

The Image Regression Training Algorithm outlines the steps involved in training a convolutional neural network (CNN) for image regression tasks using the differences between nearest neighbor images. For regression, it uses the same algorithm **??** to generate the image differences datasets as `LingerImageClassifier`. . To finalize its

Algorithm 52 Image Regression Training Algorithm

function TRAINIMAGEREGRESSOR(X, y)
  Input: Dataset of images X, Target labels y
  Output: Trained CNN model
  Generate the image difference datasets.
  $differences\_X, differences\_y \leftarrow$ GenerateDifferences(X, y)
  Initialize (CNN) model.
  Train CNN model using $differences\_X$ and $differences\_y$
  return Trained CNN model
end function

prediction it takes an average of nearest neighbors with their predicted differences added to them.

This algorithm outlines a method for image regression prediction using a Convolutional Neural Network (CNN). It involves fitting nearest neighbors to training data, calculating differences between test images and their nearest training images using, and using these differences as inputs to a pre-trained CNN model to predict target values. The algorithm computes predictions by averaging adjustments made to the nearest neighbors' target values based on the CNN's output 53, thereby producing a final prediction for each input image. This approach combines instance-based learning with deep learning to enhance prediction accuracy for image-based regression tasks. The method of learning from differences is based on the algorithms in the paper [7],

Algorithm 53 Image Regression Prediction Algorithm

function FINALPREDICTION(X, $model$, $input\_shape$)
    Input: Input data X, Trained CNN model $model$, Input shape $input\_shape$
    Output: Predicted target values y_pred
    Fit nearest neighbors to the training data
    Initialize empty list $differences\_test\_X$
    for $test$ in range(length of X) **do** ▷ Compute differences between test and training images
        for $nn\_in\_training$ in range(length of nearest neighbors) **do**
            Compute difference between test and nearest neighbor:
            $diff \leftarrow$ absolute difference between $X[test]$ and $train\_X[indices[test][nn\_in\_training]]$
            Append $diff$ to $differences\_test\_X$
        end for
    end for
    Reshape $differences\_test\_X$ to match input shape
    Make predictions using the trained CNN model: $predictions \leftarrow model.predict(differences\_test\_X)$
    Initialize empty list y^{pred}
    for $indexes$, $differences$ in zip($indices$, $predictions$) **do**
        Compute predicted results for each nearest neighbor:
        $results \leftarrow$ [self.train_y[i][0] + d for i, d in zip($indexes$, $differences$)]
        Compute average of each result: $average \leftarrow$ average of values in $results$
        Append $average$ to y^{pred}
    end for
    return y^{pred}
end function

3.12.6 Training and Prediction Implementation

Training Implementation

The same algorithm 49 is used to generate differences for the training process in the `LingerImageRegressor` as the `LingerImplicitClassifier`.

Algorithm 54 Initialize `LingerImageRegressor`

Number of nearest neighbors for difference computation
Input: $n\_neighbours\_1, n\_neighbours\_2$
procedure INITIALIZEREGRESSOR($n\_neighbours\_1, n\_neighbours\_2$)
    $self.n\_neighbours\_1 \leftarrow n\_neighbours\_1$
    $self.n\_neighbours\_2 \leftarrow n\_neighbours\_2$
    return self
end procedure

Algorithm 55 Train with `LingerImplicitRegressor`

procedure TRAIN `LINGERIMAGEREGRESSOR`($X_{train}, y_{train}, X_{test}$)
    Load dataset into $X_{train}, y_{train}, X_{test}, y_{test}$
    Initialize `LingerImageRegressor` with $n\_neighbours\_1$ and $n\_neighbours\_2$
    $differences\_X, differences\_y \leftarrow$ GenerateDifferences(X_{train}, y_{train})
    Train CNN model on $differences\_X, differences\_y$
end procedure

Prediction Implementation

Algorithm 56 Prediction with LingerImageRegressor

procedure PREDICT($X, model, dataset, input\_shape$)
    Predicts target values using a regression approach based on image differences.
    Input: X - Input data for prediction, *model* - Trained regression model, *dataset* - Dataset object containing training data, $input\_shape$ - Shape of input data expected by the model.
    Output: y^{pred} - Predicted target values for the input data.
    if $len(X.shape) > 2$ **then**
        $X \leftarrow X.reshape(len(X), -1)$ ▷ Flatten the input if it's multi-dimensional.
    end if
    Train the NearestNeighbors function on our data.
    $nbrs \leftarrow$ NearestNeighbors($n\_neighbors = self.n\_neighbours\_2$).fit($dataset.train\_X.reshape$
    $indices \leftarrow nbrs.kneighbors(X, return\_distance = False)$ ▷ Identify nearest neighbors in the training set.
    Prepare $differences\_test\_X$ by computing absolute differences between X and its nearest neighbors in the training set.
    Reshape $differences\_test\_X$ to match $input\_shape$ for compatibility with the model.
    $predictions \leftarrow model.predict(differences\_test\_X)$ ▷ Predict target values based on computed differences.
    Initialize an empty list y^{pred} to store final predictions.
    for each $indexes, differences$ pair in zipped $indices, predictions$ **do**
        Calculate the sum of the training target values and the predicted differences.
        Compute the average result to obtain the regression prediction.
        Append the average result to y^{pred}.
    end for
    return y^{pred} ▷ Return the list of predicted target values.
end procedure

3.13 Optimization Strategies

Across all Linger models, a consistent focus on hyperparameter tuning and architectural refinements enhances performance and interpretability. For instance:

- **Hyperparameter Tuning:** Each model undergoes extensive hyperparameter optimization to identify settings that maximize accuracy and efficiency. This includes adjustments to learning rates, neighbor counts, and decision thresholds.
- **Architectural Adjustments:** Modifications in model architectures, especially for variant models, aim to better capture the nuances of learning from differences.

This involves experimenting with different layers, activation functions, and aggregation methods to improve model responsiveness to dataset specifics.

- **CNN Configurations for Image-Based Models:** Image models, such as the LingerImageClassifier and LingerImageRegressor, benefit from tailored Convolutional Neural Network (CNN) configurations. By optimizing filter sizes, stride patterns, and pooling layers, these models achieve significant gains in learning efficiency and prediction accuracy.
- **Computational Efficiency:** Efforts are underway to enhance the computational efficiency of the models, focusing on reducing training times without compromising performance. Techniques include model pruning, efficient data loading, and leveraging hardware accelerations.
- **Dataset Applicability:** A key optimization goal is to ensure models can adapt to a wide range of datasets, encompassing varying sizes, dimensions, and complexity. This involves refining data preprocessing and feature extraction techniques to maintain model robustness across diverse data scenarios.

3.14 Limitations and Future Work

While the Linger models introduce innovative approaches to machine learning, they encounter certain limitations that direct future research:

- **Computational Intensity:** The models, especially when dealing with large datasets or complex image data, can be computationally demanding. Ongoing research seeks to address this through algorithmic improvements that reduce computational overhead without sacrificing accuracy.
- **Data Quality Dependency:** The effectiveness of the models is closely tied to the quality and representativeness of the training data. Efforts to make the models more robust to noisy or incomplete data include developing advanced noise filtering techniques and data augmentation strategies.
- **Scalability to Larger Datasets:** As datasets grow in size and complexity, scaling model training and inference efficiently becomes a challenge. Future developments aim to enhance the scalability of Linger models, potentially through distributed computing techniques and more efficient data handling mechanisms.

- **Extension to New Domains:** Exploring applications of Linger models in new domains, such as time series analysis, natural language processing, and more complex forms of image and video data, represents a promising avenue for future work. This includes adapting the learning from differences approach to tackle domain-specific challenges.
- **Algorithmic Efficiency:** Developing more efficient algorithms to handle the iterative processes inherent in learning from differences is a primary focus. This may involve exploring new mathematical frameworks or leveraging advancements in optimization theory.
- **Broadening Data Types and Applications:** The adaptability of Linger models to various data types and application contexts remains an area for exploration. Future research will investigate the integration of these models into multi-modal learning frameworks and their applicability in interdisciplinary fields.

Chapter 4

Experiments

4.1 Research Objective

The overarching objective of this study is to investigate the efficacy of blended learning approaches in machine learning tasks, particularly focusing on regression and classification problems. Blended learning, which integrates the strengths of model-based and instance-based learning while mitigating their respective weaknesses, offers a promising avenue for enhancing the interpretability, reliability, and performance of machine learning systems.

Specifically, this research aims to achieve the following objectives:

1. **Evaluate Blended Learning Methods**: The first objective is to evaluate the performance of blended learning methodologies in contrast to traditional model-based and instance-based approaches across various regression and classification tasks. This assessment will encompass key metrics including accuracy, interpretability, computational efficiency, and robustness. By comparing the effectiveness of blended learning methods, we aim to determine their potential to enhance the overall performance of machine learning systems.
2. **Investigate Model Interpretability**: The second objective is to explore the interpretability of blended learning models compared to traditional black-box models, such as neural networks. Understanding the interpretability of these models is crucial for enhancing trust and usability in critical domains. By investigating how blended learning methods provide insights into model decisions, we aim to demonstrate their potential to address the interpretability challenges associated with traditional machine learning approaches.

3. **Examine Efficiency and Scalability**: The third objective is to investigate the computational efficiency and scalability of blended learning methods, particularly in scenarios with large datasets. This examination will involve analyzing the memory and computational overhead of blended learning models compared to traditional approaches. By assessing the efficiency and scalability of blended learning methods, we aim to identify their suitability for practical deployment in real-world applications.

4. **Explore Novel Architectures**: The fourth objective is to explore novel architectures and techniques for implementing blended learning, including ensemble models, case-based reasoning principles, and approaches that leverage differences between sets of features. By exploring innovative approaches, we aim to further enhance the performance and interpretability of blended learning methods, thereby advancing the state-of-the-art in machine learning.

5. **Provide Insights for Practical Applications**: The fifth objective is to provide insights and recommendations for the practical application of blended learning methods in real-world scenarios, with a particular focus on critical domains such as healthcare, finance, and decision support systems. By discussing the implications of the findings for stakeholders and suggesting best practices for deploying blended learning models effectively, we aim to facilitate the adoption of these methods in critical domains.

6. **Develop Python Package for Seamless Integration**: The final objective is to develop a Python package that seamlessly integrates with scikit-learn, allowing users to incorporate blended learning methods into their machine learning pipelines and workflows effortlessly. This package should provide compatibility with scikit-learn's tools, enabling users to import it like any other scikit-learn package and use it seamlessly within their existing workflows. By developing such a package, we aim to enhance the accessibility and usability of blended learning methods for the broader machine learning community.

By addressing these objectives, this research seeks to advance our understanding of blended learning approaches and their potential to address the challenges faced by traditional model-based and instance-based methods. Ultimately, the goal is to contribute to the development of more interpretable, reliable, and efficient machine learning systems with broader applicability across various domains.

4.1.1 Data Collection and Preprocessing

All datasets used in this study were collected from the UCI Machine Learning Repository [1], a widely recognized and reliable source of machine learning datasets. These datasets cover various domains and have been extensively studied in the machine learning community.

Preprocessing For Labelled Datasets

The preprocessing procedure involves several steps to ensure the quality and suitability of the data for machine learning tasks. These steps are applied uniformly across all labeled datasets.

1. **Column Naming**: The columns of the dataset are assigned appropriate names for clarity and consistency throughout the analysis.

2. **Handling Missing Values**: Rows containing missing values in specified features are removed from the dataset. This ensures that the dataset used for training and testing is complete and does not contain any incomplete or unreliable data points.

3. **Index Resetting**: After removing rows with missing values, the index of the dataset is reset to maintain a continuous and consistent index structure.

4. **Data Splitting**: The dataset is split into two subsets: a development set and a test set. This splitting is performed using a train-test split ratio of $80:20$ to ensure a sufficient amount of data for training while also reserving a portion for evaluating the model's performance.

5. **Development Set Copying**: A copy of the development set is created to preserve the original dataset for future reference or comparison.

6. **Feature Transformation**: Numeric features undergo feature scaling using the `StandardScaler` to ensure that they are on a similar scale, which aids in model convergence and performance. Nominal features are transformed using one-hot encoding to convert categorical variables into a numerical format suitable for machine learning algorithms.

These preprocessing steps collectively prepare the data for subsequent modeling and analysis, ensuring that it is clean, complete, and appropriately formatted for use in machine learning tasks.

4.2 Results and Analysis

4.2.1 Experiment 1: Glass Dataset Classification

Methodology

In Experiment 1, we conducted classification tasks on the Glass dataset using a two-step methodology. Initially, we trained a neural network (NN) classifier through grid search and cross-validation to determine the optimal hyperparameters. These hyperparameters were then applied to train a `LingerClassifier` (LC) and its variants for improved robustness, tested across five runs. For comparative analysis, we also evaluated the performance of K-Nearest Neighbors (KNN), weighted (KNN), and (NN) on the same dataset. The results section details the hyperparameters selected and the test accuracies achieved.

The `LingerImplicitClassifier` was only run over 3 iterations due to a lack of resources.

Hyperparameter Optimization

For both the Neural Network and Linger Classifier, a comprehensive exploration of hyperparameters was conducted, as summarized in Table 4.1. This table delineates the varied hyperparameters scrutinized during the optimization phase, spanning hidden layer configurations, activation functions, regularization terms, and more, underscoring our meticulous approach to model tuning.

Table 4.1: Tested Hyperparameters for Neural Network and Linger Classifier

Hyperparameter	**Values**
Neural Network	
Hidden Layer Sizes	(256, 128), (128, 64), (100,), (200, 100), (300, 200, 100)
Activation	identity, logistic, tanh, relu
Alpha	0.0001, 0.001, 0.01, 0.1
Max Iterations	1000, 1500
Early Stopping	True
Validation Fraction	0.1, 0.2, 0.3
Learning Rate Init	0.001, 0.01, 0.1
Solver	adam, sgd
Beta 1	0.9, 0.95, 0.99
Beta 2	0.999, 0.995, 0.9
Linger Classifier	
Neighbors 1	2, 5, 7, 10, 13, 15, 17, 21
Neighbors 2	2, 5, 7, 10, 13, 15, 17, 21
Weighted KNN	False
Additional Distance Column	True, False
Duplicated on Distance	True, False
Addition of Context	True, False

Results

Table 4.2: Best Parameters for KNN Comparison Models

Classifier	**Hyperparameter**	**Value**
KNN	n_neighbors	2
Weighted KNN	n_neighbors	7

Table 4.3: Summary of Best Parameters for `LingerClassifier` and it's Variations

Parameter	**LC**	**LCV1**	**LCV2**	**LCV3**
Activation	relu	relu	relu	relu
Alpha	0.0001	0.1	0.1	0.001
Beta 1	0.9	0.9	0.9	0.99
Beta 2	0.995	0.999	0.995	0.999
Early Stopping	True	True	True	True
Hidden Layer Sizes	(300, 200, 100)	(300, 200, 100)	(300, 200, 100)	(300, 200, 100)
Learning Rate Init	0.01	0.01	0.01	0.01
Max Iter	1000	1500	1000	1000
Solver	adam	adam	adam	adam
Validation Fraction	0.3	0.3	0.3	0.3
Addition of Context	False	True	False	False
Additional Distance Column	False	False	False	True
Duplicated on Distance	False	False	True	False
N Neighbors 1	2	17	21	21
N Neighbors 2	5	17	13	13
Weighted KNN	False	False	False	False

Table 4.4: Summary of Best Parameters for `LingerImplicitClassifier` and Its Variations

Parameter	**LIC**	**LICV1 (Random)**	**LICV2 (Single)**
Activation	tanh	tanh	tanh
Alpha	0.01	0.001	0.001
Beta 1	0.9	0.9	0.9
Beta 2	0.9	0.9	0.9
Hidden Layer Sizes	(128, 64)	(128, 64)	(128, 64)
Learning Rate Init	0.01	0.01	0.01
Max Iter	2000	2000	2000
Solver	sgd	sgd	sgd
Validation Fraction	0.1	0.1	0.1
N Neighbors 1	21	7	10
N Neighbors 2	13	2	21
Random Pairs	False	True	False
Single Pair	False	False	True

Table 4.5: Summary of Training and Test Accuracy Across All Models

Model / Variation	**Training Accuracy (%)**	**Test Accuracy (%)**
KNN	68.2	60.46
Weighted KNN	70.0	62.8
Neural Network	73.1	61.4
LC (Base)	70.04	60.46
LCV1 (Context)	69.41	65.70
LCV2 (Duplicated on Distance)	67.29	**69.30**
LCV3 (Additional Distance Column)	67.65	60.93
LIC (Implicit)	41.4	51.2
LICV1 (Random)	31.17	41.08
LICV2 (Single)	50.39	47.28

4.2.2 Experiment 2: Abalone Dataset Regression

Methodology

In Experiment 2, we explored regression tasks on the Abalone dataset with a two-step approach. Initially, a neural network (NN) regressor was trained using grid search and cross-validation to identify optimal hyperparameter combinations. These selected hyperparameters were then applied to train a `LingerRegressor` (LR) and its variants, targeting improved robustness, with their performance evaluated across four runs. Additionally, to benchmark the `LingerRegressor`'s performance, K-Nearest Neighbors (KNN), weighted KNN, and NN models were similarly assessed on the dataset. The results section will detail the hyperparameters and performance metrics obtained from Experiment 2.

LRV1 run 4 times due to resource restrictions. LRV2 was run 3 times due to the computational overhead. LIR was run 3 times LIRV1 was run 4 times. Accuracy is used for LIR and all variations. NMSE is used for LR and all variations.

Tested Hyperparameters

Table 4.6: Tested Hyperparameters for Neural Network and `LingerRegressor`

Hyperparameter	**Values**
Neural Network	
Hidden Layer Sizes	(256, 128), (128, 64), (200, 100), (300, 200, 100)
Activation	relu
Alpha	0.0001, 0.001, 0.01, 0.1
Max Iterations	2000
Early Stopping	True
Validation Fraction	0.1
Learning Rate Init	0.001, 0.01, 0.1
Solver	adam, sgd
Beta 1	0.9, 0.95, 0.99
Beta 2	0.999, 0.995, 0.9
Linger Regressor	
N Neighbours 1	2, 5, 7, 10, 13, 15, 17, 21
N Neighbours 2	2, 5, 7, 10, 13, 15, 17, 21
Weighted KNN	False
Additional Distance Column	True, False
Duplicated on Distance	True, False
Addition of Context	True, False

Results

Table 4.7: Best Parameters for KNN Comparison Models

Classifier	**Hyperparameter**	**Value**
KNN	n_neighbors	17
Weighted KNN	n_neighbors	21

Table 4.8: Summary of Best Parameters for `LingerRegressor` and its Variations

Parameter	**LR**	**LRV1**	**LRV2**	**LRV3**
Activation	relu	relu	relu	relu
Alpha	0.01	0.001	0.1	0.01
Beta 1	0.95	0.95	0.95	0.99
Beta 2	0.9	0.9	0.9	0.9
Early Stopping	True	True	True	True
Hidden Layer Sizes	(200, 100)	(200, 100)	(256, 128)	(200, 100)
Learning Rate Init	0.01	0.01	0.01	0.01
Max Iter	2000	2000	2000	2000
Solver	sgd	sgd	sgd	sgd
Validation Fraction	0.1	0.1	0.1	0.1
Addition of Context	False	True	False	False
Additional Distance Column	False	False	False	True
Duplicated on Distance	False	False	True	False
N Neighbors 1	5	2	21	5
N Neighbors 2	21	17	21	17
Weighted KNN	False	False	False	False

Table 4.9: Summary of Best Parameters for `LingerImplicitRegressor` and Its Variations

Parameter	**LIR**	**LIRV1**	**LIRV2**
Activation	relu	relu	relu
Alpha	0.01	0.01	0.01
Beta 1	0.99	0.9	0.9
Beta 2	0.9	0.9	0.9
Early Stopping	True	True	True
Hidden Layer Sizes	(128, 64)	(256, 128)	(200, 100)
Learning Rate Init	0.01	0.01	0.01
Max Iter	2000	2000	2000
Solver	sgd	sgd	sgd
Validation Fraction	0.1	0.1	0.1
N Neighbors 1	13	17	15
N Neighbors 2	21	13	7
Random Pairs	False	True	False
Single Pair	False	False	True

Table 4.10: Summary of Training and Test NMSE/Accuracy Across All Models

Model / Variation	**Training NMSE / Accuracy (%)**	**Test NMSE / Acc**
KNN	−1.959	−1.930
Weighted KNN	−1.941	−1.912
Neural Network	−1.930	−1.895
LR (Base)	−1.898	−1.835
LRV1 (Context)	−1.949	−1.897
LRV2 (Duplicated on Distance)	−2.047	−1.949
LRV3 (Additional Distance Column)	−1.9006	**−1.8250**
Neural Network with Labels Classified	67.3	**64.8**
LIR (Implicit)	63.2	57.7
LIRV1 (Random)	52.6	48
LIRV2 (Single)	59.7	52

4.2.3 Experiment 3: Raisin Dataset Classification

Methodology

In Experiment 3, we conducted a comprehensive analysis on the Raisin dataset sourced from the UCI Machine Learning Repository [3], specifically detailed in [5]. Our methodology was two-fold: initially, we determined the optimal hyperparameters for a neural network (NN) classifier through an exhaustive grid search coupled with cross-validation. This foundational step ensured the NN classifier was finely tuned for the dataset. Subsequently, these hyperparameters were employed to train the `LingerClassifier` (LC) and its variants, focusing on enhancing model robustness. The performance of these classifiers was benchmarked against K-Nearest Neighbors (KNN), Weighted KNN, and the baseline NN classifier through five independent runs to ensure statistical reliability. Our comparative analysis leverages this diverse classifier ensemble to draw insights into the efficacy and robustness of each model relative to the Raisin dataset, showcasing the nuanced trade-offs between model complexity and classification accuracy. Due to time and resouce constraints the certain classifiers were run for a different number of iterations.

- `LingerClassifier` (LC) was run 4 times.
- `LingerClassifier` variant 1 (LCV1) was run twice.
- `LingerClassifier` variant 2 (LCV2) was run twice.
- item `LingerClassifier` variant 3 (LCV3) was run twice.
- `LingerImplicitClassifier` (LIC), along with its variants LICV1 and LICV2, were each run twice.

Tested Hyperparameters

Table 4.11: Tested Hyperparameters for Neural Network and `LingerClassifier`

Hyperparameter	**Values**
Neural Network	
Hidden Layer Sizes	(256, 128), (128, 64), (200, 100), (300, 200, 100), (400, 300, 200, 100)
Activation	identity, logistic, tanh, relu
Alpha	0.0001, 0.001, 0.01, 0.1
Max Iterations	1500, 2000
Early Stopping	True
Validation Fraction	0.1, 0.2, 0.3
Learning Rate Init	0.001, 0.01, 0.1
Solver	adam, sgd
Beta 1	0.9, 0.95, 0.99
Beta 2	0.999, 0.995, 0.9
Linger Regressor	
N Neighbours 1	2, 5, 7, 10, 13, 15, 17, 21
N Neighbours 2	2, 5, 7, 10, 13, 15, 17, 21
Weighted KNN	False
Additional Distance Column	True, False
Duplicated on Distance	True, False
Addition of Context	True, False

Results

Table 4.12: Best Parameters for KNN Comparison Models

Classifier	**Hyperparameter**	**Value**
KNN	n_neighbors	13
Weighted KNN	n_neighbors	13

Table 4.13: Summary of Best Parameters for `LingerClassifier` and its Variations

Parameter	**LC**	**LCV1**	**LCV2**	**LCV3**
Activation	identity	tanh	tanh	identity
Alpha	0.01	0.01	0.01	0.01
Beta 1	0.9	0.95	0.9	0.9
Beta 2	0.999	0.9	0.9	0.9
Early Stopping	True	True	True	True
Hidden Layer Sizes	(128, 64)	(256, 128)	(128, 64)	(300, 200, 100)
Learning Rate Init	0.001	0.001	0.001	0.001
Max Iter	1500	1500	1500	2000
Solver	adam	adam	adam	adam
Validation Fraction	0.1	0.3	0.3	0.3
Addition of Context	False	True	False	False
Additional Distance Column	False	False	False	True
Duplicated on Distance	False	False	True	False
N Neighbors 1	5	15	10	2
N Neighbors 2	13	21	21	13
Weighted KNN	False	False	False	False

Table 4.14: Summary of Best Parameters for `LingerImplicitClassifier` and Its Variations

Parameter	**LIC**	**LICV1**	**LICV2**
Activation	tanh	identity	identity
Alpha	0.1	0.01	0.01
Beta 1	0.95	0.9	0.9
Beta 2	0.999	0.999	0.999
Early Stopping	True	True	True
Hidden Layer Sizes	(128, 64)	(400, 300, 200, 100)	(400, 300, 200, 100)
Learning Rate Init	0.1	0.001	0.001
Max Iter	1500	2000	2000
Solver	sgd	adam	adam
Validation Fraction	0.2	0.3	0.3
N Neighbors 1	2	2	7
N Neighbors 2	13	13	13
Random Pairs	False	True	False
Single Pair	False	False	True

Table 4.15: Summary of Training and Test Accuracy Across All Models

Model / Variation	**Accuracy (%)**	**Accuracy (%)**
KNN	91.7	91
Weighted KNN	91.2	91
Neural Network	94.1	91
LC (Base)	91.7	90.5
LCV1 (Context)	92.2	89.5
LCV2 (Duplicated on Distance)	92.3	90.5
LCV3 (Additional Distance Column)	91.7	91
LIC (Implicit)	91.6	90.5
LICV1 (Random)	91.6	89.5
LICV2 (Single)	91.5	91

Chapter 5

Conclusions and Future Work

This chapter outlines the additions made in our study, acknowledges its limitations, and proposes directions for future research to extend the work's applicability and efficiency across multiple domains

5.1 Addressing Identified Limitations

Our study, while insightful, recognizes certain limitations in its current scope and methodology. Future efforts should aim to surpass these through advanced algorithmic enhancements and the integration of novel data processing techniques, this could be implemented through Scikit-learn [**?**]. Such improvements are anticipated to refine the learning process further, enhancing the models' overall performance and applicability.

5.2 Broader Applicability

Exploring the applicability of our approach to broader machine learning challenges represents a promising direction for future research. Adapting our models to handle unstructured data or complex, high-dimensional datasets could significantly broaden their utility, making them suitable for a wider range of applications.

5.3 Integration with Emerging Technologies

The potential integration of our models with emerging technologies, such as deep learning and reinforcement learning, offers exciting possibilities. Such integration could not only provide new insights and enhance model capabilities but also facilitate the use of these models in real-time applications or embedded systems where interpretability and efficiency are paramount.

5.4 Scalability and Efficiency

Investigating the scalability and efficiency of our models, especially in contexts involving large-scale datasets or distributed computing environments, is crucial. Future research could focus on optimizations that reduce training times or adaptations that leverage parallel processing architectures to improve performance.

5.5 Cross-Domain Applications

Examining the effectiveness of our models across different domains or for various types of prediction tasks could uncover new applications and opportunities for impact. This exploration might involve interdisciplinary research collaborations to adapt and evaluate our models in diverse contexts. Evaluating the learning from difference models with real time data.

5.6 Exploring Other Categorical Similarity Measures

Investigating alternative measures for assessing the similarity of categorical features could refine the process of generating and applying adaptation rules, potentially enhancing the accuracy and applicability of the Ensemble of Adaptations for Classification (EAC). As seen in the paper [10].

Bibliography

[1] David Aha. Machine learning repository, 1987.

[2] David W Aha, Dennis Kibler, and Marc K Albert. Instance-based learning algorithms. *Machine learning*, 6:37–66, 1991.

[3] Arthur Asuncion and David Newman. Uci machine learning repository, 2007.

[4] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

[5] Illkay Cinar, Murat Koklu, and Sakir Tasdemir. Raisin. UCI Machine Learning Repository, 2023. DOI: https://doi.org/10.24432/C5660T.

[6] Susan Craw, Nirmalie Wiratunga, and Ray C Rowe. Learning adaptation knowledge to improve case-based reasoning. *Artificial intelligence*, 170(16-17):1175–1192, 2006.

[7] Mathieu D'Aquin, Emmanuel Nauer, and Justine Lieber. A factorial study of neural network learning from differences for regression. In Mark T. Keane and Nirmalie Wiratunga, editors, *Case-Based Reasoning Research and Development. ICCBR 2022. Lecture Notes in Computer Science*, volume 13405, page PageNumbersHere. Springer, Cham, 2022.

[8] Preeti Gupta, Arun Sharma, and Rajni Jindal. Scalable machine-learning algorithms for big data analytics: a comprehensive review. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 6(6):194–214, 2016.

[9] Kathleen Hanney and Mark T Keane. Learning adaptation rules from a case-base. In *European Workshop on Advances in Case-Based Reasoning*, pages 179–192. Springer, 1996.

[10] Vahid Jalali, David Leake, and Najmeh Forouzandehmehr. Learning and applying case adaptation rules for classification: An ensemble approach. In *IJCAI*, pages 4874–4878, 2017.

[11] David B Leake and Andrew Kinley. Acquiring case adaptation knowledge: A hybrid approach. 1996.

[12] Leorey Marquez, Tim Hill, Reginald Worthley, and William Remus. Neural network models as an alternative to regression. In *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, volume 4, pages 129–135. IEEE, 1991.

[13] Christopher K Riesbeck and Roger C Schank. *Inside case-based reasoning*. Psychology Press, 2013.

[14] Cynthia Rudin. Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead. *Nature machine intelligence*, 1(5):206–215, 2019.

[15] scikit-learn contributors. scikit-learn. `https://github.com/scikit-learn/scikit-learn`, Accessed: Month Day, Year.

[16] Sergios Theodoridis. *Machine learning: a Bayesian and optimization perspective*. Academic press, 2015.

[17] Ian Watson and Farhi Marir. Case-based reasoning: A review. *The knowledge engineering review*, 9(4):327–354, 1994.

[18] Xiaomeng Ye, David Leake, Vahid Jalali, and David J Crandall. Learning adaptations for case-based classification: A neural network approach. In *Case-Based Reasoning Research and Development: 29th International Conference, ICCBR 2021, Salamanca, Spain, September 13–16, 2021, Proceedings 29*, pages 279–293. Springer, 2021.