Noise Complaints: Novel Noise-Resistance for kNN

Department of Computer Science

University College Cork

Student Name: Matthew Mallen
Supervisor: Dr Derek Bridge

Final-Year Project – BSc in Computer Science
April 2024

Abstract

This project enhances the k-Nearest Neighbors (kNN) algorithm's resilience to noise by developing a novel noise-resistant variant named Noise Complaints kNN (NCkNN). By integrating concepts of reliability and similarity in neighbour selection and prediction-making while respecting the lazy learning model intrinsic to kNN, NCkNN offers a dynamic solution to the challenges posed by noisy data. The research thoroughly explores kNN, weighted kNN, and various noise-editing algorithms, emphasising their limitations and proposing innovative adaptations. Comprehensive testing across multiple datasets for classification and regression tasks, coupled with rigorous statistical analysis, underscores NCkNN's promising efficacy compared to traditional methods. However, further research is needed to claim this with a high degree of confidence. This work contributes to machine learning research by improving kNN's robustness and opens new avenues for future research in algorithmic noise resistance. Implemented in Python, the project showcases the feasibility of enhancing predictive accuracy in real-world applications, offering a valuable tool for data scientists and researchers seeking to mitigate the impact of unreliable or erroneous data points.

Declaration of originality

Declaration of Originality

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

I hereby declare that:

- this is all my own work, unless clearly indicated otherwise, with full and proper accreditation;
- with respect to my own work: none of it has been submitted at any educational institution contributing in any way to an educational award;
- with respect to another's work: all text, diagrams, code, or ideas, whether verbatim, paraphrased or otherwise modified or adapted, 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in a scholarly manner, whether from books, papers, lecture notes or any other student's work, whether published or unpublished, electronically or in print.

Signed: Matt Mallen

Date: 4/4/2024

Acknowledgments

This project would not have been possible without the guidance and expertise of Dr. Derek Bridge, whose insights and mentorship were invaluable throughout the research and development process. Friends, family, and loved ones are also due appreciation for their unwavering support and encouragement.

Contents

Chapter 1

Introduction

We have observed massive Artificial Intelligence (AI) growth these last few years [11]. Among many algorithms at the heart of AI development, the decades-old k-nearest neighbours (kNN) algorithm stands out for its simplicity yet applicability to many regression and classification tasks. More recently, kNN even successfully helped model the spread of COVID-19 [4]. This project is set within this lively landscape, aiming to refine and extend kNN's capabilities to more effectively deal with the unreliable and erroneous data endemic in most real-world data sets.

The main goal of this project is to design and implement a new noise-resilient variant of kNN. We thoroughly explored existing variants, such as weighted kNN and other existing adaptations designed to deal with noise. Our thorough assessment consisted of many datasets for both regression and classification tasks. A key focus of our investigation centred on editing algorithms, which generally aim to clean a dataset of noise and irrelevant data so that the predictive accuracy of the kNN algorithm is improved.

Our exploration illuminated a fundamental characteristic of kNN: that of being a lazy learner. A lazy learner defers all computation to the moment a user asks it to make a prediction, unlike an eager learner, which does all of this in advance, training and learning a generalised model. Preprocessing steps, like the ones introduced by editing algorithms, such as Blame-Based Noise Reduction (BBNR) and Repeated Edited Nearest Neighbors (RENN), step outside the classic kNN paradigm. These algorithms introduce a pre-prediction training step aimed at cleaning the dataset, which does not adhere to the principles of lazy learning in that it performs explicit learning before prediction, which we believe makes them less practically useful given it hinders some of the unique appeal of kNN that may make a user opt for it in a given application context.

Against this backdrop, we propose the Noise Complaints kNN (NCkNN) algorithm. NCkNN introduces the concept of data point "influence" by considering data reliability and similarity, yet it cleverly maintains allegiance to the essence of lazy learning. Unlike the mentioned editing algorithms, NCkNN confines its departure from strict laziness to the minimal pre-calculation of reliability values without altering the training data, preserving the prediction-time calculations that define lazy learning. This subtle yet strategic departure allows NCkNN to enhance prediction accuracy without significantly undermining the computational efficiency and adaptability that are the hallmarks of lazy learning.

With meticulous design and implementation, our project leverages the power of the Python programming language and a suite of machine learning libraries, which enabled a comprehensive evaluation of our algorithm against well-established benchmarks, demonstrating its effectiveness in improving kNNs' robustness against noise in various contexts.

This report chronicles our journey from conceptualisation to realisation, detailing the challenges we faced, the solutions we devised, and the lessons we learned. Through a blend of technical rigour and an innovative approach, we invite the reader to explore the intricacies of kNN, the nuances of editing algorithms, and the potential of our solution, NCkNN. We aim to inspire academic discourse and play a role in building even more resilient and adaptable machine-learning algorithms in the face of an ever-complex data landscape.

Chapter 2

Background

This chapter delves into the objectives of the project and the foundational aspects of kNN, from its operational mechanics to the various strategies devised to enhance its accuracy, particularly in the face of noisy or unrepresentative data. It also highlights significant gaps in existing literature, setting the stage for exploring novel methods to refine kNN's robustness and reliability in complex datasets.

2.1 Project Objectives

The objectives of this project are to implement kNN, weighted kNN, and existing noise-resistant variations and evaluate these competing algorithms against a variety of datasets for both classification and regression tasks. Furthermore, we wish to design and implement a novel noise-resistant kNN algorithm.

2.2 Establishing Notation

In this section, we highlight some basic necessary notation in kNN: $\mathbf{x}$ refers to a dataset example (a vector of feature values); y refers to the target value of a given $\mathbf{x}$. $\mathbf{X}$ refers to a matrix of examples, with each row being an example and each column being a feature value. $\mathbf{Y}$ refers to a column vector of target values. We aim to adhere to convention, as outlined by Chrzeszczyk & Kochanowski [3].

2.3 kNN Overview

This section introduces the k-Nearest Neighbours (kNN) algorithm. First proposed by Fix & Hodges [7], kNN is a foundational method in machine learning for classification and regression tasks. We outline the algorithm's basic principles, the importance of defining similarity, identifying nearest neighbours, and how it makes predictions.

2.3.1 Basic Concept

kNN's operating principle is that similar problems have similar solutions; that is to say, it assumes that the value or class of an example can be predicted by aggregating the value/class of the data points closest in distance to it.

2.3.2 Defining Similarity

Since the principle above relies on the "similarity" of various examples, one must define "similarity", i.e. the algorithm relies on some function to measure the similarity or distance between data points. Euclidean distance is the most common, although others can be used depending on the context. If $\mathbf{x}$ is one example (a vector of feature values) and $\mathbf{x'}$ is another, Equation 2.1 is a concise definition of Euclidean distance:

$$d(\mathbf{x},\mathbf{x'}) = \sqrt{\sum_{j=1}^{n} (\mathbf{x}_j - \mathbf{x'}_j)^2} \tag{2.1}$$

If $d(\mathbf{x},\mathbf{x'})$ is less than $d(\mathbf{x},\mathbf{x''})$ then $\mathbf{x'}$ is said to be more similar to $\mathbf{x}$ than $\mathbf{x''}$.

2.3.3 Identifying Nearest Neighbours

For a given query example, the algorithm finds k other examples in the training dataset closest to the query example, i.e. the k other training examples with the smallest Euclidean distances to the query example. These k examples are then the k-nearest neighbours.

Algorithm 1 k-Nearest Neighbours

```
Require: X - A set of training examples.
Require: Y - Target values / class labels for each example in X.
Require: x - The new example to predict a target value/class for.
Require: $k$ - The number of nearest neighbours to consider.
Ensure: $k > 0$ and $k <$ ((size of X)+1)
  distances ← empty list
  for all $\mathbf{x} \in \mathbf{X}$ do
      Compute distance (e.g. euclidean distance) $d$ between X and x.
      Add the tuple ($d$, $y$ corresponding to X) to distances.
  end for
  Sort distances based on distance values.
  Select first $k$ elements from sorted distances.
  Extract the target values / class labels from these $k$ elements.
  if regression task then
      Compute the mean of the $k$ target values.
      return this mean as the prediction for x.
  else if classification task then
      Compute the mode class of the $k$ nearest neighbours.
      return this mode as the prediction for x.
  end if
```

2.3.4 Making Predictions

The algorithm gives the final prediction for the query example using its k-nearest neighbours. For regression tasks (like predicting house prices), this is typically the mean of the target values of these nearest neighbours. In classification, it is the mode (or most common class) among them.

2.3.5 Algorithm Pseudocode

Algorithm 1 depicts the algorithm's design as pseudocode.

2.4 Weighted kNN Overview

This section delves into the Weighted k-Nearest Neighbours (Weighted kNN) algorithm, an advanced variant of kNN that introduces weighting to improve prediction accuracy by considering the distance of each neighbour from the query example.

2.4.1 Basic Concept

Weighted k-Nearest Neighbours (Weighted kNN) is a variation of the basic kNN algorithm. In this variant, each nearest neighbour does not contribute equally to the prediction. Instead, the influence of each neighbour is weighted, often based on the distance from the query example [6].

2.4.2 Computing Weights

Weights in weighted kNN are typically inversely proportional to the distance between the query example and each neighbour. Commonly, the weight assigned to a neighbour is the reciprocal of the distance; this ensures closer neighbours have a more substantial influence on the prediction outcome. Where distances is a vector containing the distances of the k-nearest neighbours from the query example and ϵ is a small constant (smallest that a computer system can allow for example) to avoid divisions by zero, Equation 2.2 can be used to calculate the weights.

$$weights = \frac{1}{distances + \epsilon} \tag{2.2}$$

2.4.3 Making Predictions

For regression tasks, weighted kNN predicts the target value by calculating a weighted average of the target values of the nearest neighbours. In classification tasks, instead of a simple majority vote, the class of the query example is determined by a weighted ballot, where the algorithm multiplies each neighbour's vote by its corresponding weight.

2.5 Editing Algorithms Overview

This section discusses methods for cleaning datasets for kNN predictions, focusing on removing noisy (error-filled) or unrepresentative examples (outliers or atypical cases). It is important to distinguish between two main goals of kNN editing algorithms: removing noise and eliminating redundancy. While redundancy reduction is about discarding surplus data to improve efficiency, the algorithms we focused on are primarily concerned with noise reduction.

2.5.1 Motivation

Since kNN relies on some distance/similarity function, a significant challenge arises when the elements of the dataset (**X**) are noisy or unrepresentative. Such examples can drastically skew kNN predictions, making them less accurate and reliable; this can be particularly problematic for tasks with relatively low acceptable error rates.

Researchers have developed various editing algorithms to mitigate the risks posed by poor data quality. These algorithms aim to identify and remove problematic examples from **X**, enhancing the dataset's overall reliability.

2.5.2 Repeated Edited Nearest Neighbours

One editing algorithm we will later implement is Repeated Edited Nearest Neighbours (RENN) as first proposed by Tomek [21].

Basic Concept

RENN operates by repeatedly applying the Edited Nearest Neighbour (ENN) rule. The ENN rule involves examining each example in the dataset and removing it if its class label differs from the majority class of its k-nearest neighbours.

Repeated Applications of ENN

Unlike the basic ENN, which is applied once, RENN iteratively uses the ENN rule until the algorithm can not remove any more examples (usually determined by some arbitrarily high max iterations property). This repeated application allows for deeper dataset cleaning, ensuring that only the most representative and reliable examples remain.

2.5.3 Blame Based Noise Reduction

One editing algorithm we will later implement is Blame Based Noise Reduction (BBNR) as first proposed by Pasquier et al. [16]; "BBNRv2" in particular.

Basic Concept

BBNR operates on the principle of "blaming" examples frequently involved in incorrect kNN predictions. While RENN focuses on which examples kNN has misclassified, BBNR instead highlights which examples are helping to misclassify their neighbours. The algorithm assesses the impact of each data point on the overall prediction accuracy by giving it a "liability" score, i.e. the number of times the example was in the incorrect majority when misclassifying other examples in the training dataset. It then removes the most problematic examples one by one and reinserts them if the removal has resulted in previously correctly classified examples now having erroneous predictions.

Other Important Concepts

- **Coverage Set**: The Coverage Set of an example $\mathbf{x}$ are all the other examples that a contributed to the correct classification of, i.e. examples that kNN classified correctly and a was in the voting majority during classification.
- **Liability Set**: The Liability Set of an example $\mathbf{x}$ are all the other examples that a contributed to the incorrect classification of, i.e. examples that kNN classified incorrectly and a was in the voting majority during classification.
- **Disimilarity Set**: The Disimilarity Set of an example $\mathbf{x}$ is its k nearest neighbours that contributed to its misclassification, i.e. kNN classified a incorrectly, and the set consists of its neighbours that were in the voting majority during its classification.

The above three concepts comprise what Pasquier et al. [16] call BBNR's "case-base competence model".

2.6 Gaps in Existing Literature

Despite the considerable advancements in editing algorithms and case-based reasoning for tackling noise and redundancy in datasets used for kNN tasks, the literature reveals notable gaps that impede progress. This section delves into these identified deficiencies, highlighting the limited scope of editing algorithms, the challenge of handling mixed-quality neighbours, and the integration between reliability and similarity measures, thus setting the stage for our contributions to bridging these gaps.

2.6.1 Limited Scope of Editing Algorithms

A notable limitation in the extant research landscape is the predominant focus of editing algorithms on classification tasks. Traditionally, these algorithms tackle datasets designed to predict categorical outcomes, such as determining whether a student will pass or fail a course. This emphasis has inadvertently created a research gap concerning their efficacy and relevance in non-classification contexts. The prevalent bias towards classification is likely due to the relative ease of defining “agreement” in such tasks. In classification, agreement can be straightforwardly determined by whether the class labels of two examples match. However, this clarity is absent in regression tasks, where establishing a clear “dividing line” for agreement proves more challenging.

Although not the original objective of this project, in response to this gap, our research ventured into adapting existing editing algorithms for regression tasks, aiming to bridge the divide. After designing and implementing our modified versions of popular editing algorithms, we encountered a parallel line of inquiry by Kordos & Blachnik [12] - this paper adapts the ENN rule for regression tasks. The work of Martín et al. [13] also expanded upon this ENN adaptation to other noise filters (including RENN and BBNR). The approach in both papers closely mirrors our own, and we were unfamiliar with their research until we had finished our design and implementation. Although this means that our work is less original than initially thought, our implementation introduces unique modifications that distinguish it from their work. We highlight these novel additions to the research landscape in later sections.

2.6.2 Handling Mixed Quality Neighbours

While there is research on computing the reliability of examples in datasets, as acknowledged by Parsodkar et al. [15], there is a notable need to improve estimating the reliability of a case within a neighbourhood of mixed quality.

Existing methods struggle to provide robust reliability estimates in scenarios where reliable and unreliable cases are intermingled. This limitation hinders datasets' effective maintenance and quality assessment, particularly in complex or nuanced domains.

2.6.3 Integration Between Reliability and Similarity

Another crucial gap is the need for more research on effectively combining reliability with similarity for making kNN predictions. Current literature on Case-Based Reasoning offers ways of computing the reliability of examples. However, it does not extend to integrating these reliability measures with similarity/distance scores in kNN when actively making predictions. Such an integration could be more efficient than the current divergent model and lead to more accurate predictions.

Chapter 3

Core Design and Implementation

This chapter delineates the comprehensive architecture of the machine learning harness, the selection and utilisation of third-party Python packages, and the intricacies of implementing the kNN algorithm, including preprocessing steps and model evaluation techniques. The core design lays the groundwork for efficiently implementing and evaluating kNN variants (both existing and novel) through a blend of automated workflows, object-oriented programming principles, and meticulous algorithmic implementations.

3.1 Programming Language Choice

This project was implemented using Python due to its robust machine learning ecosystem; while there are other languages that fit this description e.g. R, as highlighted by Raschka et al. [19], Python remains the most popular for machine learning and data science (likely meaning our work will be more accessible) and is the language we were most familiar with.

3.2 Python Package Imports

The following are the third party Python packages we made use of and how:

- **Pandas (PD):** We used PD for dataset loading, preprocessing, and manipulation.

- **Numpy (NP):** We used NP for array manipulations and numerical computations.
- **Scikit-learn (sklearn):** We used sklearn for dataset preprocessing, model selection, and error estimation. The choice of machine learning library was important, and sklearn has been a well-established choice since it was first proposed by Pedregosa et al. [17] in 2012.
- **TQDM:** This package provided progress bars for loops.
- **Imbalanced-learn (imblearn):** We used imblearn for implementing RENN.
- **SciPy:** We used SciPy's implementation of the T-Test and Friedman statistical significance tests.
- **Scikit-post hocs (SP):** We used SP's implementation of the Nemenyi statistical significance test.

3.3 Machine Learning Harness Design

This section outlines the design and requirements for a machine learning harness developed to facilitate the evaluation of novel noise and redundancy reduction techniques using kNN algorithms across various datasets. It emphasises the necessity for automation in the machine learning workflow and the importance of extendability and overridability in the software architecture to accommodate the diverse preprocessing needs of different known methods. It offers insights into the harness's automated workflow and object-oriented programming approach for flexibility.

3.3.1 Harness Requirements

Since this project required evaluating novel noise and redundancy reduction techniques on many datasets and comparing their performance against several editing-based methods, we identified the need for a robust machine learning harness at the heart of the project, i.e. a software component that would make it easy to manage, organise, and facilitate the evaluation of a variety of divergent kNN methods on a variety of datasets. We then established these two critical requirements for the harness:

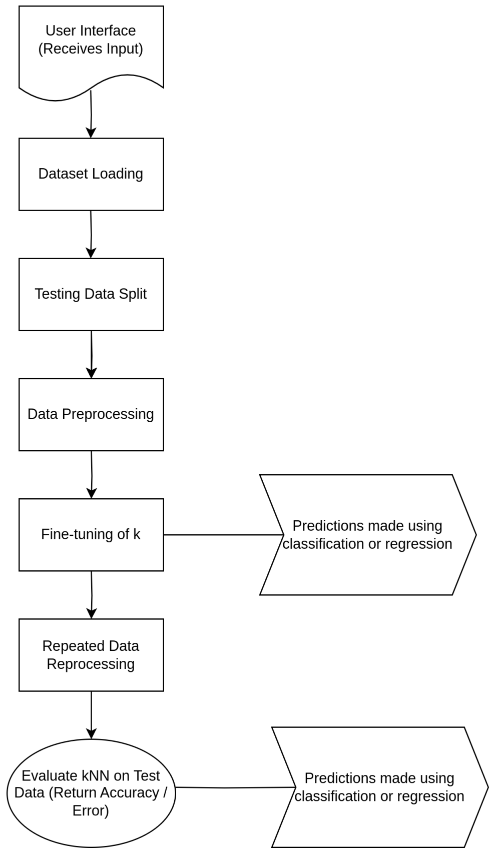
User Interface (Receives Input)
Dataset Loading
Testing Data Split
Data Preprocessing
Fine-tuning of k
Predictions made using classification or regression
Repeated Data Reprocessing
Evaluate kNN on Test Data (Return Accuracy / Error)
Predictions made using classification or regression

Figure 3.1: The workflow the harness automates.

- **Requirement 1:** It would need to automate much of the standard machine-learning workflow to facilitate the iterative nature of the experiments.
- **Requirement 2:** Its core functionality would need to be extendable and overridable, i.e. kNN instances using editing algorithms will have a different preprocessing step than those without.

3.3.2 Automated Machine Learning Workflow

This section outlines the design for requirement 1 of the harness. The harness design needed (due to the number of experiments involved) to mean that a user could interact with it at a single main entry point and get an accuracy or error estimation as a return, and it handles all the between steps.

Each kNN evaluation consisted of the same workflow depicted in Figure 3.1. Later sections detail each step's particulars (design and implementation). The following is a brief overview:

1. **User Interface:** A single entry point to which the user passes the task and dataset details of which they want to evaluate the performance.

2. **Dataset Loading:** The dataset is loaded into a format that is easier to manipulate, process, and make predictions with.

3. **Testing Data Split:** A portion of the data is reserved for the final evaluation to prevent data leakage.

4. **Data Preprocessing:** Any manipulation of the dataset to ensure its high quality and suitability for training, e.g. scaling; this is also where the editing algorithms will be applied.

5. **Fine-tuning of Hyperparameters:** Primarily focusing on k, this step systematically identifies the optimal hyperparameter values using a validation set. It is adaptable for fine-tuning additional hyperparameters beyond k.

6. **Repeated Data Preprocessing:** We repeat step 3 here before evaluating test data performance; the rationale behind this becomes more apparent in later design sections.

7. **Evaluate kNN on Test Data:** Return the accuracy or error estimate of the performance of kNN on the reserved testing data.

Both steps **5** and **7** involve making predictions using kNN **regression** or **classification**.

The user passes the following as input to the user interface to run an evaluation:

- What task do they wish to evaluate, i.e. regression or classification.

- The location of their dataset on the file system.

- The name of the dataset column they are predicting.
- Any characters used to represent missing data.

They then receive an accuracy score if they ran a classification evaluation and an error estimate if they ran a regression evaluation.

3.3.3 Extendability and Overridability

In order to minimise code redundancy, all kNN variants will share some common infrastructure, e.g. the dataset splitting functionality will be the same for all. However, at the same time, there will be divergences, e.g. for kNN variants with an editing algorithm step - the preprocessing step would be more elaborate than standard kNN. In these cases, a variant must leverage the common infrastructure while overriding the specific steps where it diverges.

Given this, we arrived at an Object Oriented Programming (OOP) design with a harness super class (the standard kNN implementation), which all variants can inherit from and extend. With an OOP approach, variants could be based on this superclass and modify its functionality (as well as add entirely new functionality). This design made it trivial to have this overridability we desired during implementation.

3.4 KNNHarness Superclass Implementation

This section outlines the implementation of the super class in our OOP design and will focus just on the private attributes and public methods of the class.

3.4.1 Class Definition

KNNHarness is the class that facilitates standard kNN regression or classification evaluation. Once initialised, it can load the dataset, preprocess the data, fine-tune K, and return the accuracy (in the case of a classifier) or mean absolute error (MAE) in the case of a regressor, all from a single *evaluate* method call. The design for this class is presented visually in Figure 3.2.

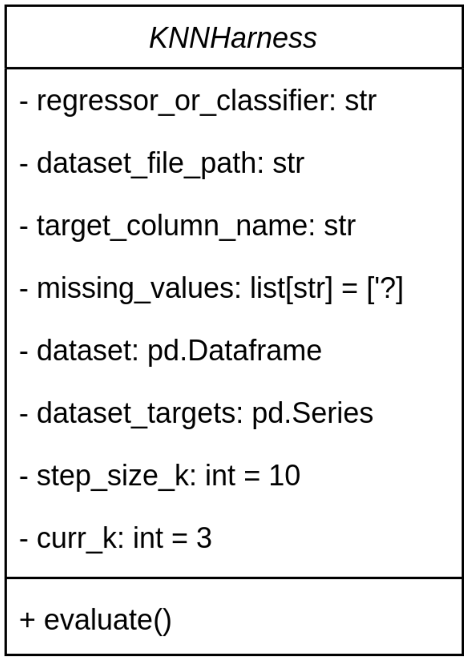
KNNHarness
- regressor_or_classifier: str
- dataset_file_path: str
- target_column_name: str
- missing_values: list[str] = ['?]
- dataset: pd.Dataframe
- dataset_targets: pd.Series
- step_size_k: int = 10
- curr_k: int = 3
+ evaluate()

Figure 3.2: UML diagram of *KNNHarness* class design

3.4.2 Class Attributes

All attributes are private, and the harness encapsulates them using getters, setters, and Python's @*property* decorator.

- **regressor_or_classifier(str):** This string denotes what task the harness is evaluating; it can either be "classifier" or "regressor".
- **dataset_file_path(str):** This string denotes the file path to the dataset that the harness should use to evaluate the prediction task.
- **target_column_name(str):** This string denotes the column's name the regressor or classifier should be predicting, i.e. the target value for regression or the target class for classification.
- **missing_values(list[str]):** Each string in this list represents a placeholder used in the dataset to denote a missing value.

- **dataset(pd.Dataframe):** This is the dataset loaded as a PD Dataframe without its target column (the entire dataset pre-split).
- **dataset_targets(pd.Series):** This is the target column of the dataset loaded as a PD Series (this Series has all target values pre-split).
- **step_size_k(int):** The current step size that is used when expanding the search space for k during fine-tuning.
- **curr_k(int):** This integer represents the value for k currently being considered (or the k to be used for the final evaluation if fine-tuning is complete).

3.4.3 Class Initialisation

Initialising the class and calling evaluation automates the user interface step of the workflow. To initialise a $KNNHarness$, one invokes the class constructor and passes values for $regressor\_or\_classifier$, $dataset\_file\_path$, $target\_column\_name$, and $missing\_values$. On initialisation, the constructor loads the dataset; see Dataset Loading Design and Dataset Loading Implementation.

3.5 Dataset Loading Design

There were four considerations when it came to designing the dataset-loading component:

- **Data Ingestion:** The component needed to load the dataset into a data structure that would allow the programmatic manipulation and analysis of the data.
- **Handling Missing Values:** The component needed to delete examples with features with missing values to ensure data consistency.
- **Segregating Target Values:** The component needed to segregate the column containing the target values of the regression or classification into a separate data structure (with the exact requirements described in consideration one) as the model evaluation would be too optimistic if the target values themselves were a feature.

- **Target Label Encoding for Classification:** String comparisons are computationally expensive compared to other datatypes such as integers and floats, and the target column for classification tasks is likely to consist of strings - in such cases, each unique class would need to be converted to a corresponding integer ranging from 1 to n, with n being the total number of distinct categories.

3.6 Dataset Loading Implementation

When the class constructor runs, it calls the *load_dataset* method, which uses PD's *read_csv* function to read the dataset from the file at $self.dataset\_file-\_path$ as a PD DataFrame. The process replaces any value matching those in $self.missing\_values$ with PD's *nan* value - any row with a *nan* is then dropped. Dropping examples with missing values ensures the integrity and reliability of the dataset, facilitating models' learning process by focusing on complete and accurate information. The function copies the column with the name $self.target\_column$ to a PD Series and then drops it from the DataFrame. The process assigns this DataFrame to $self.dataset$ and the Series to $self.dataset\_targets$. Finally, if the user specifies they wish to evaluate a classifier, then $self.dataset\_targets$ is encoded using sklearn's LabelEncoder.

3.7 Test Data Splitting Design

To evaluate the performance of a given kNN task accurately, we had to reserve some of the data for testing. This data we segregate entirely from the training and validation data - the combination of which we refer to as the "dev" dataset or "dev" data.

There were several ways we could arrive at this split. One was 'hold-out', i.e. simpling allocating a given percentage of the data at random to the training set and "holding it out" from the dev data. The main advantage of this method is its simplicity and efficiency. However, this comes at a cost to the potential accuracy of the evaluation. The error estimate or accuracy figure may be over or under-optimistic if the portion of the data we "hold out" is more straightforward or more challenging to predict than is typical for the problem.

To overcome the limitations of hold-out, we opted for k-Fold Cross Validation (KFCV), where the whole dataset splits into k equally sized “folds”, and each fold takes a turn at being the testing data. Note that the two uses of the variable k, the number of neighbours and number of folds, are unrelated. The remaining folds comprise the dev data for each iteration. Then, rather than relying on a single evaluation score, we take the mean of the error or accuracy as the final score for the task on the dataset. This design choice, of course, comes at the disadvantage of having to do k times the amount of computational work (at this high level). However, as highlighted by Yadav & Shukla [24], cross validated models often produce higher quality predictions than those trained and validated using holdout. In our case, we chose 5 for k, therefore, we have approximately five times the amount of computational work as a simple hold-out. However, given that the datasets we chose for our experiments were relatively small, we thought this trade-off was worth it.

3.8 Test Data Splitting Implementation

To split the dataset into dev and test inside the evaluate method, we create an instance of sklearn’s KFold (or StratifiedKFold for classification) class with *n_splits* set to 5, $shuffle$ set to True, and *random_state* set to 42. The class instance will perform KFCV with k set to 5 and shuffle the data prior to arriving at the fold-split; defining a *random_state* also means that our kNN experiments will be reproducible, provided the user uses the same *random_state* as the seed for the shuffle and split.

Then, we will call the instance’s split method and iterate over it; one each iteration, the method will provide *test_idx* and *dev_idx* - two NP arrays with the indices of the testing data fold and the combined remaining folds, respectively. For each iteration, the method initialises two new DataFrames and two new Series populated by copying the examples specified in *test_idx* and *dev_idx* from $self.dataset$ and $self.dataset\_targets$ - we call these *dev_data*, *test_data*; and $dev\_targets$, $test\_targets$, respectively, which concludes the segregation of the test data from the dev data as all subsequent operations involve these new DataFrames and Series rather than the class instance variables.

3.9 Dataset Preprocessing

At several points, the harness will need to "preprocess" the data, i.e. turn categorical data into numerical data we can plug into the kNN distance calculations or scale the data to ensure that no feature dominates said calculations. This preprocessing is done for the dev, validation data/training (see K-Value Selection), and testing datasets.

3.9.1 Categorical and Numerical Data Preprocessing

Design

The process involves identifying and handling categorical and numerical data within a dataset. Categorical data can often appear as strings or discrete symbols in datasets; we must encode this data appropriately for kNN, which can only handle numbers. Numerical data, on the other hand, might require normaliation or other transformations. The design here focuses on automating the identification of these data types and applying the appropriate transformations.

Implementation

In the $KNNHarness$' *_preprocess_dataset* method, the code iterates over the dataset's columns to identify categorical columns, either marked explicitly with "(cat)" or determined through a $try - except$ block that checks for columns that we cannot convert to numeric types without raising an exception. These identified categorical columns are then one-hot-encoded using-*pd.get_dummies*, meaning each unique value for a given nominal column becomes a new binary feature (zero indicating absence and one meaning presence). The method has already encoded any numerical columns as such at this stage in the cases where the $try-except$ block did not raise an exception.

3.9.2 Data Scaling

Design

Scaling numerical data ensures that each feature equally influences the model's performance. This process involves normalising data to accommodate varying ranges and distributions. Importantly, scaling parameters, such as the mean and standard deviation, should be derived from a portion of the dataset (e.g., the development set) and then applied to both this portion and the testing set. Our approach involves mean centring, subtracting the feature's mean across the dataset from each instance's value and dividing by the standard deviation.

Implementation

We implemented the scaling process in the *_preprocess_dataset* method. The method uses *StandardScaler* (which enforces standardisation as described in the design above) from sklearn. A "scaler" is taken as an argument that defaults to *None* if the user did not pass a scaler, i.e. they called the method on the dev set or training set. The critical steps in the scaling process are as follows:

- **Initialisation of Scaler:** If the function call does not include a *scaler*, it implies that the dataset is either for development or training. In this case, the method creates a new *StandardScaler* instance.
- **Fitting and Transforming the Data:** The scaler is applied to the data using the *fit_transform* method for a development or training dataset. This method calculates the mean and standard deviation for each feature in the dataset and scales the data accordingly.
- **Transforming with Existing Scaler:** If the function call included a pre-existing scaler, this indicates that the dataset is for testing or validation. The dataset is then scaled using the transform method of the scaler, which applies the same transformation as was applied to the training data.
- **Handling of Categorical Data:** Before scaling, the method identifies and one-hot-encodes categorical columns; this ensures that only numerical data is subjected to scaling, while categorical data is processed separately.
- **Returning the Scaled Data:** The function returns the scaled dataset, the dev/training targets as a NP array, the columns of the training data, and the scaler.

3.10 kNN Implementation

This section will focus on the specific implementation details of our project's k-Nearest Neighbours (kNN) algorithm, building upon the foundational understanding of kNN covered in Chapter 2. We divided our implementation of kNN into two distinct methods: *_knn_regressor* and *_knn_classifier*. Both elements share the core method _get_k_nearest_neighbours for identifying the nearest neighbours within the dataset. Even where implementation diverges between the two, we aimed for modularity, e.g.
_get_most_common_class is its own decoupled method in the classification workflow. This modularity ensures a cohesive and efficient approach to both regression and classification tasks using kNN.

Key Implementation Features:

- **Modular Design:** We implemented separate methods for regression and classification, sharing standard methods. This design enhances code reusability and clarity.
- **Efficient Distance Calculation:** For neighbour identification, we use vectorised operations in NP to compute Euclidean distances, ensuring efficient computation even for larger datasets.
- **Dynamic Neighbour Selection:** The number of neighbours, k, is adjustable, allowing fine-tuning for different datasets.
- **Handling Class Ties:** In classification, when a tie occurs in the mode of the class labels among the neighbours, we resolve it by randomly selecting among the tied classes. This avoids potential bias by adding an element of randomness to the classification in tie scenarios.
- **Return Types:** The regressor returns a float representing the mean target value of the nearest neighbours. At the same time, the classifier returns the most common class label (which is also numeric, thanks to label encoding).

3.11 Model Error Estimation

As shown in Figure 3.1, the last step in the automated machine learning workflow is to "Evaluate kNN on Test Data", i.e. return a final "score" for the model's performance on a given experimental run; this also needs to be done during the "fine-tuning of k" step shown in the exact figure (except its an evaluation on validation data rather than test data) and discussed in a later section. To evaluate the performance of the kNN variants on various datasets, we needed to provide a quantitative measure of each model's accuracy in making predictions.

3.11.1 Regression

Design

The design step for regression evaluation primarily revolved around choosing an error estimation metric. We considered several, but most notably:

- **Mean Squared Error (MSE)**: Measures the average squared difference between actual and predicted values. However, this is sensitive to outliers.

- **Root Mean Squared Error (RMSE)**: Square root of MSE, more sensitive to outliers than Mean Absolute Error.

- **Mean Absolute Error (MAE)**: Calculates the average of absolute differences between actual and predicted values. Less sensitive to outliers, providing a more robust error measure. It is often considered the best for many applications due to its simplicity and interpretability.

Ultimately, we opted for MAE for our error estimate for regression task. As Willmott & Matsuura [23] compellingly argue (in their case in the context of climate models), it is far easier to interpret, e.g. if MAE is 90,000 (USD) for predicting the price of the house, then we immediately can understand that our model is about 90,000 USD off the mark on average - whereas this leap from intuition is far more challenging with error estimates that use squaring. Where m is the number of examples, abs gives the absolute value of an expression, $h(\mathbf{z})$ gives the predicted value of the example $\mathbf{z}$, $\mathbf{x}$ is a vector of examples, and $\mathbf{Y}$ is a vector of target values for each example, MAE can be concisely defined in Equation 3.1.

$$MAE = \frac{1}{m}\sum_{i=1}^{m} abs(h(\mathbf{x}_i) - \mathbf{Y}_i) \tag{3.1}$$

Implementation

The *_get_mae_of_knn_regressor* function calculates the MAE for the kNN regressor. The following is a concise overview of its implementation.

Function Parameters:

- *k*: Number of nearest neighbours to consider for regression.
- *training_dataset*: The dataset from which the function can get the neighbours.
- *testing_dataset*: The dataset for which the function must calculate the MAE.
- *training_targets*: Target values corresponding to the training dataset.
- *testing_targets*: Target values corresponding to the testing dataset.

Prediction Generation:

- The function iterates over each example in the *testing_dataset*.
- For each example, the function *_knn_regressor* predicts the value based on the *k* nearest neighbours in the training dataset.

MAE Calculation:

- We store the predictions for each test example in a list.
- This list is converted into a pandas *DataFrame* for compatibility with the MAE calculation function.
- The *mean_absolute_error* function from *sklearn* computes the MAE, comparing the predicted values against the actual values in *testing_targets*.

Return Value:

The function returns the computed MAE, providing a quantitative measure of the model's average prediction error magnitude.

3.11.2 Classification

Design

The design step for classification evaluation primarily revolved around choosing an error estimation metric. Accuracy was the obvious choice, defined as the ratio of correct label predictions to total label predictions. We chose it as it provides a direct and intuitive understanding of a classifier's performance and is widespread in its use in the literature.

Implementation

The *_get_accuracy_of_knn_classifier* function calculates the accuracy of a kNN classifier. The following is a concise overview of its implementation.

Function Parameters:

These are precisely the same as the "Function Parameters" in the "Implementation" of "Regression" error estimation.

Prediction Generation:

- The function iteratively processes each example in the testing dataset.
- For each test example, it applies the *_knn_classifier* method. This method predicts the class based on the k nearest neighbours in the training dataset.

Accuracy Calculation:

- We accumulate the predicted classes for each test instance in a list named predictions.
- The *accuracy_score* function, from *sklearn*, is then used to calculate the accuracy. It compares the predicted class labels in predictions with the actual labels in *testing_targets*.

Return Value:

The function returns the computed accuracy, providing a quantitative measure of the model's average prediction error magnitude.

3.12 k Hyperparameter Tuning

This section details how we automate identifying the optimal number of nearest neighbours that the model should consider for making predictions on a given dataset, i.e. the third last step in Figure 3.1. This tuning is crucial for balancing the model's bias and variance, ensuring that the results of our experiments are optimum (or in proximity).

3.12.1 Design

Initial Candidate k Values Selection

We required a component to establish a starting point for tuning the k value in kNN, i.e. an initial list of candidate values to evaluate. The square root of the number of examples in the dataset is often used as a heuristic to determine an initial k value [9]. We realised this heuristic could be a starting "pivot" around which we could populate an initial list of "guesses" for the optimum k. This value could be adjusted to be odd (to help prevent ties). Then, a dynamic "step size" could be used to exhaustively search and evaluate the candidate k values surrounding this pivot.

Best k Value Identification

We required a component that would find the k value that yields the best performance in terms of accuracy (classification) or MAE; it would evaluate each k in the current list of candidates (the first list being generated by the "initial candidates k value selection" component). 5-fold cross-validation would be used to assess the performance of each candidate k value on different subsets of the data which would ensure a robust evaluation. This component would keep track of the best average score obtained so far, updating it when it finds a better performing k value. Eventually this component would halt when it meets some criteria defined in the implmentation that indicates the promising candidate k value search space has been exhausted.

Search Space Expansion

Here, we detail a novel approach for dynamically adjusting the search space for the k value in kNN, which is essential for optimising model performance. The strategy adaptively expands the search boundary based on insights gained from previous iterations. The algorithm intelligently expands the search space for k: if the current best k lies at the edge of the explored range, the search expands towards the direction with unexplored potential. This process involves adjusting the candidate k values, with the aim of honing in on the optimal k value with greater precision. This method stands out for its dynamic adaptation to the search context, a novel aspect that differentiates it from static search strategies. Algorithm 2 illustrates the search space expansion logic (it is slightly abridged so as to not be verbose).

Algorithm 2 Dynamic Search Space Expansion for k in k-NN

```
Require: $candidate\_k\_values$, $curr\_best\_k$, $tried\_k$, $step$
Ensure: New list of $candidate\_k\_values$
  Initialise $new\_candidates$ as empty list
  if $curr\_best\_k$ is not at the edge of $candidate\_k\_values$ then
      Reduce $step$ size by 75% for finer adjustment
      $step \leftarrow max(step, 2)$ to ensure meaningful step size
      Generate $new\_candidates$ around $curr\_best\_k$ using $step$
  else
      Adjust $step$ to expand search "leftwards" or "rightwards"
      Populate $new\_candidates$ based on new $step$
  end if
  Filter $new\_candidates$ to only include untried, odd, positive $k$
  $new\_candidates \leftarrow sorted(set(new\_candidates))$
  return $new\_candidates$
```

This approach ensures that the optimal k search is both efficient and exhaustive, adjusting dynamically to the findings as the search progresses. The emphasis on expanding the search space based on the position of the current best k makes this method innovative, allowing for a more nuanced exploration of possible k values.

Stochastic Search Space Expansion

This subsection introduces an innovative stochastic method for expanding the search space of the k value in k-NN and works similarly to the deterministic approach outlined previously with some important caveats. Why it proved necessary to have two methods of expanding the search space becomes apparent in Chapter 7. Unlike the deterministic method, which systematically explores the search space based on fixed steps, the stochastic method introduces randomness in the selection of candidate k values, aimed at leveraging the majority of the benefits of the original while reducing the number of candidates that we test at the validation step. Algorithm 3 illustrates the logic behind the stochastic search space expansion, emphasising the selection and evaluation process of new candidate k values.

Algorithm 3 Stochastic Search Space Expansion for k in k-NN

```
Require: candidate_k_values, curr_best_k, tried_k, step
Ensure: New list of candidate_k_values
  Initialise new_candidates as empty list
  Determine search direction based on curr_best_k position
  Reduce step size by 75% for finer adjustment
  $step \leftarrow max(step, 2)$ to ensure meaningful exploration
  if Search direction is “left” or “both” then
      Generate and filter leftward candidates.
      Add one randomly selected leftward candidate to new_candidates.
  end if
  if Search direction is “right” or “both” then
      Generate and filter rightward candidates.
      Add one randomly selected rightward candidate to new_candidates.
  end if
  $new\_candidates \leftarrow sorted(set(new\_candidates))$
  return new_candidates
```

3.12.2 Implementation

Initial Candidate k Values Selection

The function *_get_initial_candidate_k_values* returns a list of integers; it takes the number of examples as input and computes its square root; the result is the starting “pivot” described in the design. Depending on the magnitude of this initial k, different step sizes (1, 10, 100, 1000) are determined for exploring neighbouring k values. The function generates a list of candidate k values, taking steps backwards and forward from the initial k

until it produces a list of 5 candidates. The function then filters this list before returning, ensuring all k values are positive, odd, and unique.

Best k Value Identification

The *_get_best_k* function returns the optimum k for a given dataset; it creates an instance of sklearn's $KFold$ (or $StratifiedKFold$ for classification). This $KFold$ object then splits the *dev_data* (see Test Data Splitting Implementation) into training and validation datasets. Each candidate k is evaluated by training the model on the training dataset and assessing its performance on the validation set by calling *_get_accuracy_of_knn_classifier* or *_get_mae_of_knn_regressor* (depending on the task). Before the evaluation, though, the datasets are preprocessed (see Data Preprocessing). The best k value and corresponding score are updated if a candidate performs better. The function recursively calls itself with new candidate k values generated by calling either *expand_k_search_space_deterministic* or *expand_k_search_space _stochastic* (see following section). The "halt" condition occurs if the list returned by these functions is empty or consists of just the last best k value recorded, indicating that the promising search space has been exhausted.

Search Space Expansion

This section details implementing two methods to expand the search space for k: deterministic and stochastic. Both methods aim to refine the search for the optimal k value by exploring the space more efficiently based on the current best k.

Deterministic Expansion

The *_expand_k_search_space* function takes the previous list of *candidate_k _values* and returns a new list of promising candidates. It also takes the *curr_best_k* as input to inform how it expands the search space. It works similarly to *_get_initial_candidate_k_values* in that it will expand rightward or leftward using the current *step_size_k* (class attribute; see UML diagram in Figure 3.2) if the *curr_best_k* was at the start or end of the *candidate_k_values*. The critical difference is that when it is not an edge case, it will reduce the *step_size_k class* attribute by 75%, i.e. multiply it by 0.25 (and then round it up as there cannot be a decimal step size). It then produces and returns a list of 5 new candidates in the same way as *_get_initial_candidate_k_values*, i.e. stepping forward and back with this step size and applying some filters. This dynamic adjustment achieves the precision described in the design section for this component.

Stochastic Expansion

The *expand_k_search_space_stochastic* function introduces a randomised approach to expanding the search space. It adjusts the *step_size_k* similarly to the deterministic approach but selects new candidates stochastic. Depending on the position of *curr_best_k* within the current search space, it generates potential candidates on both or one side of *curr_best_k*. Then, it randomly selects one of these potential candidates from each side (or a single side if applicable), ensuring they meet the odd, positive, and untried criteria. This approach aims to be dynamic like the deterministic method but less exhaustive to reduce runtime.

3.13 Weighted kNN

In this section we discuss the design and implementation of weighted kNN - a variant of standard kNN we overviewed in Chapter 2.

3.13.1 Weighted kNN Design

Algorithm 4 depicts the algorithm's design as pseudocode. The algorithm is the same as Algorithm 1 until (and including) extracting the target values/class labels from the k-nearest neighbours. So, we only depict the divergence after that.

3.13.2 Weighted kNN Implementation

The $WeightedKNNHarness$ class inherits from $KNNHarness$ and incorporates weighted kNN for classification and regression. This implementation adjusts the influence of each neighbour based on its distance to the query example. The subclass overrides two methods of the superclass: $\_knn\_classifier$ and $\_knn\_regressor$. These methods operate like the originals, but we highlight key differences here.

Both methods calculate Euclidean distances between the query example and

Algorithm 4 Weighted k-Nearest Neighbours

```
Same as Algorithm 1 until (and including) extracting target values/labels.
$weights \leftarrow$ compute inverse of the distances for the selected $k$ elements.
if regression task then
    Compute weighted mean of the $k$ target values using $weights$.
    return this weighted mean as the prediction for $\mathbf{x}$.
else if classification task then
    Calculate weighted frequency for each class among the $k$ nearest neighbours using $weights$.
    Determine the class with the highest weighted frequency.
    return this class as the prediction for $\mathbf{x}$.
end if
```

each other example. The method adds a tiny constant ϵ to all distances to avoid dividing by zero exceptions. Weights are inversely proportional to distances, emphasising closer neighbours over farther ones. For classification, a Python dictionary accumulates the sum of weights for each class observed in the k-nearest neighbours. A simple array division operation (1 / distances) suffices for regression. For classification, the class with the highest accumulated weight is the prediction. Python's max function with a custom key (lambda function) efficiently accomplishes this. For regression, instead of returning a simple mean, it calculates a weighted mean of the target values of the k-nearest neighbours using NP array multiplication and summing.

Chapter 4

Core Optimisations

The necessity for optimisation emerged prominently during the development of our machine learning harness, specifically when addressing the computational bottlenecks encountered with the *get_best_k* function. This function, integral to our k-nearest neighbours (kNN) implementation, demonstrated significant performance limitations due to its usage in a nested k-fold cross-validation process. This setup required the function to be called multiple times – a minimum of 25 times (five outer folds $\times$ five inner folds) – each time preprocessing ran, and further compounded by its recursive nature during the exhaustive k search feature in the harness. Such an approach proved CPU-intensive, prompting a thorough investigation into potential optimisation strategies.

4.1 Encountered Challenges

Initial attempts to enhance performance through various strategies encountered several setbacks.

4.1.1 Attempt with CuPy

Utilising CuPy, a GPU-accelerated library for numerical computations, seemed a promising avenue for parallelising operations and reducing computation time. However, the anticipated performance improvements did not materialise due to significant overhead in transferring data between the CPU and GPU, alongside compatibility issues with certain parts of our data processing pipeline.

4.1.2 Caching Strategy

Caching distances or similarity measures between examples presented a logical optimisation step, potentially reducing redundant computations during the preprocessing steps of cross-validation. Unfortunately, this approach was quickly invalidated - the preprocessing step, essential for each fold, involved re-scaling of data, leading to a situation where cached distances became obsolete after each fold's preprocessing, negating any benefit from caching.

4.1.3 OpenBLAS Configuration

We tried modifying our OpenBLAS configuration to enhance matrix operations efficiency - however, this change yielded no noticeable performance improvements, likely due to the existing configuration already being optimised for our hardware setup.

4.1.4 Threading Limitations

Python's Global Interpreter Lock (GIL) is a well-known limitation for multi-threading in CPU-bound tasks [20]. It prevents concurrent execution of multiple threads within a single Python process, which made threading an unviable option for our optimisation needs.

4.2 Multiprocessing Success

The significant leap forward in our optimisation efforts came through the strategic employment of multiprocessing. This optimisation technique proved instrumental, especially given the inherently embarrassingly parallelisable nature of our problem [10]. Specifically, each outer k-fold validation process could be isolated and executed independently across multiple CPU cores, resulting in substantial performance enhancements. This method effectively sidesteps the limitations imposed by Python's Global Interpreter Lock (GIL) by initiating separate Python processes for each fold instead of threading. Each process operates with its own Python interpreter and memory space. It facilitates accurate parallel computation and significantly speeds up our exhaustive search within the *get_best_k* function during nested k-fold cross-validation.

4.2.1 Implementation Details

The Python *multiprocessing* module facilitated the distribution of computation across available CPU cores - this was particularly effective for our nested k-fold cross-validation process, where each outer fold could be processed independently in parallel.

Chapter 5

Editing Algorithms

In this chapter, we will discuss the design and implementations of the various editing algorithm kNN variants discussed in Chapter 2.

5.1 RENN Design and Implementation

In this section, we discuss our RENN design and implementation.

5.1.1 Standard RENN Design

Algorithm 5 depicts RENN's pseudocode design as described by Tomek [21] that we then simplified. This is the "standard" RENN design i.e. only suited to classification tasks.

5.1.2 Task-Agnostic RENN Design

This section introduces a versatile design for the RENN algorithm, making it applicable to classification and regression tasks with adjustable hyperparameters for enhanced flexibility.

Agrees Function

The "agrees" function, a cornerstone of our task-agnostic RENN design, determines whether an example is in harmony with its neighbours. For classification, an example agrees if it shares the same label with most of its

Algorithm 5 Repeated Edited Nearest Neighbours

```
Require: a dataset of examples X.
  Classify all examples using kNN as per Algorithm 1.
  while there are misclassified examples in X do
      Remove all misclassified examples from X.
      Classify all examples using kNN as per Algorithm 1.
  end while
```

neighbours. For regression, where outputs are continuous, we propose two nuanced approaches to agreement:

- **Standard Deviation of Neighbourhood (SDN):** Agreement is based on the standard deviation within an example's neighbourhood. An example is considered in agreement if the absolute difference between its value and its neighbour's value is within SDN times a tolerance parameter, θ. This local perspective assesses agreement within the immediate vicinity of an example.

- **Standard Deviation of the Whole Training Set (SDW):** This global approach assesses agreement based on the entire training set's variability. An example agrees if the absolute difference between its value and its neighbour's value is within SDW times θ - this ensures consistency with the overall dataset characteristics.

The choice of SDN or SDW is configurable, allowing for strategic flexibility between focusing on local consistency or global dataset alignment. For classification, the agrees function remains non-configurable.

Selection Function

The "selection" function determines whether an example should remain in the dataset based on its agreement with neighbours. It offers the choice between requiring unanimous agreement ("all") or majority agreement ("mode"). This hyperparameter enhances the model's adaptability to different levels of neighbour agreement tolerance.

Theta (θ)

For regression, the agreement concept involves a tolerance level defined by the hyperparameter θ, dictating the acceptable deviation range around an

Algorithm 6 Task-Agnostic Repeated Edited Nearest Neighbours

```
Require: a dataset of examples $\mathbf{X}$.
Require: a boolean-return function $agrees$.
Require: a boolean-return function $selection$.
  for all $\mathbf{x} \in \mathbf{X}$ do
      $agreements \leftarrow$ empty list
      $y \leftarrow$ target value or class label of $\mathbf{x}$
      for all $\mathbf{x}'$ $\in$ the $k$ closest neighbours of $\mathbf{x}$ do
          $y' \leftarrow$ target value or class label of $\mathbf{x}'$
          $agreements \leftarrow agreements + [agrees(y, y')]$
      end for
      if not $selection(agreements)$ then
          Remove $\mathbf{x}$ from $\mathbf{X}$
      end if
  end for
```

example's value for agreement with its neighbours. A larger θ allows for a broader range of agreement, potentially retaining more examples, whereas a smaller θ implies stricter agreement criteria. This hyperparameter fine-tunes the algorithm's sensitivity to neighbour variation, which applies to regression tasks.

Pseudocode

Algorithm 6 depicts task-agnostic RENN's pseudocode.

5.1.3 RENN Implementation

Our RENN implmentation follows the design featured in Algorithm 6 and makes use of a third-party library implementation of the core algorithm (*imblearn* - this is also made clear in the source code *README*); our most significant contribution is its extension to cover regression tasks (not just classification). The *RENNHarness* class, an extension of the *KNNHarness*, integrates RENN into the preprocessing phase for classification and regression tasks. This section elaborates on how RENN is woven into the *RENNHarness* subclass, focusing on the enhancements and modifications from the superclass's *preprocess_dataset* method, which the subclass overrides. For illustrative purposes, UML and process flow diagrams are provided in Figure 3.2

and Figure 3.1, respectively, mirroring the structure for both the subclass and superclass. The subclass retains the fundamental structure of the superclass's *preprocess_dataset* method, incorporating:

- One-hot encoding of categorical columns.
- Conversion of dataset columns to numeric, where applicable.
- Initialisation or application of *StandardScaler* for data normalisation depending on the processing stage.
- Returning a tuple containing the scaled dataset, training targets as a NumPy array, the columns used for training, and the scaler.

The notable distinction in the subclass method lies in the application of RENN. The implementation ensures RENN is applied exclusively to the training or development set, which the preprocessing function identifies by the absence of an existing scaler and a list of training columns in the method invocation. The *RepeatedEditedNearestNeighbours* class from imblearn is employed to facilitate RENN application. Significant to this implementation is the introduction of properties *curr_theta*, *curr_agree_func*, and *curr_kind_sel* alongside their setters; *_get_best_k* is also overridden and the helper function *_get_best_k_helper* is introduced - both these methods enable the fine-tuning of these new hyperparameters during the fine-tuning of k step described in the core design and implementation. Additionally, the class maintains a record of the best-performing combination of *theta*, *agree_func*, and *kind_sel*, ensuring the harness uses the optimum configuration for the dataset at evaluation time. To facilitate all this, we added a new *_agrees* function to the imblearn *RepeatedEditedNearestNeighbours* class that handles the various agrees functions (including classification's agrees function) described in our design. The function's logic for regression includes the computation of tolerance based on the selected *agree_func* (*sd_whole* or *sd_neighbours*). These changes extend Imblearn's RENN implementation to also work for regression tasks.

5.2 BBNR Design and Implementation

In this section, we discuss our design and implementation of BBNR.

Algorithm 7 Blame-Based Noise Reduction

```
Require: X - Dataset
  for all x ∈ X do
      Compute $CSet(\mathbf{x})$, $LSet(\mathbf{x})$, and $DSet(\mathbf{x})$.
  end for
  X ← X sorted in descending order based on $LSet$ size of each example.
  x ← the first example in X.
  while $|LSet(\mathbf{x})| > 0$ do
      X ← X - {x}
      $misClassified \leftarrow false$
      for all x' ∈ $CSet(\mathbf{x})$ do
          if If actual class label of x != predicted class label of x then
              $misClassified \leftarrow true$
              Exit the loop.
          end if
      end for
      if no misclassifications then
          $LSet$ ← Updated $LSet$ for all relevant examples.
          X ← X resorted as before.
          x ← first example in X.
      else
          X ← X + {x}
          x ← next example in X.
      end if
  end while
```

5.2.1 Standard BBNR Design

Algorithm 7 is a reproduction of the pseudocode “Listing 2: BBNRv2” by Pasquier et al. [16]. For explanations of *CSet* (Coverage Set), *LSet* (Liability Set), and *DSet* (Disimilarity Set), please see Chapter 2. This is the ”standard” BBNR design, i.e. only suited to classification tasks.

5.2.2 Task-Agnostic BBNR Design

This section outlines our design for a task-agnostic BBNR approach. This design enables BBNR to handle both classification and regression tasks effectively, with the introduction of tunable hyperparameters enhancing its flexibility and adaptability.

Agrees Function

Central to adapting BBNR for a broader range of tasks is the "agrees" function. Unlike RENN, the agrees function works in two contexts here; it assesses whether an example's actual value aligns with its predicted value and whether a neighbouring example's true value aligns with the given example's actual value. For classification, agreement is straightforward, akin to checking if the predicted class label matches the actual class label. However, for regression, agreement is nuanced, necessitating a comparison of continuous values rather than discrete class labels. We employ two specialised agreement functions for regression tasks:

- **Standard Deviation of Neighbourhood (SDN)**
- **Standard Deviation of the Whole Training Set (SDW)**

These functions assess agreement based on the deviation of an example's predicted value from its actual value, considering either its local neighbourhood's variability (SDN) or the entire dataset's variability (SDW). This dual approach allows the algorithm to adapt its sensitivity to discrepancies based on a local or global context. For regression tasks, the choice between SDN and SDW, alongside the threshold for agreement (Theta), is configurable, providing significant flexibility in tuning the algorithm's behaviour.

Theta (θ)

The θ parameter introduces a threshold for agreement in regression tasks, dictating the tolerance for the deviation between predicted and actual values. This parameter is central to the operation of the "agrees" function, especially when dealing with continuous output values.

Algorithm 8 Task-Agnostic Blame-Based Noise Reduction

```
Require: X - Dataset
Require: a boolean-return function $agrees$.
  for all x $\in$ X do
      Compute $CSet(\mathbf{x})$, $LSet(\mathbf{x})$, and $DSet(\mathbf{x})$.
  end for
  X $\leftarrow$ X sorted in descending order based on $LSet$ size of each example.
  x $\leftarrow$ the first example in X.
  while $|LSet(\mathbf{x})| > 0$ do
      X $\leftarrow$ X - {x}
      $misPredicted \leftarrow false$
      for all x' $\in CSet(\mathbf{x})$ do
          $y \leftarrow$ target value or class label of x
          $y' \leftarrow$ target value or class label of x'
          if not $agrees(y, y')$ then
              $misPredicted \leftarrow true$
              Exit the loop.
          end if
      end for
      if no mispredictions then
          $LSet \leftarrow$ Updated $LSet$ for all relevant examples using $agrees$.
          X $\leftarrow$ X resorted as before.
          x $\leftarrow$ first example in X.
      else
          X $\leftarrow$ X + {x}
          x $\leftarrow$ next example in X.
      end if
  end while
```

Pseudocode

Algorithm 8 depicts task-agnostic BBNR's pseudocode.

5.2.3 BBNR Implementation

Our BBNR implmentation follows the design featured in Algorithm 8. The $BBNRHarness$ class, derived from the $KNNHarness$, introduces the BBNR algorithm into the preprocessing phase for regression and classification tasks.

This section explains how the BBNR algorithm is integrated into the BBNRHarness subclass while emphasising the continuity and modifications from the superclass. For a UML diagram and process flow diagram of the class, refer to Figure 3.2. and Figure 3.1. respectively; they are identical for both the subclass and superclass.

As described in Subsection 5.1.3, the *BBNRHarness* class overrides its superclass' *_preprocess_dataset* method but retains its core functionality. The critical difference is that the subclass method applies BBNR. BBNR is also only applied if the dataset is a training set (or development set; i.e. it cannot be a validation set or testing set), which is determined by the absence of an existing scaler and *training_cols* list in the function call.

Building the Case-Base Competence Model

First, the method builds the competence sets by calling the new *_build_bbnr_sets* method. This method uses the *defaultdict(set)* from Python's *collections* module; this data structure will consist of a example index, and set pairs, and create and manage the coverage, liability, and dissimilarity sets. This choice optimises dynamically updating these sets without initial key-value assignment issues. To populate these sets, *_build_bbnr_sets* uses the new *_predict_on_same_dataset* method, which was a requirement as the standard methods of the superclass would include a given example in its neighbourhood, which is not correct for BBNR.

Noise Reduction

The algorithm iterates over examples with high liabilities. The *_preprocess _dataset* method creates a new NP array of example index, and liability size pairs called *lset_sizes*. It creates another NP array called *indxs_by_desc_lset _size*. It uses NP's *argsort* and list slicing applied to *lset_sizes* to get the example indices sorted in descending order by the size of each example liability set. While iterating over *indxs_by_desc_lset_size*, the method will try to remove each example it iterates over and will apply a reprediction and agrees check to ensure it has not introduced any mispredictions as a result of the removal using methods like *_predict_on_same_dataset*.

On successfully determining an instance as noise, Python's set operations (like add and remove) are used to update the competence sets accordingly.

After noise reduction, the subclass updates the dataset and targets; this involves slicing the NP arrays with a boolean array *removed_examples* to exclude removed examples, an operation that NP can handle very efficiently. Significant to this implementation is the introduction of properties *curr_theta*, and *curr_agree_func* alongside their setters; *_get_best_k* is also overridden and the helper function *_get_best_k_helper* is introduced - both these methods enable the fine-tuning of these new hyperparameters during the fine-tuning of *k* step described in the core design and implementation.

Additionally, the class maintains a record of the best-performing combination of *theta*, and *agree_func*, ensuring the harness uses the optimum configuration for the dataset at evaluation time. To facilitate all this, we added an *_agrees* function to the class that handles the various agrees functions (including classification's agrees function) described in our design. The function's logic for regression includes the computation of tolerance based on the selected *agree_func* (*sd_whole* or *sd_neighbours*).

5.3 Highlighting Novel Contributions

As mentioned in Chapter 2, we thought our design and implementation of BBNR and RENN to work for regression tasks to be original, but Martín et al. [13] had achieved this before our undertaking. However, this section aims to differentiate our work from theirs. The critical difference lies in how they define "noise", as highlighted in Algorithm 1 of their paper. Here, they check if the absolute difference between an example's target value and the mean value of its neighbours is greater than alpha (we call this theta) times the standard deviation of its neighbour's values.

Our approach differs in two ways:

1. We also calculate the standard deviation of the whole training set (not just the neighbourhood). We then validate which agrees function is better at the hyperparameter tuning stage.

2. We apply our "agrees" noise check element-wise - that is, we get the absolute difference between two given target values - not a target value and an aggregate of target values; this contrasts with the aggregation approach for regression taking by Kordos & Blachnik [12] and Martín

et al. [13], where agreement is assessed not at the individual level but across an aggregate of target values, reducing the granularity of agreement testing. We argue our approach adheres more closely to the original implementation of ENN and other noise filters by conducting an element-wise comparison—taking a "vote" from each neighbour within an example's neighbourhood.

Chapter 6

Noise Complaints kNN

This chapter introduces Noise Complaints kNN (NCkNN), a novel variant of the k-Nearest Neighbours algorithm designed to mitigate the impact of noisy and unreliable examples in datasets.

6.1 Design

In this section, we outline the design of NCkNN.

6.1.1 Basic Concept

NCkNN's guiding principle is that not all data points contribute equally to the predictive power of a kNN instance. Specifically, it posits that examples that are more similar to the query point and have higher reliability should significantly influence the outcome—with the balance between reliability and similarity being a tunable hyperparameter. This approach necessitates a departure from traditional distance-based fetching of neighbours and calculation of weights (as in Weighted kNN) by evaluating both examples' similarity and reliability.

6.1.2 Defining Reliability

When defining reliability, we take inspiration from the concept of a Coverage Set in BBNR, which we explain in Chapter 2. For NCkNN, the reliability of an example is the length of its coverage set—so if an example helps to predict ten other examples correctly, it has a reliability of 10. Here, when we say "predict correctly," - we mean it in the way we defined it in our task-agnostic designs of BBNR and RENN, i.e. we abstract an "agrees" function and introduce the hyperparameter theta - and the agrees function used (SDN or SDW) is also tunable. For more details on this, please consult Chapter 5.

6.1.3 Defining Similarity

When defining similarity, we use the inverse of Equation 2.1 (d), as shown in Equation 6.1; that is, if $s(\mathbf{x}, \mathbf{x'})$ is larger than $s(\mathbf{x}, \mathbf{x''})$, then **x'** is more similar to **x** than **x"**. Why the inverse is necessary becomes apparent in the following subsection. Similarily as with Weighted kNN we add a small constant ϵ to the distances before inverting to prevent divisions by zero.

$$s(\mathbf{x}, \mathbf{x}') = \frac{1}{d(\mathbf{x}, \mathbf{x'}) + \epsilon} \tag{6.1}$$

6.1.4 Reliability and Similarity as Influence

We term a given example's combined similarity and reliability as that example's **influence**. NCkNN can incorporate influence in two ways: when fetching neighbours and at prediction time; these modifications to standard kNN can be used mutually exclusively (alongside a standard kNN approach) or in tandem.

Calculating Influence With Lambda (λ)

NCkNN introduces a new tunable hyperparameter λ that dictates how much the algorithm weighs reliability and similarity when calculating influence. The overall influence of one example relative to another is given by Equation 6.2 where **x** is a given example, **x'** is another example, s is Equation 6.1 and r is the reliability of **x'**. If **x** has two neighbours **x'** and **x"** and Equation 6.2 returns a higher result for **x'**, then **x'** is considered more influential than **x"**. However, - as mentioned - NCkNN incorporates influence when fetching nearest neighbours and at prediction time. To aid clarity, λ_s is the λ used when fetching neighbours, λ_p is the λ used at prediction time. Equation 6.2 works as is in both contexts, with λ_s or λ_p.

$$i(\mathbf{x}, \mathbf{x'}) = (\lambda * s(\mathbf{x}, \mathbf{x'})) + ((1 - \lambda) * r) \tag{6.2}$$

Fetching Nearest Neighbours

This subsubsection details what happens when NCkNN is used to fetch nearest neighbours. For a given query example, the algorithm finds k other examples in the development or training dataset with the most influence over it, i.e., the k other training examples that return the highest values from Equation 6.2. If λ_s is 0, reliability is the solely considered metric; if λ_s is 1, NCkNN behaves precisely like normal kNN when fetching nearest neighbours (enabling the use of NCkNN style predictions with standard neighbour fetching). Here, it becomes apparent why we take the inverse of Euclidean distance when defining similarity - simply reliability and similarity would be "at odds" with each other otherwise. The k neighbours of an example $\mathbf{x}$ return maximum results from Equation 6.2 where $\mathbf{x'}$ is another example, s is Equation 6.1 and r is the reliability of $\mathbf{x'}$.

Making Predictions

This subsubsection details what happens when NCkNN is used to predict the class label or target value of an example $\mathbf{x}$ from its k nearest neighbours. We use a modified weighted kNN approach, where instead of calculating weights using Equation 2.2, the weight of each neighbour is that neighbour's influence as calculated with Equation 6.2. For more details on weighted predictions, consult Subsection 2.4.3. It is crucial to highlight a unique case concerning the hyperparameter λ_p. Specifically, when λ_p is set to -1, it is not just a variation within the range of typical values and represents a distinct operational mode. In this scenario, Equation 6.2 is bypassed entirely, reverting to the conventional kNN prediction methodology (utilising either the nearest neighbours' unweighted mean or unweighted mode) for determining the class label or target value. This provision facilitates users who wish to leverage NCkNN for neighbour selection while preferring the straightforward, unweighted prediction approach of standard kNN.

6.1.5 Normalisation Considerations

To effectively tune the hyperparameters λ_s and λ_p, we needed to ensure the range of candidate values for both was between 0 and 1 (with the addition of -1 for λ_p). Otherwise, the search space could be vast, and any search strategy could be blind. From this, we realised it was necessary to normalise both the similarities of a query example to the other examples in the dataset and the reliability of each example in the dataset; otherwise, one may dominate Equation 6.2. We thoroughly evaluated normalisation techniques.

Initial deliberations leaned towards standard scaling, normalising values by subtracting the mean and dividing by the standard deviation. This method appeared suitable for handling the reliability aspect. However, we became concerned regarding the similarity measure, particularly the potential for a query example to exhibit such a degree of dissimilarity from the training examples that it could challenge the robustness of standard scaling's normalisation effect.

Subsequently, we considered robust scaling. This method involves subtracting the median and dividing by the interquartile range (IQR), offering resilience against outliers. Robust scaling is particularly adept at handling extreme values, thus safeguarding the algorithm's performance from the adverse effects of novel or exceedingly dissimilar query examples. Despite its merits in outlier management, robust scaling introduced ambiguity in tuning λ_s and λ_p, rendering the process less intuitive. Robust scaling does not standardise the distributions to have a specific mean and variance. Thus, even after scaling, the inherent characteristics of the similarity and reliability distributions (such as skewness) can affect their contributions to the equation - meaning an optimum λ could very well lay beyond our desired range.

The blending of standard and robust scaling emerged as a potential solution, poised to harness both strengths. However, this approach risked distorting the data excessively, complicating the model's interpretability and potentially obscuring the inherent data patterns essential for reliable predictions.

Another avenue explored was to parameterise the choice of scaling method, incorporating it as a hyperparameter subject to validation. Nevertheless, given our kNN variant's already extensive hyperparameter space, this addition threatened to increase computational demands exponentially, rendering it impractical for the current scope of research. Such an expansion, while intellectually stimulating, was deferred for future investigation to maintain a manageable runtime and complexity.

The decision to revert to standard scaling, despite its potential vulnerability to extreme query examples, was made after careful consideration. The compromise between predictability in hyperparameter tuning and maintaining data integrity without overcomplicating the model guided this choice.

6.1.6 Algorithm Pseudocode

Algorithm 9 depicts the algorithm's design as pseudocode.

Algorithm 9 Noise Complaints k-Nearest Neighbours

```
Require: $\mathbf{X}$ - Dataset.
Require: $\mathbf{Y}$ - Target values / class labels for each example in $\mathbf{X}$.
Require: $\mathbf{x}$ - The example to predict a target value/class for.
Require: $k$ - The number of nearest neighbours to consider.
Require: $\lambda_s$ and $\lambda_p$ - The influence calculation weight hyperparameters.
Ensure: $k > 0$ and $k <$ size of $\mathbf{X}$
  function GetNeighbours($\mathbf{x}, \mathbf{X}, k, \lambda_s$)
      Calculate distances between $\mathbf{x}$ and examples in $\mathbf{X}$.
      Normalise above distances using fitted scaler.
      Calculate influence scores for all $\mathbf{x'} \in \mathbf{X}$ relative to $\mathbf{x}$ using $\lambda_s$.
      return indices of $k$ examples in $\mathbf{X}$ with highest influence scores.
  end function
  function Predict($\mathbf{x}, indices, \mathbf{Y}, k, \lambda_p$)
      if $\lambda_p = -1$ then
          Return mean/mode of $\mathbf{Y}[indices]$.
      else
          Return influence weighted mean/mode of $\mathbf{Y}[indices]$.
      end if
  end function
  $reliabilities \leftarrow$ empty list
  $similarities \leftarrow$ empty matrix
  for all $\mathbf{x} \in \mathbf{X}$ do
      $r(\mathbf{x}) \leftarrow$ Reliability of $\mathbf{x}$ based on coverage sets.
      $s(\mathbf{x}, \mathbf{x'}) \leftarrow$ Similarity between $\mathbf{x}$ and all $\mathbf{x'} \in \mathbf{X}$.
      $reliabilities \leftarrow reliabilities + [r(\mathbf{x})]$
      $similarities \leftarrow reliabilities + [r(\mathbf{x})]$
  end for
  $reliabilities \leftarrow reliabilities$ normalised with standard scaling.
  Fit a standard scaler to $similarities$.
  $indices \leftarrow$ GetNeighbours($\mathbf{x}, \mathbf{X}, k, \lambda_s$)
  $prediction \leftarrow$ Predict($\mathbf{x}, indices, k, \lambda_p$)
  return $prediction$
```

6.2 Implementation

This section details the implementation of NCkNN.

6.2.1 Class Initialisation

The *NoiseComplaintsHarness* class initialises with parameters to distinguish between regression and classification modes, dataset specifics, and handling of missing values. It inherits from *KNNHarness* and extends initialisation to include hyperparameters specific to NCkNN: theta (*_curr_theta*), lambda for similarity (*_curr_lambda_s*) and prediction (*_curr_lambda_p*), and the agreement function (*_curr_agree_func*).

6.2.2 Data Preprocessing

Data preprocessing is a foundational step in the NCkNN approach. It operates functionally as the standard *_preprocessn_dataset*; it calls the superclass method but calls the new *_precompute_and_scale* method to calculate and scale data point similarities and reliabilities. This preparation is crucial for the subsequent neighbour selection process.

Similarity Computation

When *_preprocess_dataset* encounters a training or validation set, it calls *_precompute_and_scale*, which calls *_precompute_similarities* to calculate pairwise similarities among data points based on the inverse of Euclidean distances and save it to the class property *_similarities*. While these similarities are not helpful for neighbour selection, they must fit a scaler.

Reliability Assessment

When *_preprocess_dataset* encounters a training or validation set, it calls *_precompute_and_scale*, which calls *_precompute_reliabilities*, which in turn calls *_build_coverage_sets* to build the coverage set of each example; *_precompute_reliabilities* then uses a for loop to calculate the length of each coverage set and saves these integers to the class property *_reliabilities*.

Scaling

When *_preprocess_dataset* encounters a training or validation set, it is called *_precompute_and_scale*, which in turn is called *_scale_similarity_and_reliability*. This new method initialises two sklearn RobustScaler instances; it fits one to *self._similarities* and saves this fitted scaler to the *_similarity* *_scaler* property, and it fits the other to *self._reliabilities* and transforms the reliabilities.

6.2.3 Fetching Nearest Neighbours

The class overrides *_get_k_nearest_neighbours*, and NCkNN diverges from traditional kNN by incorporating similarity and reliability into its neighbour selection criteria. The method calculates a query example's similarity to each example in the development or training set and scales these similarities with *self._similarity_scaler*. It then dynamically fetches the k nearest neighbours based on each example's composite reliability and similarity score (weighed by *self._lambda*).

6.2.4 Making Predictions

The prediction mechanism in NCkNN works as usual; the aggregate of the selected nearest neighbours' target values or class labels predicts a query example's target value or class label.

6.2.5 Hyperparameter Tuning

The implementation supports tuning theta, lambda, and the agreement function through cross-validation - this optimisation process seeks to identify the best combination of these parameters. The code explores hyperparameter space iteratively, adjusting λ_s, λ_p, and theta values to balance the influence of similarity and reliability. Crucially, the value of the two λ hyperparameters can "turn off" the novel functionality; if λ_s is 1, then NCkNN is just being used at the prediction stage; if λ_p is -1, then NCkNN is just being used for the fetching of nearest neighbours; this means that the optimum combination for a given dataset may use our novel kNN variant at only one of these stages or both.

6.2.6 Overriden Methods

Key to this implementation is the extension of standard kNN functionalities:

- *_get_k_nearest_neighbors* overrides the standard neighbour fetching mechanism focusing on influence scores.

- *_preprocess_dataset* overrides the superclass functionality to now call the new methods *_precompute_and_scale*, *_precompute_similarities*, and *_precompute_reliabilities* which introduce steps for calculating and scaling data point similarities and reliabilities.

- *_knn_classifier* and *_knn_regressor* overide the superclass functionality to leverage weighted approaches based on the computed influence scores.

Chapter 7

Evaluation

This chapter presents the comparative performance evaluation of all the kNN variants discussed in this report, including kNN, Weighted kNN, RENN, BBNR, and NCkNN.

7.1 Reproducibility

As emphasised by Pineau et al. [18], reproducibility is a cornerstone of sound machine learning research, ensuring that results are reliable, promoting open and accessible science, and encouraging robust experimental practices. We have strived to make it convenient to reproduce our experimental data; to that end, we have made all source code available on a public GitHub repository[1]. Once a user clones the repository, the user can easily set up the project by running the provided $Makefile$; this should succeed without manual intervention on most UNIX-based systems. If a user attempting to run this file encounters errors, this is likely due to some unforeseen system idiosyncrasy; in such a case, the setup should be trivial to reproduce by following the $Makefile$ as pseudocode. To run the experiments, we have provided commented code in the script *stat_test_suite.py*. On Windows machines, it may be necessary to call $multiprocessing.freeze\_support()$ to run the cross-validation folds on multiple cores. Furthermore, we took care to seed any pseudorandom operations.

[1]`https://github.com/mattmallencode/NCkNN`

7.2 Experimental Methodology

This section outlines the project's experimental methodology.

7.2.1 Brief Overview

We evaluated each variant (MAE and accuracy being the performance metrics for regression and classification, respectively) using nested 5-fold cross-validation to fine-tune hyperparameters and validate performance, ensuring robustness and generalisability of results. The outer fold partitions the dataset into development (union of training and validation sets) and test segments. The inner fold facilitated hyperparameter optimisation (providing a training/validation split), adjusting k for all algorithms alongside the hyperparameters specific to each variant. We first evaluated each dataset without any artificially introduced noise and then evaluated an artificially noisy version. We ran all experiments using the script *stat_test_suite.py*.

7.2.2 Hyperparameter Search Space

For the θ hyperparameter for RENN, BBNR, and NCkNN for regression tasks, we try the values 1 to 10 inclusive. We decided upon this range after discovering that there was existing literature for adapting classification noise filters to regression tasks. As noted by Kordos & Blachnik [12], ENN proved not very sensitive to changes to θ and in their experiments, the range of the best θ found on many different datasets with many different values for k remained, between 1 and 10.

The range of hyperparameters we try during validation differs for classification and regression tasks. This was because it proved infeasible, given our time and resources, to evaluate the regression datasets with the same robustness as the classification ones. The addition of the θ and $agree\_func$ hyperparameters for regression meant that the amount of computational work required was roughly 20x that required for classification for each candidate k we evaluated. Given the circumstances, it was wise to restrict the number of k's we test and focus our search for λ_p and λ_s. This decision was made to manage the increased computational workload introduced by the θ and $agree\_func$ hyperparameters for regression.

For expanding the k search space for classification tasks, we use the *_expand_k _search_space* method, while we use *_expand_k_search_space_stochastic* for regression; this effectively means fewer ks are tried when evaluating regression datasets, but we believe the trade-off is minimal. For more information, please consult Section 3.12. For classification, the range for both λ weights is 0 to 1 in increments of 0.05 (25 values, with the addition of -1 for λ_p). For classification, the range for both λ weights is 0 to 1 in increments of 0.1 (10 values, with the addition of -1 for λ_p). These disparities ultimately made evaluating the algorithms on regression tasks computationally feasible.

7.2.3 Introducing Artificial Noise

To evaluate the robustness of each algorithm under noisy conditions, we introduce artificial noise into the datasets. Noise is introduced for target values / class labels of the development or training set (depending on the stage of the harness flow). Crucially, the validation or testing targets are left untouched so that the performance can be evaluated against the original denoised targets and class labels. The method for injecting noise varies between classification and regression tasks, as outlined in the *_introduce_artificial _noise* method. For classification, we randomly alter the class label of a specified fraction of the dataset to another class. In contrast, we add Gaussian noise to the target variable of a similarly specified subset of the dataset for regression [2]. The noise level (*self._noise_level*), a fractional value between 0 and 1, determines the proportion of the dataset to be affected.

7.2.4 Determining the Noise Level

The noise levels we consider are 10%, 20%, 30%, 40%, and 50%. We noted that these noise levels are often used in the relevant literature [8]. Ideally, we would evaluate each algorithm at each noise level for each dataset; however, this would mean 5x the amount of computational work, making our experiments computationally infeasible given the size of our hyperparameter search space. We compromise with a novel methodology which ensures a systematic and empirical approach to determining the noisy experimental conditions for each dataset. The appropriate noise level for each dataset is determined based on the point at which the addition of noise results in a statistically significant degradation of model performance. The *find_noise_level* method executes this process. We begin with a baseline (0% noise) and incrementally increase the noise level. We t-test at each level, comparing the model's performance under the current noise level against the baseline. The minimum noise level that results in a statistically significant performance drop while also considering task-specific definitions of degradation (i.e., a decrease in accuracy for classification tasks or an increase in error for regression tasks) is deemed appropriate and selected.

7.2.5 Statistical Significance Testing

To rigorously evaluate and compare the algorithms' performance across various datasets and under different noise conditions, we employ a structured approach to statistical significance testing. After determining an appropriate noise level for each dataset, we proceed with performance evaluation, which leads to the application of statistical tests to assess the results' significance. The analysis is done at the fold level i.e. comparing the performance of one algorithm across 5 folds, to that of another algorithm (not an aggregate to aggregate comparison).

We derived the P-values presented in Tables 7.1, 7.3, 7.7, and 7.8 from the Friedman test which assesses whether there are any statistically significant differences in the algorithms' performance across all cross-validation folds. Understanding that the P-value is not a direct measure of how one specific algorithm outperforms another is essential. Instead, a P-value below 0.05 indicates that at least one algorithm's performance significantly differs from the others for that dataset. This result prompts further investigation through post-hoc tests, such as the Nemenyi test, to pinpoint which algorithms' performances differ significantly.

The Friedman test is a non-parametric alternative to the one-way ANOVA with repeated measures. This test is particularly suitable for our needs due to its applicability to comparing more than two algorithms across multiple datasets, which does not assume a normal distribution of the sample data [5]. Given the heterogeneous nature of our datasets and the potential for non-normal performance distributions across folds, the Friedman test provides a robust means to identify any statistically significant differences in algorithm performance.

If the Friedman test indicates significant differences ($p < 0.05$), we further analyse the results using the Nemenyi post hoc test. This test allows for pairwise comparisons between algorithms, helping identify which specific variants differ significantly. In Tables 7.2, 7.4, 7.5, and 7.6, each cell represents the P-value from the comparison between the row and column algorithms on the given dataset. A lower P-value ($p < 0.05$) indicates a statistically significant difference between the two algorithms. It is important to note that the Nemenyi test does not indicate the direction of the performance difference (i.e., which algorithm is better); it is just that a significant difference exists. We chose the Nemenyi test for its ability to handle multiple comparisons without assuming normal distributions, making it an ideal fit for our evaluation context. It complements the Friedman test by offering detailed insights into the relative performance of the algorithms.

7.3 Classification

In this section, we present the aggregate accuracy of each algorithm across different datasets, focusing on statistically significant findings.

7.3.1 Datasets

We took all our datasets from the UCI Machine Learning Repository. The evaluation covers a variety of datasets to ensure comprehensive assessment across different types and sizes of data:

- **Car Evaluation**[2]: Assessing car acceptability based on features like buying price and safety, with 4 classes, 1728 examples, and 6 features.
- **Heart Disease (Cleveland)**[3]: Binary classification of heart disease presence from clinical and demographic features, with 303 examples and 13 features. We augmented the original data slightly to turn the "num" categorical target column into a boolean feature (originally 0 to 4 with 0 being no presence of heart disease; now 0 or 1 where 1 is any value that was 1 to 4 inclusive) - researchers commonly do this with this dataset.
- **Iris**[4]: Plant species classification based on sepal and petal measure-

[2]`https://archive.ics.uci.edu/ml/datasets/Car+Evaluation`
[3]`https://archive.ics.uci.edu/ml/datasets/Heart+Disease`
[4]`https://archive.ics.uci.edu/ml/datasets/Iris`

ments, with 3 classes, 150 examples, and 4 features.

- **Congressional Voting Records**[5]: Prediction of party affiliation (binary) from voting patterns, with 435 examples and 16 features.
- **Wine**[6]: Origin classification based on chemical analysis, with 3 classes, 178 examples, and 13 features.
- **Zoo**[7]: Artificial dataset. Animal classification from traits, with 8 classes, 101 examples, and 16 features.

7.3.2 Without Noise

Here, we analyse the performance of each algorithm (but with a particular emphasis on investigating NCkNN's relative performance) on the classification datasets where no artificial noise has been introduced. To interpret the P-values, please consult Subsection 7.2.5.

Results and Initial Analysis

Table 7.1 illustrates the performance of each algorithm across the 6 datasets. The best performance in each dataset is highlighted in bold. The kNN and Weighted kNN algorithms display competitive performance across all datasets. However, Weighted kNN's notable improvement in the Iris and Wine datasets suggests that assigning weights to neighbours can offer advantages in situations where the distance metric critically influences classification accuracy. The RENN algorithm tends to underperform in comparison to its counterparts - we could attribute this to its iterative editing process, which could inadvertently eliminate valuable data points in the absence of noise. BBNR demonstrates some potential in datasets like Votes and Zoo. Notably, the NCkNN algorithm showcases superior performance in the Car dataset; the Car dataset evaluation was also the only one to acheive statistical significance with a P-value of 0.0015.

[5]`https://archive.ics.uci.edu/ml/datasets/Congressional+Voting+Records`

[6]`https://archive.ics.uci.edu/ml/datasets/Wine`

[7]`https://archive.ics.uci.edu/ml/datasets/Zoo`

Table 7.1: Aggregate Accuracy of Classification Algorithms without Noise

Dataset	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**	**P-value**
Car	0.9109	0.9057	0.8721	0.8947	**0.9178**	0.0015
Heart	**0.8214**	**0.8214**	0.8114	0.7742	0.8012	0.0551
Iris	0.9400	**0.9667**	0.9533	0.9600	0.9600	0.1024
Votes	0.9139	0.9139	0.9098	**0.9315**	0.9226	0.5878
Wine	0.9776	**0.9832**	0.9606	0.9778	0.9492	0.0708
Zoo	0.9105	0.9600	0.9105	**0.9700**	0.9405	0.1104

Post Hoc Analysis

We only analysed the Car dataset experiment post hoc as this was the only evaluation which achieved statistical significance according to the Friedman test; a Nemenyi test would not provide additional insights for the other datasets since their initial tests did not suggest significant performance variances amongst the algorithms. The pairwise comparison results of the Nemenyi post hoc test are presented in Table 7.2, showing P-values for the significance of differences between algorithm performances. NCkNN's relative performance is of particular interest. The analysis underscores the statistical distinction between NCkNN and RENN, with a P-value of 0.0042, suggesting a significant performance advantage of NCkNN over RENN in this context. However, we can not claim NCkNN is superior to the other algorithms with a high degree of confidence in this context, given the lack of other statistically significant results.

For NCkNN, the optimal parameters λ_s and λ_p were selected based on the performance across five folds. Specifically, λ_s was 1.0 for all folds, indicating that the fetching of neighbours was conducted precisely as in the standard kNN approach. Meanwhile, λ_p was predominantly set to 1.0, emphasising a preference for weighting predictions based on similarity, meaning it behaved like standard weighted kNN except for one fold where $\lambda_p = 0.95$, slightly adjusting the balance towards reliability. Overall, this means the performance of NCkNN is less promising for the Car dataset (at least in the absence of noise) than first thought, as it behaves as normal kNN / weighted kNN the vast majority of the time.

Table 7.2: Nemenyi Post Hoc Test Results for the Car Dataset

Algorithm	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**
kNN	-	0.7811	0.0120	0.2659	0.9000
Weighted kNN	0.7811	-	0.2200	0.8946	0.6108
RENN	0.0120	0.2200	-	0.7243	0.0042
BBNR	0.2659	0.8946	0.7243	-	0.1447
NCkNN	0.9000	0.6108	0.0042	0.1447	-

7.3.3 With Noise

In this analysis, we evaluate the impact of noise on the performance of the algorithms (but with a particular emphasis on investigating NCkNN's relative performance) and discuss the significance of these effects. To interpret the P-values, please consult Subsection 7.2.5.

Results and Initial Analysis

We introduced varying levels of artificial noise into the datasets to simulate real-world data imperfections. We indicate the noise levels as percentages in the results of the Friedman test (in brackets next to the name of each dataset). Table 7.3 presents the aggregate accuracy of each algorithm across the datasets with added noise, providing a comparative overview of algorithm performance in noisy conditions. The best performance in each dataset is highlighted in bold. The kNN algorithm showcases commendable robustness across varied noise intensities, maintaining a solid performance spectrum, notably excelling in the Wine and Car datasets - this demonstrates kNN's general resilience to data quality issues, suggesting foundational stability in handling noise, at least in a classification context. The Weighted kNN variant performs particularly well in the Wine dataset (30% noise), indicating that weighting strategies can effectively counterbalance noise-induced inaccuracies in specific contexts. RENN's approach exhibits a variable efficacy. It bolsters accuracy in higher noise contexts such as the Votes (50%) and Iris (50%) datasets but only sometimes surpass more straightforward methods like kNN. This mixed performance might reflect the dual-edged nature of RENN's data cleaning strategy, which may be beneficial or detrimental based on the specific dataset and noise level. BBNR generally lags in performance. Conversely, NCkNN emerges as a consistently strong contender, being the best-performing algorithm in four of the six datasets. We reach statistical significance in the Wine (P-value: 0.0116), Iris (P-value: 0.0361), and Car (P-value: 0.0032) datasets.

Table 7.3: Aggregate Accuracy of Classification Algorithms with Noise

Dataset	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**	**P-value**
Car (10%)	0.8941	0.8981	0.8646	0.8356	**0.8999**	0.0032
Heart (40%)	0.6673	0.6403	0.6866	0.6266	**0.7473**	0.4702
Iris (50%)	0.7467	0.6667	**0.7733**	0.5533	0.7667	0.0361
Votes (50%)	0.4735	0.5003	0.6463	0.5089	**0.6812**	0.1571
Wine (30%)	0.9270	**0.9381**	0.8937	0.7359	0.9379	0.0116
Zoo (50%)	0.7224	0.6638	0.6929	0.4838	**0.7433**	0.3240

Post Hoc Analysis

We only analysed the Car, Iris, and Wine dataset experiments post hoc as these were the only evaluations which achieved statistical significance according to the Friedman test; a Nemenyi test would not provide additional insights for the other datasets since their initial tests did not suggest significant performance variances amongst the algorithms. The pairwise comparison results, presented as P-values, indicate the statistical significance of performance differences between algorithms. NCkNN's relative performance is of particular interest.

Car Dataset In the Car dataset with 10% noise, the Nemenyi post hoc test results are show in Table 7.4. The P-values obtained from the pairwise comparisons highlight NCkNN's advantageous performance against BBNR with a P-value of 0.0409. However, we can not claim NCkNN is superior to the other algorithms with a high degree of confidence in this context, given the lack of other statistically significant results. The examination of the selected lambda weights for NCkNN across different folds offers crucial insights into its operational mechanism under noise. Specifically, the chosen values for λ_s and λ_p across the five folds were as follows:

- Fold 1: $\lambda_s = 0.9, \lambda_p = 1.0$
- Fold 2: $\lambda_s = 1.0, \lambda_p = 1.0$

- Fold 3: $\lambda_s = 1.0, \lambda_p = 1.0$
- Fold 4: $\lambda_s = 1.0, \lambda_p = 0.85$
- Fold 5: $\lambda_s = 1.0, \lambda_p = 0.95$

This selection pattern primarily emphasises the similarity aspect in the neighbour selection (λ_s), with a consistent preference for similarity-based weighting in predictions (λ_p), evident from the predominantly high λ_p values. Notably, λ_p values are primarily set to 1.0, aligning closely with a standard weighted kNN approach, where the algorithm calculates prediction weights from neighbour similarity. The minor adjustments in λ_p to 0.85 and 0.95 in two of the folds suggest a nuanced consideration for reliability, albeit this adjustment appears marginal. Therefore, the performance of NCkNN in the Car dataset with noise predominantly reflects a strategy that leans heavily towards similarity, with only slight adjustments for reliability. While inconclusive, these are more impressive results than those of the original dataset.

Table 7.4: Nemenyi Post Hoc Test Results for the Car Dataset with Noise

Algorithm	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**
kNN	-	0.9000	0.3172	0.0904	0.9000
Weighted kNN	0.9000	-	0.0904	0.0166	0.9000
RENN	0.3172	0.0904	-	0.9000	0.1795
BBNR	0.0904	0.0166	0.9000	-	0.0409
NCkNN	0.9000	0.9000	0.1795	0.0409	-

Iris Dataset For the Iris dataset with 50% noise, we present the Nemenyi post hoc test results in Table 7.5. The P-values from the pairwise comparisons reveal insights into the performance dynamics among the algorithms, with NCkNN's performance over BBNR standing out, indicated by a P-value of 0.0540. However, NCkNN's performance differential was not statistically significant for any of the other algorithms. The analysis of the lambda weights selected for NCkNN across different folds unveils its operational dynamics in the presence of noise. The chosen values for λ_s and λ_p across the five folds were as follows:

- Fold 1: $\lambda_s = 1.0, \lambda_p = 0.15$
- Fold 2: $\lambda_s = 1.0, \lambda_p = 0.25$

- Fold 3: $\lambda_s = 1.0, \lambda_p = 0.9$
- Fold 4: $\lambda_s = 1.0, \lambda_p = 0.2$
- Fold 5: $\lambda_s = 1.0, \lambda_p = 0.45$

This pattern consistently emphasises the similarity aspect (λ_s) in the neighbour selection process, with significant variability in the weighting of predictions (λ_p), reflecting an adaptive noise-resistance strategy. The range of λ_p values suggests that NCkNN dynamically adjusts its reliance on neighbour's votes based on the noise context, opting for lower weights in noisier conditions to mitigate the impact of unreliable data points. While inconclusive pairwise (beyond the comparison to BBNR) these results are promising, especially given how weighted predictions are far more skewed to reliability than in the other datasets we have post hoc analysed thus far.

Table 7.5: Nemenyi Post Hoc Test Results for the Iris Dataset with Noise

Algorithm	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**
kNN	-	0.5541	0.9000	0.1795	0.9000
Weighted kNN	0.5541	-	0.7243	0.9000	0.2659
RENN	0.9000	0.7243	-	0.3172	0.9000
BBNR	0.1795	0.9000	0.3172	-	0.0540
NCkNN	0.9000	0.2659	0.9000	0.0540	-

Wine Dataset The Nemenyi post hoc test results are presented in Table 7.6 for the Wine dataset with 30% noise. These results offer a comprehensive view of the performance differences between the algorithms under consideration. The Nemenyi post hoc test reveals a significant difference in algorithm performance between BBNR and NCkNN. A P-value of 0.0306 indicates that NCkNN's strategy for dealing with noise is considerably more effective in the Wine dataset. However, NCkNN's relative performance against the other three competing algorithms does not achieve statistical significance.

Examining the lambda weights selected for NCkNN across different folds provides further insights into its adaptive strategy in handling noise. The selected values for λ_s (similarity weight) and λ_p (prediction weight) across the five folds are as follows:

- Fold 1: $\lambda_s = 0.65, \lambda_p = 0.95$

- Fold 2: $\lambda_s = 1.0, \lambda_p = 0.45$
- Fold 3: $\lambda_s = 0.65, \lambda_p = 0.85$
- Fold 4: $\lambda_s = 0.6, \lambda_p = 0.7$
- Fold 5: $\lambda_s = 0.5, \lambda_p = 0.6$

The above values are the most "even" balance between similarity and reliability we saw for a classification dataset, making the results more impressive than those of the Car dataset given that the algorithm behaved far more dissimilarly to standard kNN both when fetching neighbours and making predictions.

Table 7.6: Nemenyi Post Hoc Test Results for the Wine Dataset with Noise

Algorithm	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**
kNN	-	0.9000	0.8378	0.0705	0.9000
Weighted kNN	0.9000	-	0.6676	0.0306	0.9000
RENN	0.8378	0.6676	-	0.4971	0.6676
BBNR	0.0705	0.0306	0.4971	-	0.0306
NCkNN	0.9000	0.9000	0.6676	0.0306	-

7.3.4 Classification Evaluation Summary

For the evaluations of the original datasets, NCkNN was the top performer on only the Cars dataset, which was also the only experiment to achieve statistical significance. However, its lambda parameters were close to traditional unweighted and weighted kNN across the five folds - making this result less impressive. The evaluation of the competing classification algorithms in noisy conditions revealed that NCkNN consistently outperformed other algorithms in four out of six datasets, demonstrating significant statistical superiority over BBNR in the Car Evaluation, Iris, and Wine Origin datasets under noisy conditions. However, when conducting post hoc analysis, its performance was statistically indistinguishable from the other algorithms for these three datasets. Furthermore, its "superiority" in the Car Evaluation dataset is questionable given that the lambda parameters selected make it nearly identical to kNN and Weighted kNN for most of the folds. Despite these promising results, the absence of statistical significance in some comparisons highlights the need for further research to optimise NCkNN and explore its full potential across various datasets and conditions. Overall, NCkNN emerges as a promising, albeit not definitively superior, tool for noise-resilient classification, meriting additional investigation.

7.4 Regression

In this section, we present the aggregate accuracy of each algorithm across different datasets, focusing on statistically significant findings and with a particular emphasis on investigating NCkNN's relative performance.

7.4.1 Datasets

We took all our datasets from the UCI Machine Learning Repository. The evaluation covers a variety of datasets to ensure comprehensive assessment across different types and sizes of data:

- **Auto MPG**[8]: This dataset is utilised for predicting the fuel efficiency (in miles per gallon) of automobiles, based on attributes such as displacement, horsepower, and weight. It comprises 398 instances with 7 features.
- **Automobile**[9]: This dataset is used to predict the risk rating of automobiles. It contains 205 examples and 5 features.
- **Liver Disorders**[10]: This dataset is used to predict half-pint equivalent alcoholic beverages consumed per day based on blood tests that are sensitive to liver disorders. It includes 345 instances and 5 features, with the selector attribute removed for our regression analysis.
- **Real Estate Valuation**[11]: This dataset serves to predict property prices in New Tapei City, based on factors such as house age, distance to the nearest MRT station, and location coordinates. It comprises 414 examples and 6 features.

[8]`https://archive.ics.uci.edu/ml/datasets/Auto+MPG`
[9]`https://archive.ics.uci.edu/ml/datasets/Automobile`
[10]`https://archive.ics.uci.edu/ml/datasets/Liver+Disorders`
[11]`https://archive.ics.uci.edu/ml/datasets/Real+estate+valuation+data+set`

- **Student Performance (Math)**[12]: An analysis aiming at predicting student grades in mathematics at the secondary education level. It includes 396 instances and 30 features, with the first and second year grades (G1 and G2) removed to prevent data leakage.
- **Student Performance (Portuguese)**[13]: Similar to the mathematics dataset but focuses on Portuguese language grades. It contains 649 examples and 30 features, also with the G1 and G2 features excluded for the reasons mentioned above.

7.4.2 Without Noise

Here, we analyse the performance of each algorithm on the regression datasets without the presence of artificial noise. To interpret the P-values, please consult Subsection 7.2.5.

Results and Initial Analysis

Table 7.7 illustrates the performance of each algorithm across the regression datasets in terms of MAE. The best performance in each dataset is highlighted in bold. The kNN and Weighted kNN algorithms consistently demonstrate robust performance across the datasets, with at least one of the two algorithms emerging as a top performer for each dataset. This consistent performance underscores their reliability and efficacy in regression tasks without introducing noise. The Automobile dataset serves as a prime example, where kNN and Weighted kNN algorithms significantly outperform RENN, BBNR, and NCkNN, boasting the lowest MAE scores by a significant margin. The performance of the RENN and BBNR algorithms in the Automobile dataset suggests a potential drawback in their ability to refine the training set in the absence of significant noise for regression tasks, possibly due to over-elimination of valuable data points. Notably, for our study, NCkNN is relatively competitve but crucially does not outperform the non-noise-filtering kNN approaches on any dataset.

[12] `https://archive.ics.uci.edu/ml/datasets/Student+Performance`

[13] `https://archive.ics.uci.edu/ml/datasets/Student+Performance`

Table 7.7: Aggregate MAE of Regression Algorithms without Noise

Dataset	kNN	W.kNN	RENN	BBNR	NCkNN	P-value
Automobile	**0.4657**	**0.4657**	0.7356	0.7299	0.6601	0.0011
Auto MPG	**0.1225**	**0.1225**	0.1761	0.1817	0.1700	0.0014
Liver	2.4089	**2.3297**	2.3991	2.3588	2.3504	0.0527
Real Estate	5.7758	**5.3359**	5.6789	5.7766	5.7107	0.0250
Student (M)	**3.2467**	3.2612	3.2531	3.3563	3.2888	0.0652
Student (P)	2.0946	**2.0816**	2.0841	2.0855	2.0979	0.1882

Post Hoc Analysis

While we achieved statistical significance on three of the six regression experiments without noise, none of the Nemenyi results were statistically significant for NCkNN. Given that the algorithm that is the focus of this study also did not exhibit superior performance on any of the original datasets, we have elected not to conduct a significant post hoc analysis here to avoid verbosity.

7.4.3 With Noise

In this section, we examine the impact of artificial noise on the performance of regression algorithms, focusing on NCkNN's adaptability and efficacy relative to other algorithms under noisy conditions. To interpret the P-values, please consult Subsection 7.2.5.

Results and Initial Analysis

To simulate real-world imperfections in data, we introduced varying levels of noise into the regression datasets. The noise levels are denoted as percentages, reflecting the proportion of the affected dataset elements. The performance of each algorithm in these noisy conditions is in Table 7.8, where performance is measured in terms of MAE. Notably, NCkNN demonstrates robustness to noise, outperforming or closely competing with the other algorithms across several datasets. This observation is particularly significant in the Automobile, Auto MPG, Liver, and Student Math datasets, where NCkNN achieved the lowest MAE under noisy conditions.

Table 7.8: Aggregate MAE of Regression Algorithms with Noise

Dataset	**kNN**	**W.kNN**	**RENN**	**BBNR**	**NCkNN**	**P-value**
Automobile (40%)	0.7809	0.7375	0.7577	0.7273	**0.7163**	0.5781
Auto MPG (10%)	0.2297	0.2297	0.2560	0.2575	**0.2008**	0.1249
Liver (50%)	2.3524	2.3547	2.4375	2.4298	**2.3392**	0.1933
Real Estate (50%)	6.1441	**5.9541**	6.2349	6.0926	6.0859	0.2598
Student (M,40%)	3.3037	3.3322	3.3578	3.3698	**3.2702**	0.1712
Student (P,10%)	2.1110	2.0986	2.1254	**2.0858**	2.1089	0.1424

Post Hoc Analysis

Given the absence of statistically significant differences in performance among the algorithms (as indicated by the P-values in Table 7.8), a detailed post hoc analysis was not warranted for any of the datasets.

7.4.4 Regression Evaluation Summary

In noise-free environments, the kNN and Weighted kNN algorithms demonstrated consistent and robust performance across all datasets, underscoring their general applicability and reliability in regression tasks. The NCkNN algorithm, while competitive, was not the top performer for any of the noise-free datasets. This finding suggests that, in the absence of artificial noise, the benefits of NCkNN's nuanced neighbour selection and weighting mechanism may prove unnecessary. With the introduction of noise, NCkNN's design to mitigate the impact of unreliable data points through its adaptive neighbour weighting becomes more relevant. The algorithm showed notable noise robustness, particularly in the Automobile, Auto MPG, Liver, and Student Math datasets, where NCkNN exhibited the lowest MAE values under noisy conditions. However, we did not achieve statistical significance for any of the noisy regression experiments. NCkNN appears to be a promising yet unproven tool for noise-resistant regression; the lack of definitive superiority in terms of statistical significance calls for further investigation.

Chapter 8

Conclusions

This chapter summarises the project and its accomplishments. Additionally, we highlight potential future lines of enquiry suggested by our research.

8.1 Summary of Findings

Our comparative performance evaluation of kNN variants—including kNN, Weighted kNN, RENN, BBNR, and NCkNN—revealed several key insights. In noise-free environments, kNN and Weighted kNN consistently demonstrated robust performance across classification and regression tasks, affirming their reliability and efficacy. Designed to mitigate noise, the noise-filtering algorithms RENN and BBNR did not consistently outperform the simpler kNN variants without artificial noise, supporting our assumed notion that their utility is context-dependent.

The focus of our research, NCkNN, showed promise in handling noisy conditions for both classification and regression tasks, being the top performer in four of six datasets for both task categories when we introduced artificial noise. However, the observed statistical significance of this superiority was minimal. The post hoc analysis of the three artificially noisy dataset experiments that achieved initial statistical significance (Car, Iris, Wine: all classification datasets) revealed that NCkNN was only statistically significantly better than BBNR. Overall, the findings underscore NCkNN's potential in noisy data landscapes (especially given it was the top performer on 8 of 12 artificially noisy experiments). However, given the lack of convincingly statistically significant results, further investigation is warranted to understand its advantages and fully optimise its performance.

Our findings underscore the criticality of selecting the appropriate kNN variant based on the specific context of use, particularly the presence of noise in the data. While NCkNN emerges as a promising tool for dealing with noisy datasets, its effectiveness relative to the editing algorithms and simpler variants like kNN and Weighted kNN varies. This necessitates a nuanced approach to algorithm selection, emphasising that researchers and data scientists must consider the specific context when choosing a kNN variant.

8.2 Limitations of Research

This section summarises what we believe to be the limitations inherent to our approach and potential methods of overcoming them.

8.2.1 Statistical Significance

Pursuing statistical significance in our experimental results faced several challenges, limiting the robustness of our findings. Key among these was the scale and scope of the hyperparameter search space and the datasets' inherent variability. To enhance the likelihood of obtaining statistically significant outcomes, we propose three strategies:

- **Increased Sample Size:** Expanding the number of datasets or replicating experiments across more folds could enhance the statistical power of our analyses, making it easier to detect actual differences between algorithms.
- **Refined Hyperparameter optimisation:** Implementing more sophisticated hyperparameter search techniques, such as Bayesian optimisation, could lead to better-tuned models, thereby increasing the performance differential. As highlighted by Turner et al. [22], employing Bayesian optimisation for hyperparameter tuning can significantly enhance model performance over traditional search methods.
- **Alternative Statistical Tests:** Employing more sensitive statistical tests or adjusting significance levels may uncover subtle but meaningful differences between models.

While potentially increasing the computational complexity or resource demands of the research, these approaches could significantly improve the clarity and robustness of our conclusions regarding the comparative performance of the kNN variants.

8.2.2 Diversity of Error Estimates

A limitation of our research is the reliance on a single error metric (MAE for regression and accuracy for classification) to evaluate algorithm performance. This approach may only partially capture the nuanced behaviours and strengths of the various kNN variants under different conditions. For instance, metrics such as the root mean squared error (RMSE) could provide additional insight into the algorithms' performance by penalising significant errors more heavily than the MAE. Similarly, precision, recall, and the F1 score could offer a more detailed understanding of performance in classification tasks, especially in imbalanced datasets where accuracy may not fully reflect an algorithm's effectiveness.

Future research could incorporate a broader array of performance metrics, including RMSE, precision, recall, F1 score, and area under the receiver operating characteristic curve (AUC-ROC); this would enable a more comprehensive evaluation of the algorithms, potentially revealing strengths or weaknesses not apparent through MAE or accuracy alone. Furthermore, a multi-metric approach could help identify scenarios where a particular algorithm excels, providing more explicit guidance for practitioners on the most suitable kNN variant for their specific needs.

8.2.3 Diversity of Noise

Our approach to introducing artificial noise changed the target values and class labels, which may not fully capture the algorithm's performance under other noisy conditions found in real-world data, including feature noise, outliers, systematic noise, and missing values that may influence the kNN variants differently. Moreover, our relatively clean and well-curated UCI Machine Learning Repository datasets could provide relatively weak indications of performance in the face of the challenges of inherently noisy real-world datasets. Future research could benefit from trying these algorithms in a broader range of noisy contexts: inherently real-world datasets and more artificial noise types. Such research would provide much more insight into the practical applicability of and limit to noise handling.

8.2.4 Diversity of Datasets

While the datasets used in our experiments certainly present a broad range of problem domains and types of features, our choice of datasets is likely to suffer from size, complexity, and diversity limitations; we could have included larger datasets or datasets of time-series data, thus providing a more challenging and revealing testbed for evaluating the kNN variants. Time-series datasets can reveal much more about the ability of algorithms to handle sequential data and the impact of noise therein. It would also be interesting to look into datasets with much broader feature diversity, like text or images, to see whether this sheds light on the adaptability of the algorithms to different representations of data and the effectiveness of their distance metrics. The computational demands and resource requirements for processing more extensive and more complex datasets were a constraining factor for our project. Future work could look at expanding this to cover a more varied set of data types, specifically datasets that embrace real-world application challenges in natural language processing, computer vision, and IoT sensor data analysis. This expanded diversity would enhance the generalisability of any findings and make them more pertinent to popular application scenarios.

8.3 Future Lines of Enquiry

This section summarises potential future avenues of research that may merit exploration.

8.3.1 Incremental Reliability Calculations

In dynamic environments where underlying data patterns can change in unforeseen ways — a phenomenon called concept drift — NCkNN's effectiveness (and that of the other variants) can diminish. In such scenarios, it is common for developers and data scientists to introduce new data into the development or training set to make the model/instance more relevant to the current state of the world. Introducing new training data would invalidate some of the reliability scores previously calculated with NCkNN. Calculating reliability scores incrementally in NCkNN could help address concept drift in a more computationally efficient manner. Rather than re-evaluating reliability scores for the entire dataset with each new data entry, the algorithm could pinpoint and update only the scores of directly affected examples, significantly enhancing efficiency. This approach minimises computational overhead and adapts more fluidly to new information. Future research could focus on developing methods to accurately identify these affected examples and adjust their scores dynamically.

8.3.2 Evaluating Computational Efficiency

While we believe NCkNN adheres closer to the lazy learning model than the editing algorithms, it still introduces a training step not present in standard kNN, the calculation of reliability scores, and the normalisation of similarity and reliability. Compared to various editing algorithms, its efficiency, particularly in terms of time and space complexity, remains an area ripe for further exploration. While doing more invasive work pre-prediction than NCkNN by cleaning the dataset to reduce noise, the editing algorithms could decrease the amount of work at prediction time by reducing the training and validation set sizes. However, without proper systematic analysis, this is just speculation. Even if more statistically significant work proved NCkNN to be superior for many contexts in terms of error estimates, there are other concerns regarding algorithm choice. If NCkNN, for example, was significantly less efficient than the editing algorithms in time and space, this could outweigh superior error estimates. To conclusively determine the superior approach under varying conditions, a systematic analysis encompassing both error estimation and computational efficiency—both in terms of time and space—is necessary. This investigation would provide valuable insights into the trade-offs between our approach and the cleansing of datasets using the editing algorithms.

8.3.3 Depth of Reliability

An intriguing area for future exploration involves enhancing the reliability metric of NCkNN with a new variant incorporating concepts akin to the PageRank algorithm [1]. This adaptation would assess neighbours' immediate reliability and consider their respective neighbours' reliability, thereby introducing a cascading effect of trustworthiness. The approach would fine-tune the selection of neighbours based on the more extensive network of relations and their reliability. It offers a more nuanced and comprehensive mechanism to mitigate the influence of noisy or outlier data points. Such a "depth of reliability" concept may open new dimensions for performance improvement.

8.3.4 Open Source Contribution

Contributing our NCkNN and BBNR implementations to open-source machine learning libraries like sklearn would certainly extend their impact. Open-source contributions also facilitate further development and optimisation of these algorithms through more accessible peer review, contributing to their overall technical quality [14]. Future efforts may focus on cleaning up and refining those implementations to fit the libraries' contribution standards.

8.3.5 More Comparisons

Discretization converts continuous features into discrete ones while binning groups of data points based on their range values; this is an alternative way of adapting noise filters like RENN and BBNR to regression tasks [25]. Expanding the comparative analysis to include discretization and binning approaches could enrich our understanding of how these methods stack against NCkNN and our editing algorithm implementations in handling continuous values. Furthermore, a direct comparison between our adaptations of BBNR and RENN for regression tasks and existing adaptations could illuminate the benefits or drawbacks of our unique approach. Our methodology, mainly distinct in its implementation of an element-wise "agrees" check, diverges from the aggregation strategies employed by others in the field. This juxtaposition could shed light on whether our closer adherence to the foundational principles of noise filtering is actually of greater efficacy.

Bibliography

[1] Brin, S., & Page, L. (1998). The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, *30*(1), 107–117. Proceedings of the Seventh International World Wide Web Conference.

[2] Brownlee, J. (2019). Train neural networks with noise to reduce overfitting. Accessed: [28/03/2024].
URL `https://shorturl.at/NQT45`

[3] Chrzeszczyk, A., & Kochanowski, J. (2011). Matrix computations. In *Encyclopedia of Parallel Computing*.

[4] Chumachenko, D., Meniailov, I., Bazilevych, K., Chumachenko, T., & Yakovlev, S. (2022). Investigation of statistical machine learning models for covid-19 epidemic process simulation: Random forest, k-nearest neighbors, gradient boosting. *Computation*, *10*(6).

[5] Demšar, J. (2006). Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, *7*(1), 1–30.

[6] Dudani, S. A. (1976). The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-6*(4), 325–327.

[7] Fix, E., & Hodges, J. L. (1989). Discriminatory analysis - nonparametric discrimination: Consistency properties. *International Statistical Review*, *57*, 238.

[8] Hasan, R., & Chu, C. H. (2022). Noise in datasets: What are the impacts on classification performance? In *International Conference on Pattern Recognition Applications and Methods*.

[9] Hassanat, A., Abbadi, M. A., Altarawneh, G. A., & Alhasanat, A. A. (2014). Solving the problem of the k parameter in the knn classifier using an ensemble learning approach. *ArXiv*, *abs/1409.0919*.

[10] Herlihy, M., & Shavit, N. (2012). *The Art of Multiprocessor Programming, Revised Reprint*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1st ed.

[11] Jordan, M. I., & Mitchell, T. M. (2015). Machine learning: Trends, perspectives, and prospects. *Science*, *349*(6245), 255–260.

[12] Kordos, M., & Blachnik, M. (2012). Instance selection with neural networks for regression problems. In A. E. P. Villa, W. Duch, P. Érdi, F. Masulli, & G. Palm (Eds.) *Artificial Neural Networks and Machine Learning – ICANN 2012*, (pp. 263–270). Berlin, Heidelberg: Springer Berlin Heidelberg.

[13] Martín, J., Sáez, J. A., & Corchado, E. (2021). On the regressand noise problem: Model robustness and synergy with regression-adapted noise filters. *IEEE Access*, *9*, 145800–145816.

[14] Morgan, L., & Finnegan, P. (2007). Benefits and drawbacks of open source software: An exploratory study of secondary software firms. In J. Feller, B. Fitzgerald, W. Scacchi, & A. Sillitti (Eds.) *Open Source Development, Adoption and Innovation*, (pp. 307–312). Boston, MA: Springer US.

[15] Parsodkar, A. P., P., D., & Chakraborti, S. (2022). Never judge a case by its (unreliable) neighbors: Estimating case reliability for cbr. In M. T. Keane, & N. Wiratunga (Eds.) *Case-Based Reasoning Research and Development*, (pp. 256–270). Cham: Springer International Publishing.

[16] Pasquier, F.-X., Delany, S.-J., & Cunningham, P. (2005). Blame-based noise reduction: An alternative perspective on noise reduction for lazy learning. *Computer Science Technical Report TCD-CS-2005-29*.

[17] Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., Duchesnay, E., & Louppe, G. (2012). Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, *12*.

[18] Pineau, J., Vincent-Lamarre, P., Sinha, K., Larivière, V., Beygelzimer, A., d'Alché Buc, F., Fox, E. B., & Larochelle, H. (2020). Improving reproducibility in machine learning research (a report from the neurips 2019 reproducibility program). *J. Mach. Learn. Res.*, *22*, 164:1–164:20.

[19] Raschka, S., Patterson, J., & Nolet, C. (2020). Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence. *Information*, *11*(4).

[20] Todd Anderson, & Tim Mattson (2021). Multithreaded parallel Python through OpenMP support in Numba. In Meghann Agarwal, Chris Calloway, Dillon Niederhut, & David Shupe (Eds.) *Proceedings of the 20th Python in Science Conference*, (pp. 140 – 147).

[21] Tomek, I. (1976). An experiment with the edited nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics*, *SMC-6*(6), 448–452.

[22] Turner, R., Eriksson, D., McCourt, M. J., Kiili, J., Laaksonen, E., Xu, Z., & Guyon, I. M. (2021). Bayesian optimization is superior to random search for machine learning hyperparameter tuning: Analysis of the black-box optimization challenge 2020. In *Neural Information Processing Systems*.

[23] Willmott, C. J., & Matsuura, K. (2005). Advantages of the mean absolute error (mae) over the root mean square error (rmse) in assessing average model performance. *Climate Research*, *30*, 79–82.

[24] Yadav, S., & Shukla, S. (2016). Analysis of k-fold cross-validation over hold-out validation on colossal datasets for quality classification. *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, (pp. 78–83).

[25] Álvar Arnaiz-González, Díez-Pastor, J. F., Rodríguez, J. J., & García-Osorio, C. I. (2016). Instance selection for regression by discretization. *Expert Systems with Applications*, *54*, 340–350.