Comparison of Machine Learning Algorithms

Interim Report

Nathan Kurien

CS3822: BSc Final Project

Supervised by: Prof. Zhiyuan Luo
Department of Computer Science
Royal Holloway University of London

Abstract

This project undertakes a comparative analysis of two fundamental machine learning algorithms for classification tasks: K-Nearest Neighbors and Decision Trees, with a focus on the CART algorithm. Employing Python and Jupyter Notebooks, the project aims to evaluate their accuracy and performance characteristics. Initial efforts have centered on developing proof-of-concept programs to grasp the fundamental workings of these algorithms. K-Folds Cross-Validation has been utilized to assess their performance on small datasets with continuous data, revealing interesting variations in accuracy influenced by the choice of k in K-Nearest Neighbors and the maximum depths in Decision Trees.

Moving forward, the project will delve deeper into these insights, optimizing the models and introducing more complex datasets to comprehensively evaluate their performance across diverse scenarios. This endeavour not only seeks to enhance understanding in the fields of data mining and machine learning but also aims to contribute valuable insights and methodologies relevant in today's rapidly advancing technological environment.

Contents

Navigation Guide and Declaration

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 Where other published and unpublished source materials have been used, these have been acknowledged.

I've included a navigation guide here to ease traversal through this report for assessment.

Word Count(excluding diary and references): 10171

- Aims and Objectives - Sections 1.4 and 6
- Planning and Timescales - Section 4.2
- Literature Survey and Research Material
  - Within the project scope - Sections 2 to 3.4
  - Beyond the project scope - Sections 1.1 to 1.2
- Summary of Completed Work - Section 4
- Diary - Appendix (roughly 2400 words)

1 Introduction

1.1 Machine Learning and Classification Tasks

Machine learning (ML) has taken the world by storm in recent times. In the era of unprecedented data generation, ML has emerged as a pivotal technology with profound implications for industries and everyday life. It underpins significant advancements in various sectors, including healthcare, finance, transportation, and more, by enabling intelligent systems that can learn from data and make informed decisions.

In our daily lives, the influence of machine learning is both subtle and profound. It powers the personalized recommendations we receive on streaming services like Netflix and Spotify, tailoring entertainment to our tastes [1, 2]. When we shop online, ML algorithms analyze our purchasing habits to suggest products we might like, enhancing our shopping experience [3]. Even our interactions with smartphones, from voice assistants like Siri and Google Assistant to predictive text and search, are driven by machine learning technologies, making them more intuitive and responsive to our needs [4, 5].

Beyond these conveniences, machine learning drives significant developments in critical areas. In healthcare, it aids in the early diagnosis of diseases [6], personalized treatment plans [7], and even in drug discovery, [8] revolutionizing patient care and outcomes. In the realm of finance, machine learning algorithms detect fraudulent activities and automate trading, providing more secure and efficient financial services [9]. In transportation, from optimizing logistics and delivery routes to the development of autonomous vehicles, ML is at the forefront, promising safer and more efficient transport systems [10].

At the heart of all of this are classification tasks, which are central to many ML applications. Supervised learning, a cornerstone of machine learning, involves training algorithms on labeled data, where the correct answers are provided, enabling the model to learn and make predictions on new, unseen data. Classification involves assigning categories or labels to data points based on their features, a task that is fundamental in pattern recognition, decision-making, and predictive modelling.

The significance of machine learning is not only growing in the realms of convenience and efficiency but also gaining traction as a key driver of global innovation. It is at the centre of research and development across industries, pushing the boundaries of what technology can achieve. Countries and companies are investing heavily in AI and ML research, recognizing their potential to spur economic growth, enhance competitiveness, and solve some of the world's most pressing challenges, from climate change [11] to healthcare.

Machine learning, with its ability to analyze vast amounts of data and learn from it, is not just a technological advancement but a transformative force reshaping how we live, work, and interact with the world around us. As it continues to evolve, its impact is set to become even more profound, making its understanding and application an essential facet of modern life.

1.2 Breakthroughs and Developments in Machine Learning

The landscape of machine learning has been significantly reshaped with the advent of deep learning, an advanced subset of neural networks. Deep learning models, particularly Convolutional Neural Networks (CNNs), have revolutionized image processing and analysis [12]. They have enabled machines to achieve near or even surpass human-level accuracy in tasks like image recognition and classification. Recurrent Neural Networks (RNNs), another pivotal development in this domain, have been instrumental in processing sequential data, making significant strides in speech recognition and natural language processing [13] .

Natural Language Processing (NLP) has witnessed transformative changes with the introduction of models like BERT and GPT [14, 15]. These transformer-based models have set new benchmarks in understanding and generating human language. Their applications range from sophisticated chatbots to advanced systems for language translation, offering a level of context-awareness previously unattainable in machine learning.

Reinforcement learning, a type of machine learning concerned with how agents ought to take actions in an environment to maximize cumulative reward, has seen groundbreaking successes. The development of algorithms like those used in DeepMind's AlphaGo [16] and AlphaZero [17] represents a significant leap in this area. These systems not only learned to play complex games like Go and Chess but also developed strategies that were innovative and unconventional, showcasing the potential of AI to discover solutions that humans might not conceive.

Computer vision has been another area where deep learning has made a substantial impact. Real-time object detection systems like YOLO [18] have changed the landscape of visual recognition tasks. These systems can identify and classify multiple objects in images and videos with remarkable speed and accuracy. Facial recognition technology, powered by deep learning, has also advanced, finding applications in security, authentication, and personal identification.

The breakthroughs mentioned above have generated a large amount of attention in news and media, and outlines the prospects of this field looking ahead. My project, of course, will be focused on the pure and basic fundamentals - but this section provides some perspective on its significance. It also emphasises my enthusiasm and interest in this field. It is my goal to join the forefront of this emerging technology.

1.3 Project Specification

The primary objective of this project is to conduct a thorough comparative analysis between two simple, fundamental machine learning algorithms on the classification task. The algorithms selected are K-Nearest Neighbours (KNN) [19, 20] and Decision Trees - more specifically, the CART algorithm [21]. This involves implementing these algorithms from scratch, which provides a deeper understanding of their mechanics and intricacies. The comparison will focus on various performance metrics, including accuracy, precision, and recall, to evaluate the effectiveness of each algorithm in different scenarios. These algorithms and metrics are elaborated in more detail in this report.

The models are implemented in Python [22], creating the data structures using Lists and NumPy [23], and using various existing libraries as points of reference. Sci-kit Learn [24] is a highly regarded Python library by data scientists and includes various implementations of the models that are relevant to this project. This library proved an invaluable point of reference during the development of both algorithms, and served as a comparable benchmark when initially determining the implementation's performance.

Jupyter notebooks have been used initially while the models are in development. They have also been tremendously useful for building visualisations and validating the models. Later on in development, I hope to implement these models in a standalone interface that can take datasets as inputs and provide useful insights on the performance of chose models on this data. Creating the interface will potentially be implemented using Python libraries such as PyQt5. The goal is to implement this standalone interface in Term 2, using the proof-of-concept programs put in place prior. There are vast amounts of resources I've found helpful in deciphering the theory behind this project.

Additionally, the project aims to explore the impact of hyperparameters on these algorithms. For KNN, this includes examining how different values of 'k', the number of nearest neighbours, affect the classification outcome. Similarly, for CART, the focus will be on understanding how different tree depths influence the model's ability to generalize from training data.

1.4 Milestones

Within my Project Plan, my approach was broken down by approaching the algorithms and model evaluation methods all as separate components. I had a success criteria in place for the end of Term 1 - and I believe I've managed to fulfil this. The milestones put in place from my Project Plan are listed as follows.

1.4.1 First Term

Reports

Report 1: Nearest Neighbour Algorithms - In this report I hope to discuss the theory behind nearest neighbour algorithms, 1-Nearest Neighbour and K-Nearest Neighbour, and further details behind how they function.
Report 2: Decision Tree Algorithms - In this report, I hope to discuss the theory behind the Decision Tree algorithm and explore its measures of uniformity.

I sense that completing both of these reports will fulfill my understanding of the inner workings of these algorithms, and will supplement my interim report at the end of the year.

Programs

I will begin by implementing Proof of Concept programs in Python, starting very simple, and then expanding as I begin to expand both on the algorithms, and the complexity of my datasets. My PoC Programs will be listed as follows, in three sections:

Nearest Neighbours

- 1-NN algorithm implemented on very small synthetic dataset - in essence, coordinates on a 2-dimensional axis. The algorithm will essentially calculate Euclidean Distance between two points and set a solid starting point for development.
- 1-NN algorithm implemented on the infamous iris [25] dataset from the UCI repository [26], starting on two classes and a single feature, and then eventually implemented more features and classes from the dataset.
- Implement K-neighbours:
  - Starting with k=3 - and find differences with results from the iris dataset
  - Increase value for k, towards k=10.
  - Find cases when tie-breaking is necessary i.e. when k is even
- Introduce tie-breaking policies in the algorithm
- At this point, introduce performance metrics from PoC programs made in *model validation.*

- Also, the algorithm at this point should be able to handle larger and more complex datasets.

Decision Trees

- Start on simple synthetic dataset containing features and labels and build a very basic decision tree classifier to classify data points based on their features
  - Initially, a Decision Tree Node class will be created which should store information about the split criterion - which will likely expand as the algorithm is extended with more hyperparameters
  - A tree classifier class that follows the CART algorithm will recursively partition the dataset into various subsets until the optimum Gini impurity is found. This PoC program should explore how to do that.
- Visualisation - Before an interface is implemented, and while a library isn't being used, it would be wise to investigate how to visualise the tree using the console output.
- Once the program has progressed this far, overfitting will rapidly become an issue. Adding splitting criterion hyperparameters will be useful at this point. Adding a maximum depth will be my first step into implementing this.
  - There are a wide number of extensions I can go into, when it comes to reducing overfitting on trees, such as pruning techniques and implementing forests

Model Validation

- Start a program that can perform a train-test-split on a dataset, similar to that of Scikit-Learn
- Produce small programs that can calculate metrics such as accuracy, precision, recall and f-score after the training process.
- Implement K-folds Cross-Validation - program that can split dataset into various subsets and cycle through each data set for training + testing
  - The assessment phase during each iteration can be extended with various different metric calculations, including robustness for KNN and feature importance for DTs

1.4.2 Success Criteria

By the end of the first term, it is essential that I have implemented both algorithms, K-Nearest Neighbours and Decision Trees, in some manner using Jupyter Notebooks, as well as using any form of model validation techniques to determine performance metrics of both algorithms. This is what I'd call an essential deliverable before the interim review, and I think it's feasible with the time and preparation given, and necessary compromises made.

2 Algorithms

In this chapter, we introduce and evaluate the algorithms I'll be reviewing in this project.

2.1 K-Nearest Neighbours

K-Nearest Neighbours (KNN) is a versatile, easy-to-implement, and foundational algorithm in the realm of machine learning, widely recognized for its simplicity and effectiveness in classification and regression tasks.

2.1.1 Overview

The K-Nearest Neighbours algorithm is a type of instance-based learning, or lazy learning, where the function is only approximated locally and all computation is deferred until function evaluation. It is a non-parametric method used for classification and regression. In both cases, the input consists of the k closest training examples in the feature space. The output depends on whether KNN is used for classification or regression:

In KNN classification, the output is a class membership. An object is classified by a majority vote of its neighbours, with the object being assigned to the class most common among its k nearest neighbours (k is a positive integer, typically small). If k = 1, then the object is simply assigned to the class of that single nearest neighbour.

In KNN regression, the output is the property value for the object. This value is the average of the values of its k nearest neighbours.

KNN is a type of lazy learning where the function is only approximated locally and all computation is deferred until classification. The algorithm doesn't learn a discriminative function from the training data but memorizes the training dataset instead.

KNN can be used for both classification and regression problems. However, it is more widely used in classification problems in the industry. The simplicity of the KNN algorithm is one of its biggest assets, making it a great starting point for new data scientists to familiarize themselves with machine learning and data analysis.

I chose this algorithm as my first algorithm to implement due to its simplicity, ease of implementation, and how intuitive its mechanism is to comprehend. Choosing this algorithm as my first model allowed me to have meaningful results I could deliver quite early on in my project's implementation. I would position KNN almost as an introductory tool to ML, and its simplicity can be dissected further as we introduce more sophisticated models later into the project - this was at least, my intention.

There are endless amounts of resources available to look into this algorithm, I've found two authoritative sources that are pivotal in the introduction and development of this algorithm: Evelyn Fix's Discriminatory Analysis paper from 1951 [19] and Thomas Cover's expansion on this concept in 1967 [20].

2.1.2 Mechanism

The K-Nearest Neighbours (KNN) algorithm operates on a straightforward principle of feature similarity, which makes it particularly effective for classification tasks. To understand how KNN classifies a new data point, let's break down its mechanism into two primary steps: selecting the nearest neighbours and determining the label.

Selecting the Nearest Neighbours:

Distance Measurement: When a new data point is introduced for classification, KNN begins by calculating the distance between this point and every other point in the training dataset. The most common distance metric for this calculation is the Euclidean distance, though other distance metrics can also be used depending on the nature of the data.

Identifying Nearest Neighbours: After calculating distances, the algorithm sorts these values and picks the 'k' shortest distances. The value of 'k' is predefined before the training and represents the number of nearest neighbours to consider. It's crucial to choose an appropriate 'k' value, as it influences the classification accuracy – too small a value can make the algorithm sensitive to noise, while too large a value might include irrelevant neighbours.

Determining the Label:

Voting Among Neighbours: Each of the 'k' nearest neighbours casts a 'vote' based on their class label. The basic idea is that similar data points (neighbours) will have similar labels.

Majority Wins: The new data point is assigned the class label that has the majority votes among the 'k' nearest neighbours. In case of a tie (i.e., two classes having the same number of votes), various tie-breaking strategies can be employed, such as choosing the class of the single nearest neighbour among the tied groups - which is what I have employed in my implementation.

This process exemplifies the simplicity and intuitiveness of KNN – it classifies based on the most common label among its closest neighbours, following the premise that similar things exist in close proximity.

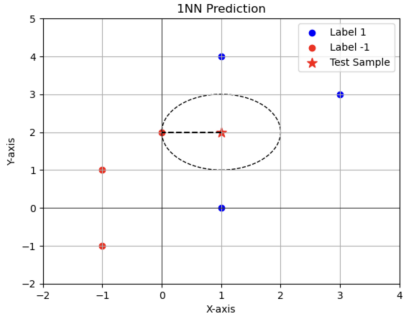


Figure 1: 1-Nearest Neighbour

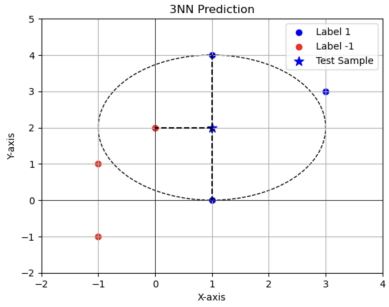


Figure 2: 3-Nearest Neighbours

In summary, the KNN algorithm for classification hinges on the concept

of feature similarity, where the class of a new data point is determined based on the classes of its nearest neighbours in the feature space. This method's effectiveness is highly dependent on the choice of 'k' and the distance metric used, making these crucial parameters in KNN's application.

2.1.3 Distance Metrics

A critical aspect of the K-Nearest Neighbours (KNN) algorithm is the method used to calculate the distance between data points. The choice of the distance metric can significantly influence the performance of the algorithm. Different types of distance metrics are suited to different types of data and applications. Here, we explore some of the most commonly used distance metrics in KNN:

Euclidean Distance is the most commonly used metric in KNN. It is the "ordinary" straight-line distance between two points in Euclidean space. Given two points, p and q, each with n dimensions, the Euclidean distance between p and q is defined as:

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

This distance is easy to understand and simple to compute, making it a natural choice for many applications.

Manhattan Distance, also known as City Block distance, is another useful metric, especially in urban settings. It is calculated as the sum of the absolute differences of their Cartesian coordinates. Formally, the Manhattan distance between two points p and q in n-dimensional space is:

$$d(p, q) = \sum_{i=1}^{n} |q_i - p_i|$$

This metric can be more useful than the Euclidean distance in certain scenarios, particularly when data points have grid-like structures.

Minkowski Distance is a generalized metric form that includes both Euclidean and Manhattan distances as special cases. It is defined as:

$$d(p, q) = (\sum_{i=1}^{n} |q_i - p_i|^p)^{1/p}$$

For $p = 2$, it becomes the Euclidean distance, and for $p = 1$, it is the Manhattan distance. This flexibility allows for more nuanced distance calculations, adaptable to different problem settings.

2.1.4 Advantages and Limitations

Advantages of K-Nearest Neighbors

- **Simplicity and Intuitiveness:** KNN is very straightforward and easy to understand, making it an excellent choice for beginners in machine learning. The concept of "the nearest neighbors" is intuitive and doesn't require complex mathematical understanding.
- **Non-Parametric Nature:** Since KNN makes no assumptions about the underlying data distribution, it is suitable for non-linear data. This feature makes it adaptable to many real-world scenarios where the data distribution is unknown.
- **Adaptability to Complex Decision Boundaries:** KNN can capture complex decision boundaries depending on the value of k.
- **Useful for Multiclass Problems:** KNN can handle multi-class settings without needing any adjustments, making it versatile in handling various classification tasks.

Limitations of K-Nearest Neighbors

- **Computationally and Memory Intensive:** KNN can be computationally expensive, especially with large datasets, as it requires calculating the distance to all training samples for each prediction. Since the algorithm stores all the training data, memory consumption can be high for large datasets.
- **Sensitive to the Choice of k and Distance Metric:** The performance of KNN heavily relies on the choice of the number of neighbours (k) and the distance metric used. An inappropriate choice of k or distance metric can lead to very poor performance.
- **Vulnerable to Irrelevant Features:** KNN can be significantly affected by the presence of irrelevant or redundant features because all features contribute equally to the distance computation.
- **Curse of Dimensionality:** Its performance degrades as the number of features (dimensions) increases, due to the difficulty in calculating distances in high-dimensional spaces.
- **Performance Deterioration with Noisy Data:** KNN is sensitive to noise in the dataset, as noisy features can distort the true distance between points.
- **Class Imbalance Problems:** KNN can be biased towards the majority class in cases of class imbalance.

2.1.5 Comparative Analysis with other ML Algorithms

KNN's simplicity has been emphasised throughout this section, and it remains as its key strength when compared to other algorithms, such as Decision Trees and parametic methods.

KNN vs. Decision Trees: One of the fundamental differences between KNN and decision trees lies in their approach to learning and prediction. KNN is a typical example of lazy learning and instance-based learning. Unlike decision trees, which are eager learners, KNN does not build a model during the training phase. It simply stores the data and performs computations only at the time of prediction. This characteristic leads to different performance implications:

- **Speed and Efficiency:** KNN can be slower at making predictions than decision trees, as it requires searching through the entire dataset for each query. In contrast, decision trees, once trained, can make predictions quickly by traversing the tree structure.
- **Memory Usage:** KNN is memory-intensive since it needs to store the entire dataset. Decision trees, after training, only require the tree structure for predictions, making them more memory-efficient.
- **Interpretability:** Decision trees offer a more interpretable model compared to KNN. The tree structure of decision trees provides clear rules and paths for decision-making, whereas KNN's predictions are based on the proximity to the nearest neighbors, which might not be as intuitive to understand.

KNN vs. Parametric Methods: KNN also differs significantly from parametric methods, such as linear regression or logistic regression. Parametric methods assume a specific form for the function that relates inputs to outputs and learn the parameters of this function from the training data. In contrast, KNN is a non-parametric method:

- **Model Assumptions:** KNN makes no assumptions about the underlying data distribution, making it more flexible in handling a variety of data types and distributions. Parametric methods, with their fixed structure, can be limited in this aspect.
- **Complexity and Overfitting:** Parametric methods can be prone to overfitting or underfitting if the assumed model form is not well-aligned with the true underlying relationships. KNN, with its instance-based approach, can adapt more closely to the data but may also capture noise, leading to overfitting, especially with a small value of 'k'.
- **Scalability:** In terms of scalability, parametric methods often scale better to large datasets, as they summarize the data into a set of parameters. KNN's need to store and search through the entire dataset can become a bottleneck as data size increases.

2.2 Decision Trees

Decision Trees are a prominent and widely used form of supervised machine learning algorithms, known for their simplicity, interpretability, and versatility in both classification and regression tasks.

2.2.1 Introduction

Among machine learning algorithms, Decision Trees stand out for their straightforward yet powerful approach to classification tasks. Characterized by their tree-like structure, they simplify decision-making by breaking it down into a series of binary choices, leading to clear, actionable outcomes. This makes Decision Trees not only easy to interpret to anyone but also highly adaptable to various classification challenges.

They are quite capable when it comes to transforming complex decision boundaries into a sequence of simpler, binary splits. This process starts at the root node and progresses through various levels, with each node representing a decision point based on one of the input features. The branches, stemming from these nodes, lead to new decision points or leaf nodes, which represent the final classification decision. This hierarchical structure captures the essence of decision-making in a way that is both logical and intuitive.

Focusing on the **Classification and Regression Tree (CART)** algorithm within the Decision Tree family, this project explores its application in classification tasks. The CART algorithm distinguishes itself through its binary splitting approach, where each node is split into two child nodes, offering a balanced and manageable way to dissect the data. For classification tasks, CART employs **Gini Impurity** as a measure of split quality. Gini Impurity is a statistical metric that quantifies the likelihood of incorrect classification, making it an effective tool for evaluating the purity of the split. By minimizing Gini Impurity at each step, CART guides the decision-making process toward the most informative features in the dataset, with the aim of making accurate predictions on the target label.

This report delves into the inner workings of the CART algorithm, demonstrating how its structured approach to decision-making can be effectively harnessed for classification purposes. By implementing CART from scratch and utilizing Gini Impurity as the key criterion for data partitioning, the project aims to provide an insightful exploration into one of the more fundamental yet very versatile algorithms in the field of machine learning.

My primary source of material when researching how this algorithm functions was from the original text by Leo Breiman, Classification and Regression Trees [21] as well as a chapter within Machine Learning by Tom Mitchell [27, pp 52-80].

2.2.2 Fundamentals of Tree Mechanism

Decision Trees operate on the fundamental principle of recursive partitioning of the dataset. This process begins at the root of the tree and involves splitting

the data based on feature values. Each node in the tree represents a feature in the dataset, and the branches represent the outcomes of the decision made at that node. This leads to further subdivisions in the tree until a leaf node is reached, which provides the final prediction or classification.

The decision at each node is made based on certain criteria, typically aiming to maximize the homogeneity of the resultant subsets. For categorical features, the splits are based on the different categories, while for continuous features, the tree selects a threshold value for splitting the data. This recursive partitioning continues until certain stopping criteria are met, which prevents the tree from growing indefinitely and potentially overfitting the data.

My implementation uses this idea of a threshold with splitting continuous data, the approach to categorical data is to be implemented soon.

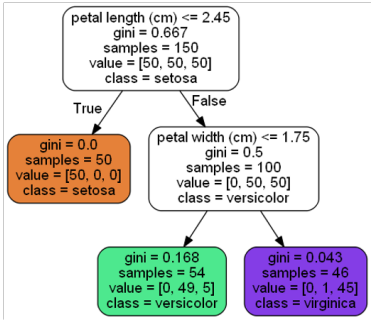


Figure 3: Example of a tree using Gini

2.2.3 Splitting, Stopping and Pruning Criteria

In Decision Trees, the splitting criterion is a crucial aspect that determines how the data at each node is divided. The CART algorithm utilizes Gini Impurity as a primary measure for this purpose. Gini Impurity is a statistical metric used to quantify the "purity" of a node in the tree. The objective is to minimize this impurity, ensuring that after the split, the subsets are as homogeneous as possible with respect to the target variable.

Gini Impurity: Gini Impurity measures the frequency at which any element of the dataset will be incorrectly labeled if it was randomly labeled according to the distribution of labels in the subset. The Gini Impurity for a set can be calculated using the formula:

$$Gini(I) = 1 - \sum_{j} p_j^2$$

Here, p_j represents the proportion of the samples that belong to class j in the set I. This formula implies that a node is 'pure' (Gini = 0) if all its samples belong to a single class.

We can think of p as representing the probability that a single sample picked from this once belongs to its corresponding class. We find the probabilities for each class, square them and sum these squared values. We then subtract the sum found from 1 to find the impurity value.

The goal of the CART algorithm is to find splits that decrease the Gini Impurity the most across the nodes it creates.

The algorithm considers all possible thresholds for each feature and calculates the Gini Impurity for the resulting partitions. The feature and the threshold that result in the highest impurity reduction are chosen to split the node. This process is repeated recursively for each new child node.

Stopping Criteria: The growth of the tree is regulated by stopping criteria to prevent overfitting. These criteria include:

- Setting a maximum depth for the tree.
- A minimum number of samples required to split a node.
- A minimum number of samples required to be at a leaf node.

These parameters ensure that the tree does not grow too complex and models the data well without capturing noise.

Pruning: After building the tree, pruning is performed to reduce its complexity and enhance its generalization abilities. Pruning involves cutting off branches of the tree that provide little predictive power. This is based on a cost-complexity trade-off, where smaller, simpler trees are preferred unless more complex trees provide substantial improvement in predictive ability. Pruning helps in reducing overfitting and improving the model's performance on unseen data.

By controlling the tree's growth and complexity through these mechanisms, Decision Trees can be a powerful tool for classification, striking a balance between fitting the training data well and maintaining good generalization to new, unseen data.

My implementation makes use of the Gini Impurity, using a fairly greedy strategy to find the most pure node at each split. It uses maximum depth, minimum number of samples at node and a node being totally pure as stopping criteria. Pruning is yet to be implemented, but it would be an ideal step to reduce overfitting in my implementation.

2.2.4 Advantages and Limitations

Advantages:

- **Interpretability:** Decision Trees are highly interpretable, making it easy to understand the decision-making process.
- **Handling Non-linear Relationships:** They can capture non-linear relationships between features and the target without the need for transformation.

- **No Need for Feature Scaling:** Decision Trees do not require normalization or standardization of features.
- **Versatility:** Capable of handling both numerical and categorical data.

Limitations:

- **Overfitting:** Prone to overfitting, especially in cases with many features or complex structures.
- **Sensitivity to Variance in Data:** Small changes in the data can lead to different tree structures.
- **Biased with Imbalanced Data:** Can create biased trees if some classes dominate.

2.2.5 Comparative Analysis with other ML Algorithms

Decision Trees, as a fundamental machine learning algorithm, have distinct characteristics when compared to other methods like K-Nearest Neighbors (KNN) and parametric approaches such as Support Vector Machines (SVMs) and Logistic Regression. Each algorithm has its strengths and weaknesses, making them suitable for different types of problems.

Decision Trees vs. K-Nearest Neighbors:

- **Learning Style:** While Decision Trees are eager learners that build a model based on the entire dataset before making predictions, KNN is a lazy learner that makes predictions using the dataset at the time of query. This difference impacts both training time and prediction time.
- **Interpretability:** Decision Trees are highly interpretable, as they provide a clear decision-making path from root to leaf. KNN, on the other hand, bases its predictions on the proximity to the nearest neighbors, which might not be as intuitive.
- **Sensitivity to Data Scale:** KNN is sensitive to the scale of data and requires feature scaling for optimal performance. Decision Trees are less sensitive to the scale of features.
- **Handling Non-linear Data:** Both algorithms can handle non-linear data, but Decision Trees can be more effective in capturing complex relationships due to their hierarchical structure.

Decision Trees vs. Parametric Methods:

- **Interpretability:** Decision Trees excel in interpretability due to their clear, hierarchical decision-making structure. In contrast, parametric methods, especially complex ones like SVMs with non-linear kernels or logistic regression models with many predictors, can be less transparent due to their mathematical formulations.

- **Handling Non-linear Relationships:** Decision Trees naturally adapt to non-linear relationships through their hierarchical structure of splits and decisions. Parametric methods, like logistic regression, can be extended to handle non-linearity using techniques such as feature transformation or kernel methods in SVMs.
- **Flexibility in Data Types:** Decision Trees are versatile in handling both categorical and numerical data directly. Parametric methods typically require numerical input, often necessitating additional steps such as one-hot encoding for categorical data.
- **Robustness to Overfitting:** Both Decision Trees and parametric methods can overfit data. Decision Trees can grow overly complex, capturing noise, but pruning helps mitigate this. Parametric methods, particularly those with many parameters, are also prone to overfitting, but regularization techniques can help.
- **Scalability and Efficiency:** Parametric methods, like logistic regression, can be computationally efficient for large datasets, summarizing information into parameters. Decision Trees, however, may become less efficient with large datasets or feature spaces, especially if the tree grows very deep or complex.

3 Model Evaluation Techniques

In this section, we'll cover concepts and techniques used to evaluate the performance of these models we've undertaken.

3.1 Bias And Variance

In the realm of machine learning, two key concepts that play a pivotal role in model performance are bias and variance. These terms describe the types of errors a machine learning model might encounter and are crucial in understanding the trade-offs in different model choices.

Bias refers to the error introduced by approximating a real-world problem, which may be complex, with a too-simplified model. In essence, bias measures how far off, on average, a model's predictions are from the actual values. High bias can cause a model to miss relevant relations between features and target outputs (underfitting), leading to poor performance on both training and unseen data.

Variance, on the other hand, refers to the model's sensitivity to fluctuations in the training dataset. A model with high variance pays a lot of attention to training data and can capture noise present in the dataset rather than the intended outputs (overfitting). Such a model performs well on its training data but poorly on any unseen data.

The relationship between bias and variance is a balancing act and is often referred to as **the bias-variance tradeoff**. Minimizing one typically leads to an increase in the other. For instance, a highly complex model (low bias) might fit the training data too closely, capturing noise and anomalies (high variance). Conversely, a too-simple model (high bias) might fail to capture important patterns, leading to poor prediction accuracy.

An ideal model is one that achieves a good balance between bias and variance, ensuring it performs well on both the training data and unseen data. This balance is often achieved through model selection, feature engineering, and tuning of model parameters.

Accuracy in this context is a measure of a model's overall correctness in classifying data or predicting outcomes. However, it's essential to consider that a highly accurate model on training data may not necessarily generalize well to new, unseen data if it has high variance.

3.2 Hold-Out Test Set

Quite a central process when evaluating machine learning models involves dividing our data into training and test sets. The test set, and this particular process, is sometimes more formally known as a **Hold-Out Test Set**. This technique involves partitioning the dataset into two segments: one that is used to train the model and another that is used to test and evaluate its performance.

Training Set: This is the portion of the dataset used to train the model. The training set allows the algorithm to identify patterns, learn the relationships

between features and the target variable, and build the model. The size of the training set can significantly influence the model's learning, with larger training sets generally providing more comprehensive learning opportunities.

Test Set: Once the model is trained, it is essential to evaluate its performance on data it hasn't seen before. This is where the testing set comes in. It is a separate portion of the dataset not used during the training phase. By assessing the model's performance on the testing set, we get a sense of how well the model generalizes to new, unseen data.

All in all, this is rather simple to implement and undertake - but the performance we find can heavily vary depending on the splits of our data. So our assessment can have a lot of variability. This issue is directly addressed with the following technique.

3.3 Cross-Validation

While the Hold-Out Test Set method provides an initial assessment of a model's generalization capability, it can lead to variability in performance evaluation due to different splits of the data. To address this issue and obtain a more robust evaluation, the technique of Cross-Validation is employed - and is generally utilised to evaluate a classification model's performance in making accurate predictions.

Cross-validation is a resampling procedure used to evaluate machine learning models on a limited data sample. The fundamental idea behind this technique is to divide the entire dataset into multiple subsets and then systematically use different subsets for training and testing the model. This approach ensures that every data point is used for both training and testing, providing a comprehensive assessment of the model's performance.

Essentially, we're carrying on this idea of splitting the dataset into training and test sets, and we're repeating it in several different segments or subsets of the dataset. We could name these several test sets as 'folds'.

With this in mind, we find that the most common form of Cross-Validation is **K-fold Cross-Validation**. In this method, the dataset is randomly partitioned into 'k' equal-sized folds. For each iteration, one of the 'k' subsets is used as the test set, and the remaining 'k-1' subsets are put together to form a training set. The model is then trained on this training set and validated on the test set. This process is repeated 'k' times, with each of the 'k' subsets used exactly once as the test set. The results from all 'k' iterations are then averaged to produce a single estimation.

In this project's implementation, we incorporate functions that return a list of accuracy values calculated on each fold, as well as the average of this list of values.

Another notable variant of Cross-Validation is **Leave-One-Out Cross-Validation** (LOOCV). In LOOCV, the 'k' in k-fold Cross-Validation is set equal to the number of observations in the dataset. This means that for a dataset with N observations, the model is trained N times, each time using all observations except one for training and the left-out observation for testing.

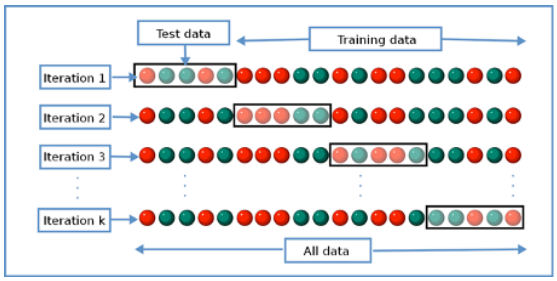


Figure 4: Illustration of K-fold Cross-Validation

LOOCV is particularly useful when dealing with small datasets, as it maximizes the amount of data used for training the model. Each iteration provides a test on a unique single data point, offering a thorough exploration of the model's performance across the entire dataset. However, LOOCV can be highly computationally intensive for larger datasets, as it requires the model to be retrained from scratch for each of the N iterations. I've found this to be the case in my implementation, especially for the tree model implementation.

3.4 Metrics

Various metrics are used to assess different aspects of a model's performance, with each metric offering unique insights. In this project, we focus on several key metrics: Accuracy, Precision, Recall, and the F1 Score.

Accuracy was introduced earlier as a measure of a model's overall correctness. It is the ratio of correctly predicted instances to the total instances in the dataset. While accuracy is straightforward and widely used, it may not always be the best indicator of a model's performance, especially in cases where the data is imbalanced.

Precision refers to the proportion of positive identifications that were actually correct. It is calculated as the ratio of true positives (correct positive predictions) to the sum of true and false positives. Precision is a critical measure in scenarios where the cost of a false positive is high.

Recall, also known as Sensitivity or True Positive Rate, measures the proportion of actual positives that were correctly identified. It is calculated as the ratio of true positives to the sum of true positives and false negatives. Recall becomes a key metric when the cost of a false negative is high.

F1 Score is the harmonic mean of Precision and Recall. It provides a single metric that balances both the concerns of precision and recall in one number. The F1 Score is particularly useful when we need a balance between precision and recall and there is an uneven class distribution (as is often the case in real-world scenarios).

4 Implementation

4.1 Overview

By the end of Term 1, I managed to implement proof-of-concept programs for the implementation of the K-Nearest Neighbours algorithm, and the Decision Tree algorithm using Gini Impurity. I implemented a train-test-split function to separate any loaded datasets. I also implemented a cross-validation module which contains functions that undertake K-Folds Cross-Validation and Leave-One-Out Cross-Validation on a given model with given data, and retrieve the accuracy scores from this process. I present all of this on several notebooks, which also essentially display the progress I made across the First Term. My final interim review notebook presents all of this work in one place, as well as an implementation of a Min-Max Scaler - the first implementation made thus far of data preprocessing.

4.2 Planning and Timescales

Before we dive into my implementation of the algorithms, I'll bring forward the project plan put in place almost two months ago. The timeline for Term 1 was created with a weekly outline of expected deliverables.

4.2.1 Term 1 Timeline

Timeframe	Tasks
Week 1 (Oct 2 - 6)	Focus on implementing Project Plan Handwritten example of NN algorithm Test out Scikit-Learn classifier functions in Jupyter Notebooks
Week 2 (Oct 9 - 13)	1NN on simple dataset Design NN data structures and UML diagrams Begin PoC Programs. Fully test DTs and NN on Scikit-Learn
Week 3 (Oct 16 - 20)	1NN on iris data, Work on multi-class data Begin implementing K-Neighbours on Iris data Demonstate KNN in Jupyter Notebooks, Start basic DTs
Week 4 (Oct 23 - 27)	Complete NN Report - 29th October Start working on DTs on Iris Data with binary classification Begin working on hold out test set validation programs
Week 5 (Oct 30 - Nov 3)	Start DT Report Continue working on DT data handling with missing values Start working on cross validation programs and test on KNN
Week 6 (Nov 6 - 10)	Use hold out tests on Decision Trees Implement K-folds Cross Validation on both algorithms Continue working on report
Week 7 (Nov 13 - 17)	Complete DT Report - 18th November Deal with any issues that have arisen over previous weeks Use different datasets from UCI (e.g breast cancer)
Week 8 (Nov 20 - 24)	Focus on completing Interim Report Amend any issues with supplementary reports if not yet finished Draw meaningful tests and conclusions from results
Week 9 (Nov 27 - Dec 1)	Work on Interim Report Prepare for Demonstration and Presentation Finalise project work + prepare for submission
Pres. Week (Dec 4 - 8)	Presentation Week! Submit project, and present work Make demo and presentation

4.3 KNN

4.3.1 1-NN

Following my Project Plan, my first step was to begin working on implementing One-Nearest Neighbours on a simple synthetic dataset. Fortunately, my lectures at Prof. Vovk's CS3920 Machine Learning course were very helpful with much

of KNN's implementation - and Vovk provided a useful example in the lectures illustrating the mechanism of Nearest Neighbours in two dimensions. I took inspiration from his dataset and focused on implementing a model that could work with this 2-D coordinate data, before moving on to the Iris dataset.

```
    import numpy as np

class OneNearestNeighbour:
    def __init__(self):
        self.training_data = None
        self.training_labels = None

    # training of data onto labels
    def fit(self, X, y):
        self.X_training_data = X
        self.y_training_labels = y

    # testing function, classifies using helper method which finds
    euclidean distance
    def predict(self, X):
        predictions = []
        for point in X:
            predictions.append(self._predict_point(point))
        return predictions


    def _predict_point(self, point):
        distances = [self._euclidean_distance(point, x) for x in
    self.X_training_data]
        # print(distances)
        nearest = np.argmin(distances)

        return self.y_training_labels[nearest]

    def _euclidean_distance(self, p1, p2):
        return np.sqrt(np.sum((np.array(p1) - np.array(p2))**2))
```

Listing 1: one_nn.py

This implementation is rather trivial. As is customary, fit() is used to train the model and predict() is used to make predictions on a test sample. The predict() method calls a helper function _predict_point() which predicts a label for every individual point given to it. This function calls upon the Euclidean distance function. The Euclidean Distance function basically utilises element-wise subtraction between two numpy arrays, utilising Numpy vectorization - which is generally far more efficient in practice compared to alternatives.

This general structure is maintained into the KNN implementation.

I demonstrated the way this works within my first Jupyter Notebook in Figure 5.

This notebook elaborates further on how we can also retrieve all the distances calculated and used to find the nearest neighbour for the test sample.

Once this was demonstrated, I focused on incorporating the iris[25] dataset. This also was relatively straightforward, as I essentially just utilised the load_iris()

We will use test data from one of Prof. Vovk's lectures to test this simple algorithm

In [2]:
```
data = [(0,3),(2,2),(3,3),(-1,1),(-1,-1),(0,1)]
labels = [1, 1, 1, -1, -1, -1]

test_sample = [(1,2)]
```

In [3]:
```
nn1 = NearestNeighbour()
nn1.fit(data,labels)
```

The algorithm has been fit to the data above. When testing the data, we are expecting the prediction to give the test sample a label of 1 - 2,2 would be the nearest neighbour.

In [4]:
```
print(nn1.predict(test_sample))
```

```
[1]
```

Figure 5: Notebook 1 - Fitting 1NN on simple synthetic data

function from the SKLearn [24] library. This way of loading iris remains incorporated throughout the project, though other datasets are handled more manually. I could alter this later and manually add the iris dataset as a txt file using numpy[23].

It was at this point that I realised I needed to split this dataset somehow, which brings us to the next step.

4.3.2 Train-Test-Split

It became quickly apparent that I needed to immediately implement a way to divide the iris dataset into a training and test set, if I'm to find anything useful about my KNN implementation on the dataset.

Though not anticipated by my Plan, this was the next thing I decided to focus on putting together.

Fortunately, I found this generally to be also rather simple to implement thanks to Python's list-slicing functionality.

Within a train_test_split module that can be imported, I wanted to implement a function of the same name that took inputs X and y as features and labels respectively. The size of the test set is the key parameter, and I thought it would be useful to not only input a ratio for the test set to the training set but also a specific number of samples for the size of the test set too.

Over time during development, I realised during testing for edge cases that it's important to implement intensive exception handling in this function- in case an erroneous value is passed as test set size, (for example 7.25).

It's also important that the dataset is randomly shuffled in some way, and so a complementary shuffle_data() method is called to handle this - taking X and y as inputs with an inputted random seed value - so that we can retrieve repeatable results when we need to.

This overall implementation is described in Listing 2. This is the final imple-

mentation after various iterations. Some comments and PyDoc documentation have been removed for the sake of fitting this here.

```python
def train_test_split(X, y, test_size=0.25, seed=None):  # Default test_size is now 0.25
    X = np.array(X) if not isinstance(X, np.ndarray) else X
    y = np.array(y) if not isinstance(y, np.ndarray) else y

    if len(X) == 0 or len(y) == 0:
        raise ValueError("Input data cannot be empty.")

    if len(X) != len(y):
        raise ValueError("The number of samples in X and y must be equal.")
    X, y = shuffle_data(X, y, seed)

    num_samples = len(y)

    if isinstance(test_size, float):
        if 0.0 < test_size < 1.0:
            train_ratio = num_samples - int(num_samples * test_size)
        else:
            raise ValueError("test_size as a float must be in the range (0.0, 1.0)")

    elif isinstance(test_size, int):
        if 1 <= test_size < num_samples:
            train_ratio = num_samples - test_size
        else:
            raise ValueError("test_size as an int must be less than the number of samples")

    else:
        raise ValueError("Invalid test_size value")

    X_train, X_test = X[:train_ratio], X[train_ratio:]
    y_train, y_test = y[:train_ratio], y[train_ratio:]

    return X_train, X_test, y_train, y_test

def shuffle_data(X, y, seed=None):
    rng = np.random.RandomState(seed) if seed is not None else np.random

    indices = np.arange(X.shape[0])
    rng.shuffle(indices)
```

Listing 2: train_test_split.py

At this point, it was now finally possible to train and predict our 1-Nearest Neighbour implementation on the iris dataset. This was demonstrated in the second Jupyter Notebook.

And here we have our first evaluation of an algorithm! This was the rudimentary first step of the project, but it was a good foundation for me to work on

```
In [2]: #splitting data into training and test set using train_test_split
        iris_X_train, iris_X_test, iris_y_train, iris_y_test = split(iris_features,iris_labels, seed=20004)

        #printing sizes of training and test sets to demonstrate ratio
        print(iris_X_train.shape)
        print(iris_X_test.shape)

        (113, 4)
        (37, 4)

In [3]: #fitting training data
        NN = OneNearestNeighbour()
        NN.fit(iris_X_train,iris_y_train)

In [4]: #predicting labels for test data
        iris_predictions = NN.predict(iris_X_test)

In [5]: #printing correct predictions with accuracy value
        print(iris_predictions == iris_y_test)
        accuracy = np.mean(iris_predictions == iris_y_test)
        print("Accuracy :", accuracy)

        [ True  True  True  True False  True  True  True  True False  True  True
          True  True  True  True  True  True  True  True  True False  True  True
          True  True  True  True  True  True  True  True  True  True  True  True
          True]
        Accuracy : 0.918918918918919

In [6]: print("Error :", 1- accuracy)

        Error : 0.08108108108108103
```

Figure 6: Notebook 2 - Implementing 1NN on iris hold-out test set

the main K-Nearest Neighbours proof-of-concept program. Jupyter Notebooks were also my main way of demonstrating my progress in a visible way.

4.3.3 Implementing K-Neighbours

Implementing K-Nearest Neighbours followed a similar structure, but instead involved making various alterations to the way that the helper function, _predict_point(), works. We begin by adding an attribute, k to the constructor of the classifier - it's a parameter that we can pass to the classifier during initialisation.

When a point is passed from the main predict() function to the helper function, we now need to keep track of a list of points and their corresponding distances from the test point in a list. We do this with k_nearest in the snipped shown below, where we make a list of tuples storing the index and distance, initialised as '-1' and infinity respectively.

We then iterate through the training set and compare each point to our test point - by iterating through our k-nearest list and replacing when we find a training point that is lower than one we've kept stored. By the end of these loops, we have a list of nearest neighbours, with their indices and distances

From this list, we generate a new list of tuples, containing the label of the stored point and its corresponding distance. Doing this makes it easier for us

to vote for a label and handle ties if necessary

We then iterate through this list and count the instance of each label as well as the minimum distance found from each label compared to the test point. Finally, we use a max function to find our majority vote on the predicted label. I use a lambda function here to establish a condition such that if there's a tie in votes, we pick the closest label to the test point to predict our point.

I made the choice to handle ties when voting for K by simply predicting the label of the closest point - it seemed like the most intuitive way to break a tie without bringing in weighted voting - which could be implemented later but I was worried of making this calculation too inefficient.

```
def _predict_point(self, point):
    k_nearest = [(-1, float('inf')) for _ in range(self.k)]  # [(
    index, distance), ...]

        for i, training_point in enumerate(self.X_training_data):

            distance = self._euclidean_distance(point,
    training_point)

            # Check if the distance is smaller than the current k-
    nearest distances
            for j, (idx, dist) in enumerate(k_nearest):
                if distance < dist:
                    k_nearest.insert(j, (i, distance))
                    k_nearest = k_nearest[:self.k]  # Keep only the
     k-nearest distances
                    break
        # Get labels and distances of the k-nearest points
        k_labels_distances = [(self.y_training_labels[idx], dist)
    for idx, dist in k_nearest]

        # Count the occurrences of each label and store minimum
    distance for each label
        label_counts = {}
        for label, dist in k_labels_distances:
            if label not in label_counts:
                label_counts[label] = [0, float('inf')]
            label_counts[label][0] += 1  # Increase count
            label_counts[label][1] = min(label_counts[label][1],
    dist)  # Store the minimum distance

        # Find the label with the maximum count and if there's a
    tie, choose the one with the smallest distance
        majority_label = max(label_counts, key=lambda x: (
    label_counts[x][0], -label_counts[x][1]))

        return majority_label
```

Listing 3: knn.py - _predict_point()

This method is called for every point in the test set that needs to be predicted and thus the predict() method returns a list of predicted labels.

At this point, we can now start using KNN on our iris data.

In Figure 7 we can see KNN getting initialised at a 3-Nearest Neighbours

```
In [2]: from sklearn.datasets import load_iris

        iris = load_iris()
        threeNN = knn.KNearestNeighbours(k=3)
```

```
In [3]: #%load_ext autoreload
        import sys
        sys.path.append("..")
        from train_test_split import train_test_split
        iris_X = iris['data']
        iris_y = iris['target']
        iris_X_train, iris_X_test, iris_y_train, iris_y_test = train_test_split(iris_X, iris_y, seed=345)
        print(iris_X_train.shape)
        print(iris_X_test.shape)
        print(iris_y_train.shape)
        print(iris_y_test.shape)

        (113, 4)
        (37, 4)
        (113,)
        (37,)
```

```
In [4]: threeNN.fit(iris_X_train, iris_y_train)
        pred_threeNN = threeNN.predict(iris_X_test)
        print(pred_threeNN == iris_y_test)
        accuracy = np.mean(pred_threeNN == iris_y_test)
        error = 1 - accuracy
        print("Accuracy: ", accuracy)

        [ True  True  True  True  True  True  True  True  True  True  True  True
          True  True  True  True  True  True  True  True  True  True  True  True
          True  True  True  True  True  True  True  True  True  True  True False
          True]
        Accuracy:  0.972972972972973
```

Figure 7: Notebook 3 - Initialising and predicting with 3 nearest neighbours

classifier. In code block 4, we can see the predict() function getting utilised and a list of instances of when these predictions are made correctly.

Within this notebook, I also investigated how the value of the hyperparameter K affected the accuracy of the KNN classifier.

We are essentially expecting our accuracy to decrease once the value of K becomes exceedingly high. I used a loop to iterate through various values of k and graphed the accuracies using matplotlib [28]. In Figure 8 we can see that KNN performs surprisingly well.

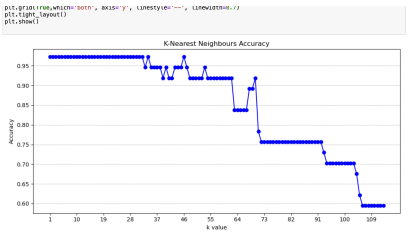


Figure 8: Notebook 3 - Evaluating k on iris data

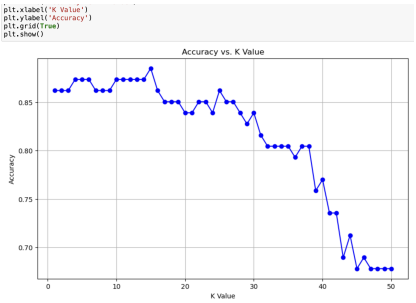


Figure 9: Notebook 3 - Evaluating k on ionosphere data

I investigated this further by introducing the ionosphere [29] dataset, an-

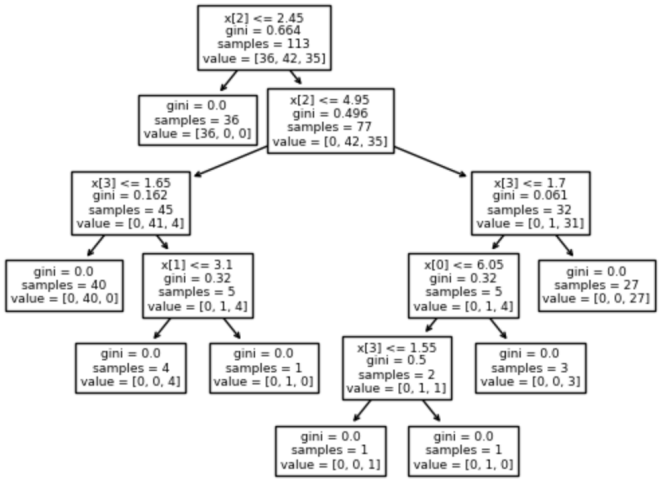
other UCI dataset that needs minimal preprocessing by many more features per sample. Here we can see in Figure 9 that KNN retrieves lower accuracy values in general, and the accuracy falls more sharply. Likely due to the much larger number of features in ionosphere.

It's at this point that I began to work on the Tree algorithm implementation.

4.4 Classification Tree

4.4.1 SKLearn Implementation

Implementing the tree algorithm was possibly the most challenging aspect of development in Term 1. I felt unfamiliar with the way that the fit() and predict() functions worked with Trees, so I utilised the SKLearn [24] library in Notebook 4 to understand what I should expect at a higher level from my implementation.




```
In [4]: cart.predict(x_test)

Out[4]: array([2, 2, 0, 1, 0, 2, 1, 2, 2, 1, 2, 0, 0, 0, 0, 2, 0, 0, 2, 1, 2, 2,
               0, 0, 0, 2, 1, 1, 0, 1, 2, 2, 0, 2, 2, 2, 0])
```

Figure 10: Notebook 4 - Investigating using SKLearn

4.4.2 Building the tree

When fit() is called, we want our classifier to build a tree that models that training data. To do this we need to essentially, find a way to split the data continuously and recursively, such that the nodes produced are more pure than their roots. We also need to find a way to calculate this impurity and use it to

traverse all the training data's features and values, to find the best thresholds that will produce this split.

After considering how the gini impurity should be implemented, I realised it's important to incorporate a weighted gini impurity that we can combine across both branches, so that we can aim to find a left and right branch, that have the lowest possible combined impurity. I realised that the size of both branches is something that needs to be accounted for, we want larger branches to have a greater influence if they have a low purity, as we're able to classify a larger group of samples in a split.

```
    # Gini index function to evaluate the quality of a split.
    def gini_index(self, groups, classes):
        # Count all samples at split point
        n_instances = float(sum([len(group) for group in groups]))
        # Sum weighted Gini index for each group
        gini = 0.0 # Initialize Gini index.
        for group in groups:
            size = float(len(group))
            if size == 0:  # Avoid division by zero
                continue
            score = 0.0
            # Score the group based on the score for each class
            for class_val in classes:
                p = [row[-1] for row in group].count(class_val) /
    size # Proportion of each class.
                score += p * p
            # Sum weighted Gini index for each group.
            gini += (1.0 - score) * (size / n_instances)
        return gini
```

Listing 4: classification_tree.py - gini_index()

Using this as a helper function, we have a way to determine the quality of our splits.

Looking at this at the higher level, we want our fit() function to build a tree, and we'd want a build_tree() function to handle the split finding. I approached this in the following manner.

```
    def fit(self, X, y):
    if len(X) == 0 or len(y) == 0:
            raise ValueError("Cannot fit a tree with an empty
    dataset")


        dataset = np.column_stack((X, y)).tolist()
        #print(dataset)
        self.root = self.build_tree(dataset, self.max_depth, self.
    min_size)
```

Listing 5: classification_tree.py - fit()

The fit() function combines the features and labels into a single dataset using numpy's column_stack() functionality, and we use to_list() to arrange this as a list of lists. What we have is, in essence, a 2-D array where each row represents a sample of features and a label at the end of the row. We now have a neat structure to manipulate within the tree.

```
    def build_tree(self, train, max_depth, min_size):
    root = self.get_split(train) # Find the best initial split for
    the root.
        self.split(root, max_depth, min_size, 1) # Recursively
    split the tree from the root node.
        # The value 1 above is passed as the first layer depth from
     the root
        return root
```

Listing 6: classification_tree.py - build_tree()

The tree-building function is then called by fit() and this method calls get_split() to find the best split at this node, followed by the split() function which essentially performs this split on the tree and also assesses the stopping criteria to determine whether these nodes should continue being split, or if they can become leaf nodes - in which case the to_terminal() method is called.

Within get_split(), the test_split() method is used to iterate through various candidate splits to help search for an ideal split. It's worth noting that this implementation is a rather greedy and inefficient algorithm, that is committed to finding the best possible split. This is something I hope to optimise to reduce training time and improve performance - by reducing overfitting.

```
     # Method to find the best place to split the dataset.
    def get_split(self, dataset):
    class_values = list(set(row[-1] for row in dataset)) # Get the
    unique class values.
        b_index, b_value, b_score, b_groups = 999, 999, 999, None
        for index in range(len(dataset[0]) - 1): # Iterate over all
     features
            for row in dataset:
                groups = self.test_split(index, row[index], dataset
    ) # Test split on each unique feature value.
                gini = self.gini_index(groups, class_values) #
    Calculate Gini index for the split.
                if gini < b_score: # Check if we found a better
    split.
                    b_index, b_value, b_score, b_groups = index,
    row[index], gini, groups
        return {'index': b_index, 'value': b_value, 'groups':
    b_groups} # Return the best split.
```

Listing 7: classification_tree.py - get_split()

The method for finding the split is shown above.

Other helper methods I implemented include is_homogenous(), which checks if the node only contains samples of the same class, and get_depth(), which finds how deep the tree has managed to reach from the root - which mostly helped me with checking if the tree is being constructed the way I expect it to, and at what point the stopping criteria prevents the tree from reaching the maximum depth.

Finally, mostly to maintain an understanding of the structure and to retain my sanity, I made a print_tree() which prints the created tree from training to the terminal. Perhaps the only implementation I've personally put together that shows a visualisation, so I'm somewhat proud of its implementation.

```
In [2]: tree_iris.fit(X_train, y_train)
        tree_iris.print_tree(feature_names = iris.feature_names)
        depth = tree_iris.get_depth()
        print("Depth :", depth)

petal length (cm) <= 3.0
Left:
|   --> Class: 0.0
Right:
|   petal length (cm) <= 5.0
|   Left:
|   |   petal width (cm) <= 1.7
|   |   Left:
|   |   |   --> Class: 1.0
|   |   Right:
|   |   |   sepal width (cm) <= 3.2
|   |   |   Left:
|   |   |   |   --> Class: 2.0
|   |   |   Right:
|   |   |   |   --> Class: 1.0
|   Right:
|   |   sepal length (cm) <= 6.1
|   |   Left:
|   |   |   sepal length (cm) <= 6.0
|   |   |   Left:
|   |   |   |   --> Class: 2.0
|   |   |   Right:
|   |   |   |   sepal width (cm) <= 2.7
|   |   |   |   Left:
|   |   |   |   |   --> Class: 2.0
|   |   |   |   Right:
|   |   |   |   |   --> Class: 1.0
|   |   Right:
|   |   |   --> Class: 2.0
Depth : 5
```

Figure 11: print_tree() example

4.4.3 Making Predictions

In theory, once we've made our tree from training - making predictions is relatively quick and straightforward. We simply take our test sample and traverse the decision nodes in the tree, until we can make our prediction at a leaf node.

Similar to how KNN works, we have a predict() function and a helper _predict() function that handles each sample getting passed to it which predict iterates.

```
    def predict(self, X):
        predictions = [self._predict(self.root, row) for row in X]
        return np.array(predictions)

    def _predict(self, node, row):
        if row[node['index']] < node['value']: # Check condition at the node.
            if isinstance(node['left'], dict): # If the left child is a dictionary, it's another decision node.
                return self._predict(node['left'], row)
            else:
                return node['left'] # If it's not a dictionary, it's a terminal node.
        else:
            if isinstance(node['right'], dict):
                return self._predict(node['right'], row)
            else:
                return node['right']
```

Listing 8: classification_tree.py - predict() functions

4.5 Cross-Validation

4.5.1 Retrieving K-Folds

Once both algorithms has been implemented, I wanted to Implement K-folds Cross-Validation immediately to properly evaluate and compare these two algorithms at how well they made predictions. Approaching this wasn't too difficult compared to the Tree, and the fold-making aspect essentially had a lot of similarities with the train-test-split implementation, with more iterations.

I approached this by making a function, _k_folds(), that separates the set given into the number of folds given. This essentially returns a list of folds, comprised of training and test data for features and labels - so four lists of data within each 'fold' of the folds lists.

```
    def _k_folds(X, y, k=5, seed=None) :
     X, y = split.shuffle_data(X,y,seed)
num_of_samples = len(y)
# Check if 'k' is a valid value
#K can't be 1, we accept 2 and higher
if k < 2 or k > num_of_samples or not isinstance(k, int):
    raise ValueError(f"'k' must be an integer between 2 and {num_of_samples}, got {k}")
indices = np.arange(num_of_samples)

# Calculate the standard fold size
fold_size = num_of_samples // k
remainder = num_of_samples % k
#unused but may be useful if we optimise remainder-handling

# Initialize an array to store the training and testing sets for each fold
folds = []

current = 0
for i in range(k):
    if i < k - 1:
        start, stop = current, current + fold_size
    else:
        # For the last fold, use the remaining samples
        start, stop = current, num_of_samples

    test_indices = indices[start:stop]
    train_indices = np.concatenate([indices[:start], indices[stop:]])

    X_train, X_test = X[train_indices], X[test_indices]
    y_train, y_test = y[train_indices], y[test_indices]

    folds.append(((X_train, y_train), (X_test, y_test)))

    current = stop

return folds
```

Listing 9: cross_validation.py - _k_folds()

I realised that there are various ways to approach the problem regarding the remainder of samples that don't evenly fit into the division of folds - my implementation simply allowed the final fold to handle the remainder, so this fold could be larger or smaller in comparison to the rest. This could be optimised a little, such that the remainder is evenly distributed across all the folds, but for now I chose the simpler approach. The caveat is if a significantly large number of folds is requested (say where $k > (n/2)$), the final fold will be quite unusually sized in comparison to the rest. But this is an unusual situation to be in anyway.

4.5.2 Calculating the scores

I decided, for simplicity, to allow a function to take the model, the features and labels, and the number of folds with a shuffle seed - all as parameters. And the function will perform all the training and testing using the model's defined fit() and predict() functions and retrieve a list of accuracy scores to be interpreted.

```
    def k_folds_accuracy_scores(model, X, y, k=5, seed=None):
     # Check if the model has 'fit' and 'predict' methods
    if not hasattr(model, 'fit') or not callable(getattr(model, '
    fit')):
        raise AttributeError("The provided model does not have a
    callable 'fit' method.")
    if not hasattr(model, 'predict') or not callable(getattr(model,
     'predict')):
        raise AttributeError("The provided model does not have a
    callable 'predict' method.")


    # Generate folds using the previously defined k_folds function
    folds = _k_folds(X, y, k, seed)

    # List to store the scores from each fold
    scores = []

    # Iterate over each fold
    #n=0
    #mismatch = 0
    for (X_train, y_train), (X_test, y_test) in folds:
        # Fit the model on the training set
        model.fit(X_train, y_train)

        # Predict on the test set
        y_pred = model.predict(X_test)

        # Compute the score - here, we use accuracy as an example
        accuracy = np.mean(y_pred == y_test)


        # Append the score to the list
        scores.append(accuracy)


    return scores
```

Listing 10: cross_validation.py - accuracy scores

This was an exciting thing to put together, as it would essentially be the first time all these abstractions I've made can finally fit together in one function. Implementation here was relatively simple, but it was important that exception handling here was robust, as we're assuming that the models in place have the functions we need to retrieve these performance scores.

Once this was implemented, it was natural to make more functions that retrieves the mean of these scores, and also perform LOOCV. Implementing these was trivial, as it essentially just called the function above.

4.6 Data Preprocessing

Data preprocessing hasn't been covered a lot in this report but it's significance became far more apparent to me rather late in the term, when I realised there's many factors preventing me from using these algorithms on more interesting datasets. My supervisor also emphasised the importance of normalising data that will be trained by KNN, as KNN is scale-sensitive.

Due to this, I very quickly implemented a MinMaxScaler - which essentially fits all data passed to it on a scale from 0 to 1. The scaler is in the form of a class, which means that it can be fitted in accordance to a particular training set, and then apply that same transformation to a test set, as is necessary for a scaler in ML settings.

```
import numpy as np

class MinMaxScaler:
    def __init__(self):
        self.min_ = None
        self.max_ = None
        self.range_ = None

    def fit(self, X):
        """Fit the scaler to the data."""
        self.min_ = np.min(X, axis=0)
        self.max_ = np.max(X, axis=0)
        self.range_ = self.max_ - self.min_
        # Handle zero range to avoid division by zero
        self.range_[self.range_ == 0] = 1
        return self


    def transform(self, X):
        """Transform the data using the fitted scaler."""
        if self.min_ is None or self.max_ is None:
            raise RuntimeError("The scaler has not been fitted yet.
")
        return (X - self.min_) / self.range_

    def fit_transform(self, X):
        """Fit to data, then transform it."""
        return self.fit(X).transform(X)
```

Listing 11: preprocessing.py - MinMaxScaler

This module will come useful in future implementations, I see this class as the first of various other functions in this module, with handling categorical data and missing data.

4.7 Unit Tests

Testing was an imperative past of development, especially as the complexity of these algorithms started to increase. The most important issue was ensuring that data was handled between components correctly and securely. I kept my tests in a folder named 'tests' and used the 'unittests' library within the standard Python [22] library to set up my test suite.

Below I've added an example of some unit tests from the tree test suite:

```
    def test_gini_index(self):
        """Test the Gini index calculation."""
        tree = ClassificationTree()
        groups = [[[1, 0], [2, 0]], [[3, 1], [4, 1]]]
        classes = [0, 1]
        gini = tree.gini_index(groups, classes)
        self.assertGreaterEqual(gini, 0)
        self.assertLessEqual(gini, 1)

    def test_tree_depth(self):
        """Test if the tree respects the maximum depth limit."""
        tree = ClassificationTree(max_depth=3)
        tree.fit(self.X, self.y)
        depth = tree.get_depth()
        self.assertLessEqual(depth, 3)

    def test_homogeneity_check(self):
        """Test if the is_homogeneous function correctly identifies
     a homogeneous group."""
        tree = ClassificationTree()
        homogeneous_group = [[1, 0], [2, 0]]
        heterogeneous_group = [[1, 0], [2, 1]]
        self.assertTrue(tree.is_homogeneous(homogeneous_group))
        self.assertFalse(tree.is_homogeneous(heterogeneous_group))


    def test_terminal_node_creation(self):
        """Test if terminal nodes are correctly created from a
    group of samples."""
        tree = ClassificationTree()
        group = [[0,0], [1, 0], [2, 0], [3, 1], [4, 1]]
        terminal_node = tree.to_terminal(group)
        # The most common class in the group is 0
        self.assertEqual(terminal_node, 0)
```

Listing 12: test_tree.py - example tests

Listing through the unit tests in each of my test suites:

- K-Folds has 8 unit-tests
- KNN has 19 unit-tests

- The MinMaxScaler has 2 unit-tests
- Tree has 10 unit-tests
- Train-Test-Split has 15 unit-tests

As I begin to work on unifying my programs into a standalone interface, these tests will help greatly in ensuring that the functionality of my algorithms and other data-handling functions is retained.

4.8 Datasets

The datasets used in this project include the Iris [25] dataset, the ionosphere [29] dataset and the banknote authentication [30] dataset - all from the UCI repository [26]. These datasets are all comprised of continuous, complete data and are well suited towards the classification task with specified target labels for each sample. This worked perfectly for my current implementation

This is discussed further in Evaluation - but I had plans to implement the Breast Cancer dataset following my project plan. Unfortunately, due to my programs not being able to handle missing data, this dataset couldn't be incorporated and the choice of datasets was rather limited.

4.9 Jupyter Notebooks

Jupyter Notebooks [31] are mentioned quite a few times in this section. These 'notebooks' are essentially a succinct interface that's very helpful with tracking a workflow in Machine Learning with its mix of code blocks and markdown blocks. In the context of my project, Jupyter Notebooks have been instrumental in not only developing and testing the algorithms but also in visualizing data and results in a coherent and interactive manner. They provide a platform where code, output, graphical visualizations, and explanatory text can coexist, making it an ideal environment for iterative coding, experimenting with data, and drawing conclusions.

They've served as a useful tool in tracking my progress in implementation, from the simple 1NN implementation, up until we've implemented scaling and K-Folds Cross-Validation on both algorithms. They've been a useful playground for me to gain insights into my models before building the interface.

5 Evaluation and Reflection

When comparing this progress with my success criteria outlined in section 1.4.2, I think I successfully achieved the base goal of implementing two algorithms, evaluating them in some way, on some kind of dataset. I think this term has been deeply insightful for me in understanding the intricacies of putting these algorithms together, and how rigorous one must be in evaluating that they function correctly. I've learned a lot about how to handle training and test sets properly, and avoiding data snooping during the tuning process. I feel quite fortunate to be working such a relevant project topic. In this section I'll list some issues I faced in this term and how I may try to amend them moving forwards.

5.1 Approach

5.1.1 Underestimating Preprocessing

It seems obvious in retrospect, but I feel that I'm unable to observe much potential in my algorithms until I've managed to implement more data handling. I think that what makes these algorithms interesting is the data that they perform on, and I'm limited to what I've chosen due to the lack of work in this aspect.

I would blame this on the fact that my project plan, as airtight as it seemed at the time, simply didn't consider how early on it needed to be considered into the workflow.

5.1.2 Time and Task Management

It's a tale as old as time itself, but I unfortunately did struggle with keeping up with the rigidity of my project plan in term 1, especially with the deadlines for my reports and my scheduled diary entries - which looking back, were rather lofty ambitions when considering the other responsibilities that come into play in final year.

To go deeper into this, I would add that much of this was also from hesitation to push forward changes that I did not feel certain would work in the long-term. The largest gap in my workflow this term coincides with the time I began trying to implement the core structure of my tree implementation - and I think I would have struggled less if I broke the challenge down into segments in the same way that I did with the Nearest Neighbours implementation.

5.2 Implementation

5.2.1 Inefficiencies in Tree Splitting

My algorithm when searching for the next split in the tree-building process is a greedy one. It's searching for the optimal split through the entire set of features and iterating through every single one. Not only is this inefficient, but

data preprocessing may unveil that this is unnecessary. Using the ionosphere dataset as an example, there are various features that don't show much variance whatsoever and likely can be either ignored or have less priority during the search for the split - in essence, investigating feature importance of my dataset. It's also likely that I can utilise NumPy's vectorisation capabilities to find a more efficient approach, or implementing some kind of parallelization.

5.2.2 Tree Nodes Lacking Information

Essentially, my implementation when building the tree is yet to give me much information about how many samples are allocated to each leaf node, as is found in the SKLearn implementation. This is quite a vital oversight, as I'm unsure of where inefficiencies may lay within my implementation until I am able to evaluate this. However, amending this may involve making larger changes to the data structure of the tree implementation, which will require some time and planning.

6 Future Developments and Objectives

With the close of this term, I feel quite driven to deliver changes and make progress upon the foundation I've built so far.

6.1 Data Preprocessing

The most immediate plan I have, right after the interim submission - is to work on handling missing and categorical data. For the former, I hope to implement some form of listwise deletion for any sample with a missing feature. If this shrinks the set too much, I could investigate with some form of statistical inputation.

For categorical data, I intend on working on an encoder of some kind - one-hot or label - and investigating how I may need to use this for both ordinal and nominal data samples. All of this will take place in the preprocessing.py module that's in place already.

6.2 Performance Metrics

A common mantra in the machine learning realm is that "accuracy is not enough". I intend to build a metrics module that automates the process of finding the accuracy, and also checks the precision and recall where it's applicable to do so. I believe this works in binary classification scenarios, so I'll need to investigate how I can implement this with multi-class data.

6.3 Hyperparameter Tuning

As shown in my interim review Jupyter Notebook - tuning the hyperparameters for a model is rather repetitive. I'd like to investigate if I could integrate this

into an automated process. This may be necessary when implementing my standalone interface.

6.4 Interface

The end-goal for this project is to have a standalone interface using TKinter or QtPy5. I aim to have some implementation of this by the end of January next term.

6.5 Third Algorithm

I think I could add a lot of depth to this project by introducing one more algorithm to compare, ideally a parametric one to draw comparisons with these two. I've been considering either Logistic Regression or an SVM, though I think the latter could be very ambitious considering the challenges faced so far.

References

[1] E. Elahi *et al.*, "Reinforcement learning for budget constrained recommendations," Aug 2022. [Online]. Available: https://netflixtechblog.com/reinforcement-learning-for-budget-constrained-recommendations-6cbc5263a32a

[2] F. Tomasi, J. Cauteruccio, S. Kanoria, K. Ciosek, M. Rinaldi, and Z. Dai, "Automatic music playlist generation via simulation-based reinforcement learning," in *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, ser. KDD '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 4948–4957. [Online]. Available: https://doi.org/10.1145/3580305.3599777

[3] A. Singhal, P. Sinha, and R. Pant, "Use of deep learning in modern recommendation system: A summary of recent works," *International Journal of Computer Applications*, vol. 180, no. 7, p. 17–22, Dec. 2017. [Online]. Available: http://dx.doi.org/10.5120/ijca2017916055

[4] T. A. Nguyen and S. Chaudhuri, "Look and talk: Natural conversations with google assistant," Jul 2022. [Online]. Available: https://blog.research.google/2022/07/look-and-talk-natural-conversations.html

[5] T. Capes, P. Coles, A. Conkie, L. Golipour, A. Hadjitarkhani, Q. Hu, N. Huddleston, M. Hunt, J. Li, M. Neeracher *et al.*, "Siri on-device deep learning-guided unit selection text-to-speech system." in *Interspeech*, 2017, pp. 4011–4015. [Online]. Available: https://www.isca-speech.org/archive/interspeech_2017/capes17_interspeech.html

[6] A. Choudhury and D. Gupta, "A survey on medical diagnosis of diabetes using machine learning techniques," *Recent Developments in Machine Learning and Data Analytics: IC3 2018*, vol. 740, pp. 67–77, 2018. [Online]. Available: doi.org/10.1007/978-981-13-1280-9{_}6

[7] S. Zhang, S. M. H. Bamakan, Q. Qu, and S. Li, "Learning for personalized medicine: A comprehensive review from a deep learning perspective," *IEEE Reviews in Biomedical Engineering*, vol. 12, pp. 194–208, 2019. [Online]. Available: https://doi.org/10.1109/RBME.2018.2864254

[8] H. Chen, O. Engkvist, Y. Wang, M. Olivecrona, and T. Blaschke, "The rise of deep learning in drug discovery," *Drug Discovery Today*, vol. 23, no. 6, pp. 1241–1250, 2018. [Online]. Available: https://doi.org/10.1016/j.drudis.2018.01.039

[9] A. M. Ozbayoglu, M. U. Gudelek, and O. B. Sezer, "Deep learning for financial applications: A survey," *Applied Soft Computing*, vol. 93, p. 106384, 2020. [Online]. Available: https://doi.org/10.1016/j.asoc.2020.106384

[10] K. Tsolaki, T. Vafeiadis, A. Nizamis, D. Ioannidis, and D. Tzovaras, "Utilizing machine learning on freight transportation and logistics applications: A review," *ICT Express*, 2022. [Online]. Available: https://doi.org/10.1016/j.icte.2022.02.001

[11] D. Rolnick, P. L. Donti, L. H. Kaack, K. Kochanski, A. Lacoste, K. Sankaran, A. S. Ross, N. Milojevic-Dupont, N. Jaques, A. Waldman-Brown, A. S. Luccioni, T. Maharaj, E. D. Sherwin, S. K. Mukkavilli, K. P. Kording, C. P. Gomes, A. Y. Ng, D. Hassabis, J. C. Platt, F. Creutzig, J. Chayes, and Y. Bengio, "Tackling climate change with machine learning," *ACM Comput. Surv.*, vol. 55, no. 2, feb 2022. [Online]. Available: https://doi.org/10.1145/3485128

[12] T. Deng, "A survey of convolutional neural networks for image classification: Models and datasets," in *2022 International Conference on Big Data, Information and Computer Network (BDICN)*, 2022, pp. 746–749. [Online]. Available: https://doi.org/10.1109/BDICN55575.2022.00145

[13] Z. C. Lipton, J. Berkowitz, and C. Elkan, "A critical review of recurrent neural networks for sequence learning," 2015. [Online]. Available: https://doi.org/10.48550/arXiv.1506.00019

[14] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1810.04805

[15] K. Ethayarajh, "How contextual are contextualized word representations? comparing the geometry of bert, elmo, and gpt-2 embeddings," 2019. [Online]. Available: https://doi.org/10.48550/arXiv.1909.00512

[16] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016. [Online]. Available: https://doi.org/10.1038/nature16961

[17] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv preprint arXiv:1712.01815*, 2017. [Online]. Available: https://doi.org/10.48550/arXiv.1712.01815

[18] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788. [Online]. Available: https://doi.org/10.1109/CVPR.2016.91

[19] E. Fix and J. L. Hodges, "Discriminatory analysis. nonparametric discrimination: Consistency properties," *International Statistical Review*

/ *Revue Internationale de Statistique*, vol. 57, no. 3, pp. 238–247, 1989. [Online]. Available: https://doi.org/10.2307/1403797

[20] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Transactions on Information Theory*, vol. 13, no. 1, pp. 21–27, 1967. [Online]. Available: https://doi.org/10.1109/TIT.1967.1053964

[21] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984. [Online]. Available: https://doi.org/10.1201/9781315139470

[22] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual*. Scotts Valley, CA: CreateSpace, 2009. [Online]. Available: https://dl.acm.org/doi/abs/10.5555/1593511

[23] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-2649-2

[24] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011. [Online]. Available: https://dl.acm.org/doi/10.5555/1953048.2078195

[25] R. A. Fisher, "Iris," UCI Machine Learning Repository, 1988. [Online]. Available: {DOI}:https://doi.org/10.24432/C56C76

[26] D. Dua and C. Graff, "UCI Machine Learning Repository," 2017. [Online]. Available: http://archive.ics.uci.edu/ml

[27] T. M. Mitchell, *Machine learning*. McGraw-Hill, 1997. [Online]. Available: https://dl.acm.org/doi/10.5555/541177

[28] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. [Online]. Available: doi.org/10.5281/zenodo.592536

[29] V. Sigillito, S. Wing, L. Hutton, and K. Baker, "Ionosphere," UCI Machine Learning Repository, 1989, DOI: https://doi.org/10.24432/C5W01B. [Online]. Available: https://doi.org/10.24432/C5W01B

[30] V. Lohweg, "banknote authentication," UCI Machine Learning Repository, 2013. [Online]. Available: https://doi.org/10.24432/C55P57

[31] T. Kluyver, B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing, "Jupyter notebooks – a publishing format for reproducible computational workflows," in *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, Eds. IOS Press, 2016, pp. 87 – 90.

A Appendix

A.1 Diary

The diary appended below is in rather informal shorthand but illustrates my honest thought process while approaching and delivering the project. The diary can be found in the diary folder of the project repository, titled "FYP-diary.MD"

A.1.1 18th September

Allocated supervisor and informed by supervisor of my project title (allocated on 15th September). Supervisor will be Prof. Zhiyuan Luo. Project will be Comparison of Machine Learning Algorithms.

- Made notes and research on "Breaking the ties" with K-NN algorithms
- Made notes and basic research on Uniformity with Decision Trees, noted measures of uniformity (Gini, Entropy, Info Gain) - will need to read more about these soon. ^^Both points are highly relevant for the early deliverables.

Watching CS229 ML lecture video[1] given by Andrew Ng at Stanford, 2018: Very easy to absorb lecture on overview of different types of ML. My project so far seems to be directed towards Supervised Learning-Classification

A.1.2 19th September

Attended Project Talk given by Dr Argyrios, covered 2 presentations. Useful talk, Argyrios clarified to me personally that LaTeX Project Plan can be in any format, isn't an institution-locked layout. Heavy emphasis on focusing on project plan.

As of writing, the outline of plan is somewhat structured, but much work needs to be done.

Will meet Prof. Zhiyuan on 25th, guidance given on how to access his office

A.1.3 20th September

Started trying to plan my meeting with Prof. Zhiyuan. Many questions to ask, just trying to find the right questions. Noted that there are various paths I can take this project, now considering if I should exclusively focus on Classification. Noted that it's difficult to research on Classification without needing to read into Regression techniques. There's also lots of notation to digest.

- Understood the functionality of parameters (theta) alongside features/inputs (x). Learning a lot on notation in ML. The algorithms given in the brief seem to be more-or-less non-parametric, however - though I could be mistaken.

[1] `https://youtu.be/jGwO_UgTS7I?si=TcRudZ_jwvuylEkv`

- Research on kernel functions and how they transform data into a different representation such that data is easier to evaluate and more separable. I understand this very superficially. Need to read more on examples: Linear, Polynomial, Gaussian (RBF) and Sigmoid Kernels.
- Some basic research on Multi-Class Support Vector Machines("one-vs-all", "one-vs-rest"). Complicated for now, will be useful for final deliverables. Also tribute to Vapnik.
- Research on RSS (Residual Sum of Squares) Criterion - not sure if relevant to KNN/Trees/Classification algorithms, seems moreso relevant to linear regression Started watching CS229 Lecture 2[2], very useful information on notation by Andrew Ng, mostly focused on Regression however.
- Some concerns that I'm mostly focused on theory and not so much on implementation. Started digging around sci-kit learn documentation[3] today: - will continue to get familiar in upcoming days.

Upon arrival at campus, I've borrowed 5 books from library physically: 1. The Elements of Statistical Learning, 2nd Edition. (TEoSL) 2. Pattern Recognition and Machine Learning by Bishop. 3. Foundations of Machine Learning by Mohri 4. Introduction to Machine Learning with Python by Muller & Guido. (MLwP) 5. Hands-on Machine Learning with Scikit-Learn and TensorFlow

Primarily I've been using (1) TEoSL as reference and using the others, or the internet to decipher concepts or notation when it's difficult to at first glance. I've not been using (4) yet but I think it will be vital as I start making proof of concept programs.

Today found out from Prof. Vovk via Moodle that a new textbook has been released online: An Introduction to Statistical Learning with Applications in Python, 2023. It seems to be a simpler version of TEoSL, so it might be the perfect resource for me. I'm in luck! Exchanged emails with the library services today but book isn't available physically, they've provided me an online copy.

GitLab repo is online! Aiming to put this diary on there.

-Research on using OverLeaf (LaTex editor) in sync with GitLab such that I can track version history of my reports. Also research on Git, branches and recapping version control studies. Emailed Prof. Argyrios for confirmation of repo, as well as gitlab classes and slides from earlier talk.

A.1.4 21st September

Made various changes and commits to repo last night and this morning, it now contains folders for this diary and the project plan I'm working on in LaTeX. I've been pushing to master(or main) lately and it doesn't feel right to me - so now that I've made my repo somewhat easy to navigate, I'm making a branch to continue research on: plan

[2] `https://youtu.be/4b4MUYve_U8?si=GvrB1HdXNJ66JWNE`
[3] `https://scikit-learn.org/stable/modules/tree.html`

I'll use the planning branch up until my Project Plan is complete, I can then use my repo to explore my research and put proof of concept programs for the algorithms together while I'm still trying to define my project. This branch will represent my initial development phase. I've spent a lot of time last night and this morning keeping up to date with git conventions such that everything is done orderly. Creating this branch felt like the natural thing to do in this formative phase of my project. My intention is to merge the plan branch with main around when the Project Plan is complete and submitted, marking the beginning of development (and likely, a new dev branch).

I purchased a pocketbook for meetings and ideas with my supervisor

I will spend today looking at sci-kit learn and trying to play with library functions that have implements knn and decision trees already. I have barely looked into datasets yet and I must do so, this will likely define the direction of my project. Still need to look into whether Classification is what I want to exclusively work on. I will also work on my Project Plan, this should be a priority.

A.1.5 22nd September

Yesterday I continued reading about kernel functions and SVMs and honestly - found myself overwhelmed with theory and how much I still need to understand, let alone bring to the meeting on 25th. I think I need to focus more now on implementation. I'll work on the simplest ML algorithms and focus on implementing them and assessing them, further theory can wait I think. I aim to now focus on working through the exercises in MLwP and get familiar with working with datasets. Yesterday I also found Kaggle as a useful resource for datasets. Kaggle is the largest open-source ML community and it's highly likely I'll use their resources in this project, along with UCI and Delve, as long as the data isn't too difficult to process.

Also thinking of turning these diary files into markdown format for easier use.

Setting up python and Jupyter on macOS surprisingly time-consuming.

A.1.6 23rd September

Been playing with GUI elements today, tkinter and pyqt5 - I've not really decided how the end-product of this project is going to look and function, so I started investigating this today. Wasted a ton of time configuring python environments.

Spent good amount of time today planning for meeting, have a decent outline of things to cover. In the process, have structured my approach to tackling the project a little better.

A.1.7 24th September

Wrote meeting outline for supervisor before tomorrow's meeting, during the process did a lot of research on model evaluation concepts and metrics

Sent email with a decent amount of detail, very much looking forward to the meeting. I feel this has been a productive week but I need to make many decisions still on how I'm going to tackle this project.

Will likely change these to .MD files tomorrow and perhaps make a research folder. Hoping to start implementing Jupyter Notebook projects next week.

My priority next week however, will be my project plan.

A.1.8 25th September

Met supervisor! Meeting was thoroughly helpful. Main takeaway from meeting was to 'keep it simple'. I will focus on Nearest Neighbours for now and try implementing the simplest variation of this algorithm in handwritten form. Doing this will allow me to fully understand the algorithm at its core. I'll likely just focus on simple implementations of Nearest Neighbours, Decision Trees and perhaps Logistic Regression - I need to read more into this.

My priority is the Plan right now, I think I know which reports to focus on for this term at least. I should start with mathematical implementations of the algorithms to further my understanding - and this might help while I'm creating the Plan. I should then get started on building the algorithm and can structure my approach in SE terms via the Plan too. It's clearer to me now that the Plan is a tool to help me, rather than just a deliverable for assessment.

A.1.9 27th September

Attended FYP Talk on LaTex and Referencing today. Since classes have started, it's gotten a lot busier and less time to focus entirely on research and project progress.

Additions have been made to Project Plan abstract. I hope to have an outline of some kind prepared by Friday and ideally a draft before the end of the week which can be reviewed by my supervisor.

I've forgotten to mention that I've been using a 6th physical book by Tom Mitchell called Machine Learning for much of my research. I find it easier to absorb theory on there, and it covers decision tree theory in much more detail than TEoSL. I think it's probably much more suited for Classification problems.

A.1.10 28th September

Following the previous FYP talk, I've started using Google Scholar to find papers to use as references - and already found some great authoritative texts on the NN algorithm. I aim to make real headway on the Plan today and I really hope to complete some kind of draft by tomorrow, such that I can get early feedback from my supervisor. Today I really need to make some executive decisions on the direction and goals of my project. I've decided it may be wise to include some kind of *success criteria* in my Abstract, such that I can make critical goals to deliver for the interim review. This project is so open to expansions that it's important to define what should be expected by the end of

this. Creating a set of possible expansions may make it more flexible. Once I start assigning a time-frame to this project, it'll become clearer.

A.1.11 15th October

It's been a while since the last diary entry. Unfortunately, I may have underestimated how challenging it would be to balance the project with the other modules I'll be studying. The week following my last entry was entirely focused on implementing my Project Plan and submitting something that motivated and outlined my approach to the project sufficiently. The week after this should have been focused on developing my proof of concept programs, designing my UMLs and beginning my Nearest Neighbours report. Unfortunately, last week was not fruitful and I'll have to spend this week catching up with my outline immediately.

A.1.12 16th October

I've made small progress with the NN algorithm proof of concept and it works as expected. The labs in my machine learning module have been very useful with learning python syntactic sugar and dealing with datasets in Jupyter Notebooks. I'd like to somehow implement some kind of unit testing this week that I can carry forwards for the rest of term. The repo needs to progress to a new branch for development and I hope to do this immediately after this entry. I'll be transferring from the 'plan' branch to a 'dev-poc' branch, updating the main branch in the process. It's likely that this will be the first of many dev branches in the overall process. I'll need to update my README too with all the new structure I have put in place. I'll likely setup some structuring of my interim and supplementary reports(NN) this week. I'll move forwards with haste, and hopefully hear feedback on my outline soon, just to find if it's too ambitious, though I feel I'm already seeing that it is.

A.1.13 2nd November

With the start of November, I do feel that I'm falling behind, not only with the work outlined in my plan - but also unfortunately, the diary entries. I do hope to make these more frequent and get back on track with my weekly diary entries.

I've made progress on Nearest Neighbours and K-Nearest Neighbours proof of concept programs in Jupyter Notebooks. I realised quite quickly that it was imperative to create train-test-split functionality immediately just to test these algorithms functionally, and I've managed to do so. I think that my implementation of train-test split will help in implementing k-folds cross-validation. I've essentially realised that I need to implement model evaluation functions in conjunction with my models otherwise it's difficult to know if I'm going the right direction with my model implementation.

If I focus on completing an implementation Nearest Neighbours and beginning an implementation with the Tree algorithm - which I believe will carry

its own challenges - before the end of this week, as well as implementing cross-validation in some way. I may be able to catch up with my plan's outline.

I'm moreso worried about making progress on my reports, as I'm very behind on this and I believe this will be more time-consuming than the algorithm's implementation. I'll be making immediate work on the NN algorithm report, and have already built the skeleton of the report in LaTeX. My only consolation with this is that I don't feel lost with the theory of this much at all and I think research should be straightforward thanks to the reading I did out of interest during the summer.

I had my second supervisor meeting a week ago, and the main theme of this meeting was that I essentially need to simply put my head down and deliver. My understanding is there, I simply need to push work forwards without fear of making errors.

Areas that I might need to look into next week include handling missing data in datasets and how to implement PyUnit tests for my validation functions. I've played with a heart disease dataset from the UCI repository and noticed that handling missing values might be a challenge that I need to focus on within my data preprocessing phase.

CS3920 lectures and labs have been handy for me looking ahead, and I think investigating normalisation techniques and how they affect model accuracy may be something to do next term.

All in all, I'm running behind, I'm aware of it, and I need to move quickly to catch up in time for the interim review.

A.1.14 15th November

I attended the FYP talk today by Prof. Dave Cohen about Presentations and how to bring forward our project during Presentation Week. I'm actually looking forward to talk about the research I've made. However I believe I'll need to make some compromises in order to deliver my targets on time, and I think this will have to take the form of the report deadlines set by myself. I've simply not been keeping up with working on reports in adjacent form with my development, along with the intense assignments I'm working on currently this month. I think I'll have to work solely on the interim report and put together my findings on the two algorithms concurrently within my interim report - rather than simply bringing in two completed algorithm reports to introduce within the interim report. Once the interim report submission is complete, I can review if I want to add more detail or background to the algorithm reports during the second term. It's likely that I'll need to perform some kind of review before the end of the the year to create a more thorough plan for Term 2.

I've made some progress with the Decision Tree with Gini Impurity but still trying to work out how to introduce stopping criteria. I'm tentative to commit something that breaks. From the library, I've picked up the book by Leo Breiman - Classification and Regression Trees. It's verbose but goes into pretty deep detail about tree splitting, stopping and pruning strategies. I'm curious about introducing entropy/information gain but the benefits don't seem

obvious to me unless I bring about categorical data into the mix - which I've just not done yet.

I've discovered a really neat dataset on Kaggle called the Titanic dataset, it seems like a fun thing to bring in and a little more interesting than classifying flowers. I'd like to try and bring this into play before the end of term, but it depends on the difficulty of data preprocessing. I think it'd be nice to talk about for my presentation.

Looking back at my early research, it's quite funny to see how much of the theory I was reading through is rather irrelevant to the implementation I'm putting together - (RBF Kernels, Multi-class SVMs) - and it makes sense now why my supervisor told me to keep things simple back during our first meeting.

A.1.15 24th November

Judgement day is almost upon us as my Interim Review deadline is approaching. I believe I have two functional algorithms deemed worthy for evaluation, though both could be extended in various ways. I now need to piece together my report and ensure I have sufficient notebooks that evaluate the algorithms' performance. I've not quite completed the cross-validation functionality, but I think I can have this complete in a few days. My third supervisor meeting has been scheduled for the 30th. I need to ensure I have a testing strategy in place that checks the robustness of the algorithms while I made alterations to them.

I do feel that I could bring in a third algorithm into play during second term, to make things more interesting. I think after learning about SVMs in CS3920, I have an idea on how this could be a third classification algorithm I could use for comparison to KNN and Trees.

A.1.16 27th November

Since my last entry I've managed to implement K-Folds Cross-Validation, and made a few tweaks to my models such as adding a get*depth() function to my decision tree. I think my the end of today I can have Leave-One-Out Cross-Validation implemented (as it's essentially when K-Folds = N) with its own get*score functions. Implementing K-F CV feels significant as it seems to be the first time that all of these modules are working together, and seeing it work in the notebook has been satisfying, confirming that the data structures are all working as I'd hoped. One thing I'm a little disappointed about is how my algorithms don't yet work on datasets with missing data, or with categorical data - which significantly limits which datasets my models can work on. I have to make a decision on whether I can somehow implement this soon, or just focus on delivering my report and interim review.

I've managed to implement a more explicit way of handling ties within my KNN algorithm today.

I feel that I could find a way to automate some kind of "hyperparameter tuning" for my algorithms - I'm just currently unsure if this would lead to data snooping. From what I'm reading - it's important to use one hold-out set for

hyperparameter tuning, and then a different test set for general performance. I'll need to clarify this with my supervisor.

As for categorical data, I've been reading on how ordinal data and nominal data are generally handled - using one-hot encoding and label encoding. It seems to me that I may need to build some kind of encoder for my data in the future to handle future datasets. I've not considered normalisation - and it'll be necessary for me to implement this, particularly scaling, as if I want to start training on the breast cancer dataset and titanic data - which has missing data, but also a wide range of scales of data - I'll need to bring this forward soon.

On another note, today the deadline for the interim review has been postponed by a week. This is generally good news, but does make me a little confused about whether I should keeping implementing new changes - or wrap things up for review and focus on my report and presentation.

A.1.17 28th November

Having read more about normalisation, handling categorical data and missing data - I think it would be wise to delay this until after the review. I could try and quickly hash something together that works, but I do feel that I need to thoroughly research and plan this implementation out in such a way that they're integrated with model models smoothly.

For KNN, there's a chance that I may need to implement a new distance measure such as Hamming Distance or Jaccard Distance to handle categorical data sufficiently, without making misleading effects to the distance between samples.

A.1.18 29th November

I've managed to add various test suites to thoroughly check edge cases for the functionality implemented thus far. Doing this helped me notice gaps in my exception handling when passing values around - such as in K-folds and train-test-split when specifying the size of the fold or split.

A.1.19 30th November

Today I finally had my third meeting with my supervisor. It went smoothly and my supervisor seems satisfied with my progress thus far. I managed to clarify doubts I had about the following:

- Handling missing and categorical data:
  - For missing feature data, it's usually find to remove the entire sample for that instance. As long as there's enough data samples to make training sufficient.
  - I asked about implementing an encoder to handle nominal and ordinal data, this seemed like a suitable idea.

  - I asked about how this may affect the distance function in KNN, and if nominal data could be handled correctly, if I should using Hamming, Jaccard or Cosine function as an alternative. He suggested I keep trying with Euclidean as this should work okay with the situations I'm dealing with. He reminded me to just try it before I doubt the implementation.

- He cleared up misunderstandings I had about other performance metrics besides accuracy - such as the difference between Precision and Recall, and using the F1 score. I think I could quickly calculate these with the implementation I have so far. He also reminded me that I can find the variance of my model by analysing the range of accuracy values I find during K-Folds Cross-Validation.

- Hyperparameter Tuning - I asked about the procedure of finding the ideal hyperparameters of KNN and Trees, i.e. the number of neighbours K for KNN and the maximum depth of the tree constructed. He reminded me to avoid data snooping, which I seemed to be doing in Notebook 6. I need to keep the test set separated and utilise a validation set for this.

- He emphasised that it's very important that there's some form of data normalisation for KNN, as it's rather distance-sensitive. This may have explained unusually low values I was getting for the optimum k-value on large datasets such as ionosphere (though perhaps this is the curse of dimensionality in effect).

Following the meeting, I immediately implemented a MinMaxScaler so that I have some form of data normalisation I can use to handle misaligned data ranges, particularly for KNN. I will need to correct my hyperparameter tuning procedure too.

Overall in project development, I have built a pretty strong foundation for me to use for Term 2. I just need to show the results found on the three datasets I've chosen - iris, ionosphere and banknote authentication. Right now, my priority is putting the presentation and report together - this needs more urgent work.

A.1.20 6th December

Obligatory diary entry - I've been working away at completing my deliverables for the interim review - the interim report and the presentation. The presentation has been submitted and will take place on the 8th. I'm finding it difficult to not add further functionality to the project last minute. I think there's still a lot I can do to automate various processes, such as the hyperparameter tuning process, and finding metrics such as accuracy. I think I can work on the preprocessing.py module this month after the review, and also work on integrating the process of calculating accuracy, recall and precision within a new metrics.py module. My report is coming together, it leans quite heavily into the theory.

Overall, I do feel quite proud of what I've managed to do this term, and I think I can carry this momentum forwards, once submission is completed.