CSF and Plasma Biomarkers as Predictors of the Progression of Mild Cognitive Impairment to Alzheimer's Disease

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER FOR THE DEGREE OF BACHELOR OF SCIENCE IN THE FACULTY OF SCIENCE AND ENGINEERING

Rao Ali Hasan

Supervised by Dr. Fumie Costen

Department of Computer Science, The University of Manchester

May 2025

Abstract

With an estimated annual conversion rate of 10–15%, Mild Cognitive Impairment (MCI) is considered a prodromal stage of Alzheimer's Disease (AD). Early identification of individuals at risk of progression is vital for timely intervention to delay AD onset. However, existing diagnostic tools such as imaging biomarkers like Positron Emission Tomography (PET) and Magnetic Resonance Imaging (MRI) scans remain expensive and inaccessible for many individuals, particularly in third-world countries and under-developed regions. This research explores the potential of Cerebrospinal Fluid (CSF) and plasma biomarkers as economical predictors of MCI-to-AD conversion over a two year period. Using a ensemble model, this research integrates diverse inputs, including demographic, genetic, biomarker, cognitive, and functional assessment data, to identify MCI participants who develop AD within 2 years from the initial diagnosis. The project further evaluates various permutations of these predictors to assess the robustness of the model against different limited datasets. The final proposed model demonstrated strong predictive performance on the test set, achieving an accuracy of 0.82 ± 0.02, a precision of 0.70 ± 0.04, an F1-score of 0.55 ± 0.06, and an AUC of 0.88 ± 0.02 across a stratified 4-fold cross-validation.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

I. The author of this thesis (including any appendices and/or schedules to this thesis) owns certain copyright or related rights in it (the "Copyright") and s/he has given The University of Manchester certain rights to use such Copyright, including for administrative purposes.

II. Copies of this thesis, either in full or in extracts and whether in hard or electronic copy, may be made **only** in accordance with the Copyright, Designs and Patents Act 1988 (as amended) and regulations issued under it or, where appropriate, in accordance with licensing agreements which the University has from time to time. This page must form part of any such copies made.

III. The ownership of certain Copyright, patents, designs, trade marks and other intellectual property (the "Intellectual Property") and any reproductions of copyright works in the thesis, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property and/or Reproductions.

IV. Further information on the conditions under which disclosure, publication and commercialisation of this thesis, the Copyright and any Intellectual Property and/or Reproductions described in it may take place is available in the University IP Policy (see `http://documents.manchester.ac.uk/DocuInfo.aspx?DocID=24420`), in any relevant Thesis restriction declarations deposited in the University Library, The University Library's regulations (see `http://www.library.manchester.ac.uk/about/regulations`) and in The University's policy on presentation of Theses.

Acknowledgments

I wish to express my gratitude to Dr. Fumie Costen for her unwavering support, encouragement, and invaluable guidance throughout the entirety of this research project. Her mentorship has been crucial in shaping the direction and quality of this work, and it has been an privilege to learn and collaborate under her supervision. I am also very thankful to my family and friends, especially my parents, for their steadfast support and encouragement throughout this endeavour.

Lastly, I extend a heartfelt thanks to my grandfather, Rana Atta Muhammad, whose influence and guidance have left an indelible mark on my academic and personal journey, and this work is dedicated to honouring his profound impact.

List of Tables

2.1 Evaluation metrics for classification tasks. 24
2.2 Confusion Matrix. 25

5.1 Abbreviations of some of the input features in the dataset. 34
5.2 Participant overview at the two-year follow up from the initial visit after data imputation. 35

6.1 Hyperparameter values tested for the K-Nearest Neighbours model 48
6.2 Hyperparameter values tested for the Logistic Regression model . 50
6.3 Hyperparameter values tested for the Support Vector Machine model 51
6.4 Hyperparameter values tested for the Multilayer Perceptron model 52
6.5 Hyperparameter values tested for the Random Forest model . . . 53
6.6 Hyperparameter values tested for the Decision Tree model 55
6.7 The classification accuracy of all ensemble model combinations and classification techniques 58

7.1 The different datasets used for ensemble model evaluation 63
7.2 Mean and standard deviation of each metric in relation to the performance of the ensemble model on each dataset 64

List of Figures

2.1 Machine Learning Pipeline. 18
2.2 Linear SVM model representing the classification of two classes [18]. 20
2.3 The MLP topology [20]. 21
2.4 The Random Forest topology [23]. 22

3.1 FE Model, Feature Engineered Model; DA, Demographics and APOE4; P, Plasma; C, CSF; H, Adjusted Hippocampal Volume [24]. 27
3.2 The quantity of missing data visualised for 928 participants with MCI [27]. 28

5.1 Pie charts of the categorical features in the dataset. 36
5.2 Percentage of missing data per feature in the dataset. 37
5.3 The Mean Squared Error and Mean Absolute Error for varying K-values in KNN imputation. 38
5.4 The Mean Squared Error and Mean Absolute Error across different iteration and dataset split configurations for MICE Forest imputation. 39
5.5 Histogram and statistics of the features before imputation which have more than 30% data missing. 40
5.6 Histogram and statistics of the features after imputation which initially had more than 30% data missing. 41
5.7 Example of random undersampling. 42
5.8 Example of random oversampling [39]. 43
5.9 Visualisation of the different oversampling methods in a 2-dimensional space. 46

6.1 Mean classification accuracy across folds for different values of K and weighting schemes. 49
6.2 Mean classification accuracy across folds for different values of C and optimisation methods. 50
6.3 Mean classification accuracy across folds for different values of C and kernel functions. 51

6.4 Mean classification accuracy across folds for different activation functions and hidden layer sizes. 53
6.5 Mean classification accuracy across folds for different number of decision trees and maximum tree depth values. 54
6.6 Mean classification accuracy across folds for various hyperparameter combinations. 55
6.7 Heatmap of mean classification accuracies across folds for different combinations of maximum depth and criterion. 56
6.8 ROC curve for each of the models with their optimal hyperparameter configuration. 57
6.9 A soft voting ensemble model built using all 6 models. 58
6.10 A stacking ensemble model built using all 6 models. 61
6.11 The structure of the final ensemble model. 61
6.12 The confusion matrix of the best performing ensemble model. . . 62
6.13 The ROC curve of the best performing ensemble model. 62

7.1 The ROC curve of the ensemble model when tested on the full dataset. 65
7.2 The confusion matrix for the prediction of MCI to AD conversion by the ensemble model on the full dataset. 65
7.3 Mutual information-based feature importance analysis of the ensemble model on the full dataset. 67
7.4 Mean permutation importance score of each feature in the dataset. 68

Contents

1 Introduction **11**
1.1 Motivation 11
1.2 Aims and Objectives 11
1.2.1 Aims 11
1.2.2 Objectives 11
1.3 Report Structure 12

2 Background **14**
2.1 Alzheimer's Disease (AD) 14
2.1.1 Causes 14
2.1.2 Stages of Development 14
2.1.3 MCI and AD in ADNI 16
2.1.4 Diagnosis 16
2.2 Machine Learning 17
2.2.1 The Fundamentals of Machine Learning 17
2.2.2 Machine Learning Pipeline 17
2.2.3 Hyperparameter Tuning 22
2.3 K-Fold Cross Validation 25

3 Literature Review **26**
3.1 Model Input 26
3.1.1 Missing Data 27
3.1.2 Limitations 28
3.2 Classification Methods 29
3.2.1 Random Forest 29
3.2.2 Logistic Regression 29
3.2.3 Support Vector Machine 29

4 Experimental Methods **30**
4.1 Methods 30
4.2 Data 30
4.3 Model 31
4.4 Considerations 32

5 Preprocessing **33**
5.1 Data Exploration 33
5.2 Missing Data 37
5.3 Class Imbalance 42
5.3.1 Undersampling 42
5.3.2 Oversampling 43
5.3.3 Results 45
5.4 Further Preprocessing 46
5.5 Feature Selection 47

6 Model Construction **48**
6.1 Developing K-Nearest Neighbour Model 48
6.1.1 Hyperparameters 48
6.1.2 Results 49
6.2 Developing Logistic Regression Model 49
6.2.1 Hyperparameters 49
6.2.2 Results 50
6.3 Developing Support Vector Machine Model 51
6.3.1 Hyperparameters 51
6.3.2 Results 52
6.4 Developing Multilayer Perceptron Model 52
6.4.1 Hyperparameters 52
6.4.2 Results 52
6.5 Developing Random Forest Model 53
6.5.1 Hyperparameters 53
6.5.2 Results 54
6.6 Developing Decision Tree Model 55
6.6.1 Hyperparameters 55
6.6.2 Results 56
6.7 Developing Ensemble Model 57
6.8 Model Construction Results 60

7 Results and Discussions **63**
7.1 Model Evaluation Results 63
7.1.1 Full Dataset 64
7.1.2 Base + CSF Dataset 64
7.1.3 Misclassification 64
7.2 Feature Importance Analysis 66
7.2.1 Mutual Information-Based Feature Importance 66
7.2.2 Permutation Feature Importance 67
7.3 Discussion 68
7.3.1 Discussion of Results 68
7.3.2 Libraries Used For Implementation 69
7.3.3 Evaluation of Objectives 69
7.3.4 Reflection 70

8 Conclusion and Future Work **71**
8.1 Conclusion 71
8.2 Future Work 72

A Background **80**
A.1 Treatment for Alzheimer's Disease 80

B Source Code **81**
B.1 Common Utility Functions 81
B.2 Data Imputation Classes 87
B.3 Model Training & Evaluation Utility Functions 88
B.4 Plasma Data Collection 97
B.5 Data Preprocessing 100
B.6 Model Training & Hyperparameter Optimisation 110
B.7 Model Evaluation 115

Chapter 1

Introduction

1.1 Motivation

Mild Cognitive Impairment (MCI) is characterised by an excessive memory or cognitive decline compared to those of similar age placing such individuals at a significantly elevated risk of progressing to Alzheimer's Disease (AD). Early detection of MCI cases that will potentially develop into AD is crucial to ensure timely preventive measures and monitor disease progression. Current diagnostic methodologies primarily rely on advanced brain imaging techniques such as Magnetic Resonance Imaging (MRI) and Positron Emission Tomography (PET) scans [1]. Although these tools have high accuracy, they are prohibitively expensive and resource-intensive, limiting their accessibility in settings with less resources such as third-world countries. The purpose of this research is to investigate the potential of plasma biomarkers in conjunction with Cerebrospinal Fluid (CSF) as a more cost-effective and scalable alternative for predicting MCI-to-AD progression over a two-year period.

1.2 Aims and Objectives

1.2.1 Aims

This research aims to develop a robust Machine Learning (ML) classification model to predict the progression of MCI to AD or the likelihood of remaining in a stable MCI state over a two-year period using data sourced from the Alzheimer's Disease Neuroimaging Initiative (ADNI) repository.

1.2.2 Objectives

The following objectives are in line with the development of an ML project:

1. **Data Collection and Preprocessing:** Retrieve the relevant datasets from the ADNI repository. Collate and preprocess this data by dealing with any missing values, extracting pertinent features, filtering the dataset to focus on the two-year time window, and ensuring the data is properly formatted and ready for subsequent analysis.

2. **Model Implementation and Training:** Evaluate the performance of various classification algorithms on the preprocessed dataset. Conduct hyperparameter tuning to optimise each model's configuration, maximising the predictive performance on the validation set. An ensemble model will be constructed by integrating the best performing models to create a robust predictor that generalises effectively to unseen data.

3. **Ensemble Model Evaluation:** Assess the ensemble model's performance across different test datasets, each incorporating varied feature permutations. Identify the impact of different neuropsychological and cognitive assessments, as well as Cerebrospinal Fluid (CSF) and plasma biomarkers on the predictive accuracy to determine which combination of features provide the most accurate predictions of disease progression.

4. **Input Feature Evaluation:** Investigate the relative importance of each feature in the dataset with respect to the model's predictive capability. Rank the features according to their significance in influencing the model's output, thus providing insight into the most critical data for predicting the transition from MCI to AD.

1.3 Report Structure

This report is structured into chapters, each formulated to explore the various stages of design, development and evaluation of the project. The content of each chapter is as follows:

Chapter 2 - Background

This chapter provides an in-depth overview of AD, describing its progression through the various stages, it's underlying causes and diagnostic methods. The chapter then delves into the role of Machine Learning (ML) in this context, explaining the techniques and models employed in this study. Key aspects of the ML process such as model selection, hyperparameter optimisation and evaluation strategies are discussed to outline the methodological framework of the project.

Chapter 3 - Literature Review

This chapter analyses prior research on the prediction of MCI-to-AD conversion, focusing on the ML methodologies and their application. The discussion describes the range of techniques employed for model development and evaluation, alongside a review of the features leveraged as input variables for the model.

Chapter 4 - Experimental Methods

A detailed account of the datasets aggregated to form a unified dataset for the model, as well as outlining the software environment, computational setup, and libraries utilised for development.

Chapter 5 - Preprocessing

The preprocessing pipeline applied to the collated data, including class-specific filtering, temporal filtering to align with the two-year study window, and data imputation for handling missing values. Techniques such as oversampling and dataset augmentation are explained, along with the steps taken for data cleaning and preparation to optimise the model construction phase.

Chapter 6 - Model Construction

A description of the hyperparameter tuning process for each ML classifier is presented, alongside a performance analysis of each individual algorithm. The construction of the ensemble model is outlined, highlighting the methodology and the rationale behind it.

Chapter 7 - Results and Discussions

An analysis of the performance of the final ensemble model across the different datasets is presented. It examines the importance of each feature in determining the output class, offering insight into each features' predictive capability. A discussion of the results is included, highlighting their implications and clinical utility.

Chapter 8 - Conclusion and Future Work

A summary of the project's key findings, addressing the limitations and challenges encountered during this research. Potential approaches for future work are proposed, aimed at addressing these limitations and expanding the study's relevance and usability.

Chapter 2

Background

2.1 Alzheimer's Disease (AD)

Alzheimer's Disease (AD) is a progressive neurodegenerative disorder caused by damage to brain neurons which slowly destroys memory and thinking skills. It is the most common type of dementia, characterised by memory loss and an overall decline in cognitive and functional capabilities. An estimated 6.7 million Americans aged 65 years and older currently live with this disease, with this number projected to rise to 13.8 million by 2060 barring the breakthrough in medical treatment [2].

2.1.1 Causes

A crucial feature of AD pathology is an abnormal production and clearance of amyloid-β (Aβ) peptides, particularly Aβ42, leading to the deposition of amyloid plaques in the brain. This occurs alongside the accumulation of hyperphosphorylated tau (p-tau), a marker of progressive neurofibrillary pathology [3, 4]. Although the main causes of these pathological changes remain elusive, it is known that Aβ and p-tau collectively disrupts neuronal integrity which results in cognitive and functional decline. Key risk factors for AD include age, genetics, and conditions such as cerebrovascular disease, diabetes, hypertension, and obesity [5].

2.1.2 Stages of Development

Stage 1: Subjective Cognitive Decline (SCD)

Subjective Cognitive Decline (SCD) involves a self-perceived decline in cognitive function that is not corroborated by external observation. A key criterion is that individuals with SCD perform within the normal range on the standardised cognitive tests typically used for classifying MCI [6]. Despite the absence of measurable impairment, individuals with SCD often report symptoms similar to those seen in MCI, including increased forgetfulness, difficulty with decision-making and planning, and depression or anxiety. However, these symptoms do not significantly disrupt daily activities which distinguishes SCD from more advanced stages of cognitive impairment.

Stage 2: Mild Cognitive Impairment (MCI)

Mild Cognitive Impairment (MCI) is a clinical condition characterised by a cognitive or memory impairment that is greater than expected for an individual's age and education level, but not severe enough to significantly interfere with daily tasks. It is regarded to be a potential prodromal stage of Alzheimer's Disease (AD), representing a critical window for early intervention and monitoring [7]. Key indicators of this decline could include frequent misplacement of personal belongings, recurrent forgetfulness such as missing appointments or social events, and difficulty in remembering words during conversations.

Stage 3: Alzheimer's Disease (AD)

Alzheimer's Disease (AD) represents the final stage of neurodegenerative decline and is the leading cause of dementia. Diagnosis is primarily based on neuropsychological assessments and clinical symptoms such as progressive memory impairment, aphasia, apraxia, and agnosia. However, definitive confirmation can only be achieved through a biopsy or autopsy [8]. Early signs of Alzheimer's Disease often begin with mild memory deficits but progressively worsen to include language difficulties, personality changes, impaired mobility, depression and anxiety. The progression of Alzheimer's Disease follows a series of the following stages [8]:

1. **Preclinical Stage:** This phase can span several years and is characterised by mild memory loss and early pathological changes, particularly in the hippocampus and cortical regions.

2. **Mild AD:** At this stage, cognitive impairments becomes noticeable as it begins to affect daily routines and leads to mood changes, including increased anxiety and depression.

3. **Moderate AD:** Disease progression extends to the cerebral cortex, worsening memory loss and impairing impulse control, language comprehension, reading, writing, and speech.

4. **Severe AD:** The disease spreads to the entire cortex with large accumulation of neuritic plaques and neurofibrillary tangles, with patients often becoming bedridden, losing the ability to swallow and control bladder function, and failing to recognise close family members. The disease ultimately leads to death.

2.1.3 MCI and AD in ADNI

The data for this research was obtained from the Alzheimer's Disease Neuroimaging Initiative (ADNI), a longitudinal study providing quantitative metrics for classifying individuals with MCI or AD. Participants classified as MCI met a strict set of diagnostic criteria to ensure consistency and reliability. The criteria includes a Mini-Mental State Examination (MMSE) score between 24 and 30 (inclusive), subjective memory complaints confirmed by objective memory impairment, measured using education-adjusted scores on the Wechsler Memory Scale Logical Memory II, a Clinical Dementia Rating (CDR) of 0.5, no significant deficits in other cognitive domains, and preserving sufficient functional performance such that a diagnosis of dementia cannot be made.

In contrast, participants meeting the criteria for a diagnosis of dementia exhibited more pronounced cognitive decline. This includes a MMSE scores ranging from 20 to 26 (inclusive), subjective memory complaints, objective memory impairment on the Wechsler Memory Scale Logical Memory II, and a CDR of 0.5 or 1.0. These diagnostic thresholds established by ADNI were designed to ensure both clinical relevance and reproducibility across studies [9].

2.1.4 Diagnosis

Clinical Information

The first step in diagnosing AD is a comprehensive clinical evaluation to assess cognitive impairment and determine its severity. Standardised neuropsychological tests, such as the Mini-Mental State Examination (MMSE), Alzheimer's Disease Assessment Scale-Cognitive (ADAS-Cog), and Rey Auditory Verbal Learning Test (RAVLT) are commonly used. These assessments are conducted by trained clinicians and are designed to evaluate a wide range of cognitive functions including memory, attention, language and problem-solving abilities. The results of these tests when combined with the clinical history and observational data assist in determining the diagnosis, providing valuable insight into the most affected cognitive domains.

Biomarkers

To confirm the clinical diagnosis, biomarker analysis is crucial. This can include neuroimaging and fluid biomarkers, both offering objective evidence of pathological changes. MRI is used to identify structural brain changes, such as atrophy in the hippocampus, entorhinal cortex, and temporal lobe, along with ventricular enlargement, all of which correlate with cognitive decline [10]. PET imaging allows for the investigation of functional changes, such as a change in cerebral glucose metabolism, neuroinflammation, and disruptions in neurotransmitter systems. The ability of PET to detect amyloid plaques is particularly relevant as it is one of the hallmark features of AD pathology [11].

CSF biomarkers, including amyloid-beta 42 (Aβ42), total tau (t-tau), and phosphorylated tau (p-tau), are important in supporting the diagnosis of AD. These biomarkers have demonstrated high diagnostic sensitivity and specificity to detect the disease, particularly when used in combination [12]. Furthermore, blood-based biomarkers such as Neurofilament Light Chain (NfL), p-tau, Glial Fibrillary Acidic Protein (GFAP) and Aβ42 have recently gained attention for their potential role in AD diagnosis. While blood biomarkers are primarily used for screening purposes currently, their usefulness in early detection, monitoring disease progression, and cost-effectiveness is becoming more recognised [13].

2.2 Machine Learning

2.2.1 The Fundamentals of Machine Learning

Machine Learning (ML) is a subset of artificial intelligence (AI) that encompasses a diverse range of algorithms designed to derive predictive insights from data. Recent advancements in ML have significantly enhanced the ability to extract meaningful information and detect complex, abstract patterns within datasets, often surpassing human-level performance in tasks that require semantic understanding and data-driven inference [14]. ML models can generally be classified into one of the four categories [15]:

1. **Supervised Learning:** This approach utilises a labelled dataset to train algorithms for classification or predictive tasks. The model iteratively adjusts its parameters to minimise error and optimise its fit to the input data.

2. **Unsupervised Learning:** The algorithm is applied to an unlabelled dataset, with the goal of identifying underlying structures and patterns without prior defined classes/values.

3. **Semi-supervised Learning:** A hybrid approach leveraging both labelled and unlabelled data. The labelled subset guides feature extraction and classification to improve the model's ability to generalise from the limited labelled examples.

4. **Reinforcement Learning:** A reward-based framework wherein an agent interacts with an environment, takes actions, and receives feedback in the form of positive or negative rewards. Over time, the algorithm learns optimal decision-making strategies through trial and error rather than relying on a predefined dataset.

2.2.2 Machine Learning Pipeline

The ML pipeline is comprised of several key stages: data collection and preprocessing, feature engineering, model selection and training, hyperparameter

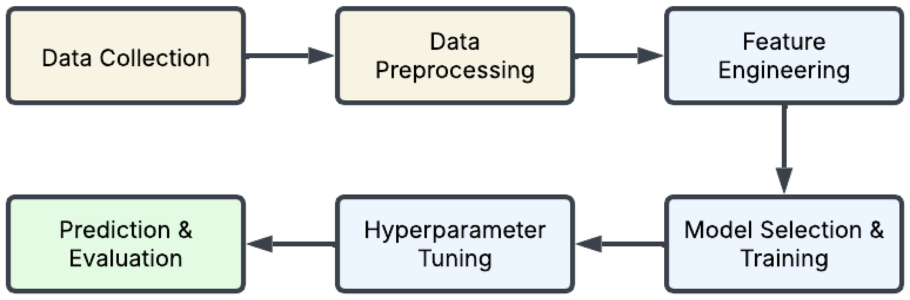


Figure 2.1: Machine Learning Pipeline.

optimisation, and model evaluation. A visual representation of this pipeline is provided in Figure 2.1.

Data Collection

The ML pipeline begins with data collection, a foundational stage that majorly influences model performance. The quality and quantity of the dataset directly impacts the learned parameters, shaping the model's predictive accuracy and generalisability. Data is often aggregated from multiple independent sources, encompassing a diverse range of both numerical and categorical features. Ensuring data integrity, consistency, and representativeness is vital as biases or inconsistencies can propagate through the pipeline which may result in sub-optimal model outcomes.

Data Preprocessing and Feature Engineering

Data preprocessing is an important stage in ensuring data quality and integrating heterogeneous sources into a unified dataset to serve as an input for the modelling phase. Key preprocessing steps include handling missing values either through deletion or imputation to minimise information loss and maintain dataset integrity [16], encoding categorical variables to a numerical representation, and outlier detection and removal to mitigate the influence of anomalous data points. Additionally, data transformation involves standardising mixed data types and normalising numerical features into a defined range to improve comparability. Feature engineering involves selecting, extracting, and transforming relevant attributes from raw data to build a robust and efficient model. While these represent key preprocessing and feature engineering techniques, additional methods can be employed based on the dataset's characteristics and the specific requirements of the task.

Model Selection and Training

Model selection involves evaluating multiple models, comparing their performance, and selecting the most optimal one. It begins with identifying the suitable algorithms based on the problem type, such as classification, regression, time-series analysis, or clustering. In the context of this research, the task is a binary classification problem of determining whether an individual with MCI remains stable or progresses to AD. Given the variety of classification algorithms, we assessed the performance of the widely used models and developed an ensemble model where we incorporated the highest performing ones. The structure of the selected classification models are explained below:

1. **K-Nearest Neighbours**
   K-Nearest Neighbours (KNN) is a simple non-parametric machine learning algorithm used for classification where unlabelled data points are assigned to the class of the most similar labelled instances. The algorithm works by computing the pairwise distance, typically using Euclidean or Manhattan distance, between the unlabelled sample and all instances in the training set. It then identifies the K nearest neighbours and assigns the data point being predicted to the mode class among them. The selection of the K parameter can have a significant impact on the output, where the key is to balance overfitting and underfitting [17].

2. **Logistic Regression**
   Logistic Regression (LR) is a statistical method used for binary classification for modelling the relationship between a dependent variable and one or more independent variables. It operates by defining a decision boundary, also known as a hyperplane, which in a 2D feature space corresponds to a straight line. The decision boundary is computed as:

   $$z = w_0 + w_1x_1 + w_2x_2 + \cdots + w_nx_n \tag{2.1}$$

   where $w_0, w_1, \ldots, w_n$ represent the model's learned weights, $x_1, x_2, \ldots, x_n$ are the input features, and w_0 is the bias term. The predicted probability $\hat{y}$ is computed by applying the sigmoid activation function to constrain the output to the range [0,1]:

   $$\sigma(z) = \frac{1}{1+e^{-z}} \tag{2.2}$$

   Classification is performed by thresholding $\sigma(z)$, where $\sigma(z) \geq 0.5$ is classified as class 1, and $\sigma(z) < 0.5$ as class 0. It uses log loss (binary cross-entropy) to quantify the discrepancy between the predicted probabilities and true labels:

   $$L_{loss} = -\frac{1}{m}\sum_{i=1}^{m}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)] \tag{2.3}$$

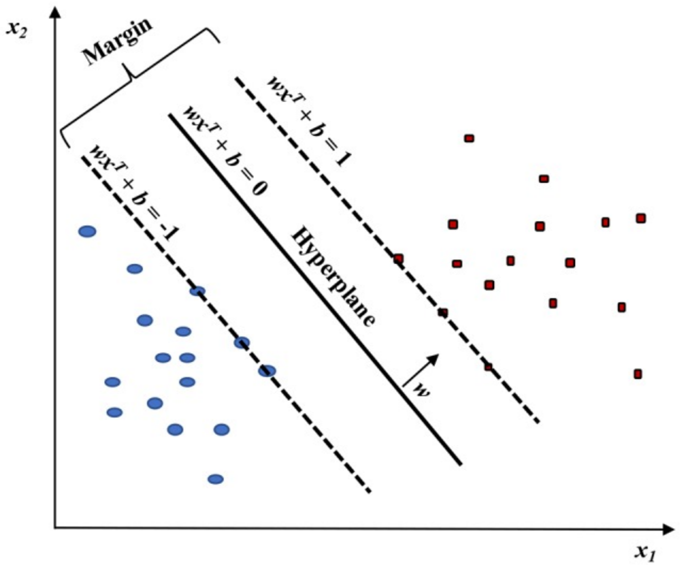


Figure 2.2: Linear SVM model representing the classification of two classes [18].

where y_i denotes the true class label, $\hat{y}_i$ is the predicted probability, and m represents the number of training samples. The loss function increases classification accuracy by ensuring the model optimises its parameters to minimise the divergence between the predicted and actual class probabilities.

3. **Support Vector Machine**
Support Vector Machine (SVM) constructs a decision boundary, also known as the hyperplane, to optimally separate data points into distinct classes. The hyperplane, as shown in Figure 2.2, is positioned to maximise the margin between the closest data points of each class, referred to as support vectors. The hyperplane is defined as $wx^T + b = 0$, where w represents the weight vector, x is the input feature vector, and b is the bias term. The primary objective during training is to determine the optimal values of w and b that maximise the margin given as $\frac{1}{||w||^2}$, hence enhancing the model's generalisability [18].

4. **Multilayer Perceptron**
Artificial Neural Networks are ML algorithms inspired by the structure and functionality of biological neural networks, where interconnected nodes transmit information through weighted connections that adjust based on their ability to provide the desired outcome [19]. A Multilayer Perceptron (MLP) is a type of ANN characterised by multiple layers of neurons and the use of a non-linear activation function, thus enabling it to learn

complex data patterns. A MLP is an example of a feed-forward neural network, which is the simplest form of ANNs, where information flows unidirectionally through an input layer, one or more hidden layers, and an output layer, as illustrated in Figure 2.3.

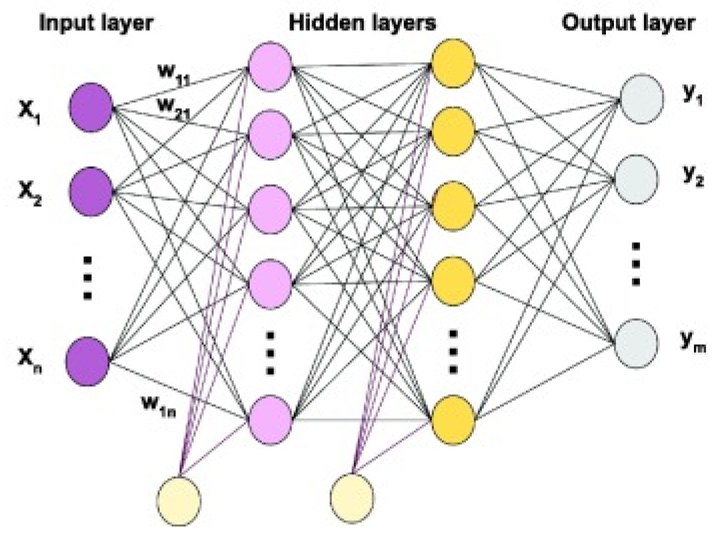


Figure 2.3: The MLP topology [20].

5. **Random Forest**
Random Forest is a robust ML algorithm that constructs an ensemble of decision trees, each trained on randomly sampled subsets of the data [21], as visualised in Figure 2.4. Unlike individual decision trees that are susceptible to bias and overfitting, the ensemble approach enhances accuracy and generalisability by averaging multiple tree outputs which mitigates variance and reduces overfitting. The overall generalisation error of the model is impacted by both the strength of the individual trees and the correlation between them [21].

6. **Decision Trees**
A decision tree is a non-parametric ML algorithm structured as a hierarchical, directed tree comprising a root node, internal nodes, branches, and leaf nodes. The internal nodes iteratively partition the instance space into subspaces based on feature values which facilitates recursive decision-making. Each leaf node is assigned either a class label representing the most probable target value or a probability distribution that reflects the likelihood of different target outcomes [22].

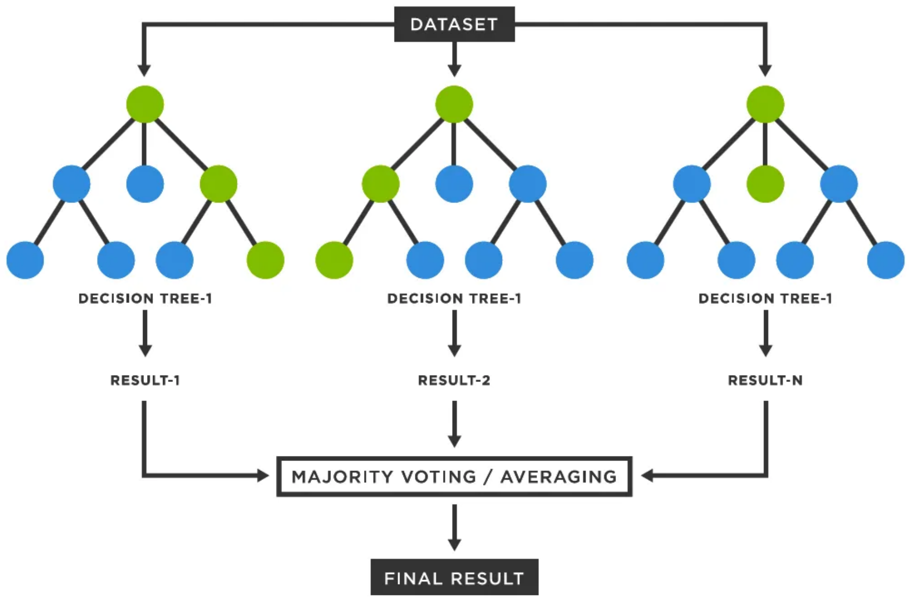


Figure 2.4: The Random Forest topology [23].

2.2.3 Hyperparameter Tuning

Hyperparameters are configurable variables that govern the learning process rather than being derived from the data itself. Their optimisation is critical for enhancing model performance as they influence both accuracy and generalisation. Hyperparameter tuning is the process of systematically training the model with various parameter configurations, evaluating performance through experimental results, and applying statistical analysis to identify the optimal set of parameters. This process is both essential and computationally demanding within the ML pipeline. In this study, the following hyperparameters were tuned for each ML algorithm:

1. **K-Nearest Neighbours (KNN)**
   - **Number of Neighbours:** Defines the number of nearest neighbours considered when predicting the output class.
   - **Weighting Scheme:** Governs the influence of each neighbour on the classification outcome. A uniform weighting assigns equal importance to all selected neighbours, whereas a distance-based weighting gives greater influence to closer neighbours.
2. **Logistic Regression (LR)**

- C **(Regularisation Parameter):** Controls the trade-off between model complexity and generalisation by penalising large parameter values. It is the inverse of regularisation strength, where smaller values enforce stronger regularisation which reduces overfitting while larger values allow the model to fit the training data more closely.
- **Optimisation Algorithm:** Determines the approach to use for minimising the loss function by iteratively updating model parameters. Common solvers include *lbfgs*, *liblinear*, *sag*, *saga*, *newton-cg*, and *newton-cholesky*, each suited depending on the dataset size and structure.

3. **Support Vector Machine (SVM)**
    - C **(Regularisation Parameter):** Regulates the balance between maximising the margin and minimising classification error. A lower C value leads to a larger margin at the cost of potential misclassification but improves generalisability, whereas a higher C penalises misclassification more strictly, potentially leading to overfitting.
    - **Kernel Function:** Transforms the input data into a higher dimensional space to enable non-linear decision boundaries. Common kernel functions include Linear, Polynomial, Gaussian, Sigmoid, and Radial Basis Function (RBF), where each of them are suited to different data structures and classification challenges.
4. **Multilayer Perceptron (MLP)**
    - **Activation Function:** Introduces non-linearity to the network by determining whether a neuron should be activated based on the weighted sum of inputs and a bias term. Common activation functions include ReLU (Rectified Linear Unit), Sigmoid, Identity, and Tanh.
    - **Hidden Layer Size:** Defines the number of neurons within each hidden layer which impacts the model's capacity to capture complex patterns. A greater number of neurons enables the network to learn more intricate representations but also increases computational cost and introduces the risk of overfitting.
5. **Random Forest**
    - **Number of Decision Trees:** Specifies the ensemble size, where a greater number of trees generally improves predictive accuracy but increases computational cost.
    - **Maximum Depth:** Constrains the depth of each individual tree, where the aim should be to balancing model complexity and generalisation. An excessive depth can lead to overfitting where the model memorises training data instead of learning general patterns.

6. **Decision Trees**
   - **Splitting Criterion:** Determines the measure to use to evaluate split quality at each node. Common criteria include Gini impurity, Entropy, and Log-Loss where the goal is to guide the tree building process towards optimal class separation.
   - **Maximum Depth:** Similar to Random Forest, this limits the tree growth to prevent excessive complexity that may reduce generalisability.
   - **Maximum Features:** Specifies the number of features considered for each split.

Model Evaluation

To evaluate the performance of the trained ML model, it is tested against unseen data with the predicted outputs being compared against the true values. Various evaluation metrics exist for classification models, each providing distinct insights into performance. The most widely used metrics are summarised in Table 2.1.

Metric	**Formula**
Accuracy	$\frac{\text{TP} + \text{TN}}{\text{Size of the Dataset}}$
Precision	$\frac{\text{TP}}{\text{TP} + \text{FP}}$
Recall	$\frac{\text{TP}}{\text{TP} + \text{FN}}$
F1 Score	$\frac{2\times\text{Precision}\times\text{Recall}}{\text{Precision} + \text{Recall}}$
Area Under Curve	Measures the area under the ROC curve, which plots the true positive rate against the false positive rate across different classification thresholds.

Table 2.1: Evaluation metrics for classification tasks.

Model performance can also be assessed visually using a confusion matrix which provides insight into class-wise predictions. It displays the distribution of correctly and incorrectly classified instances, revealing potential biases in the model's predictions. An example confusion matrix is presented in Table 2.2.

To assess the model's generalisability, cross-validation is commonly used. This involves partitioning the dataset into multiple subsets where in each iteration one fold is used for validation while the remaining folds are used for training. An advantage of cross-validation is its ability to avoid overfitting, where a model learns patterns specific to the training data but fails to generalise to unseen data. By validating across multiple subsets, cross-validation provides a more reliable measure of the model's robustness and generalisation capability.

	Prediction	
Actual	Positive	Negative
Positive	True Positive (TP)	False Negative (FN)
Negative	False Positive (FP)	True Negative (TN)

Table 2.2: Confusion Matrix.

2.3 K-Fold Cross Validation

K-Fold Cross-Validation is a resampling technique for assessing a model's performance and generalisability. It involves partitioning the dataset into K equally sized subsets with the model trained on $K-1$ folds and validated on the remaining fold. This process is repeated K times, each time using a different fold for validation to ensure that every instance in the dataset is used for both training and validation. The model's performance is then averaged across all K iterations to provide a robust estimate of its generalisability prevent overfitting.

Stratified K-Fold Cross-Validation is an extension of this technique that preserves the class distribution within each fold, making it particularly useful for imbalanced datasets. This is relevant to our study as our dataset contains a disproportionately higher number of individuals with stable MCI compared to those who progress to AD over the two year period. By maintaining class proportions across folds, it ensures that the model is exposed to a representative distribution of converters and non-converters in each iteration which can lead to a more reliable evaluation of its predictive performance.

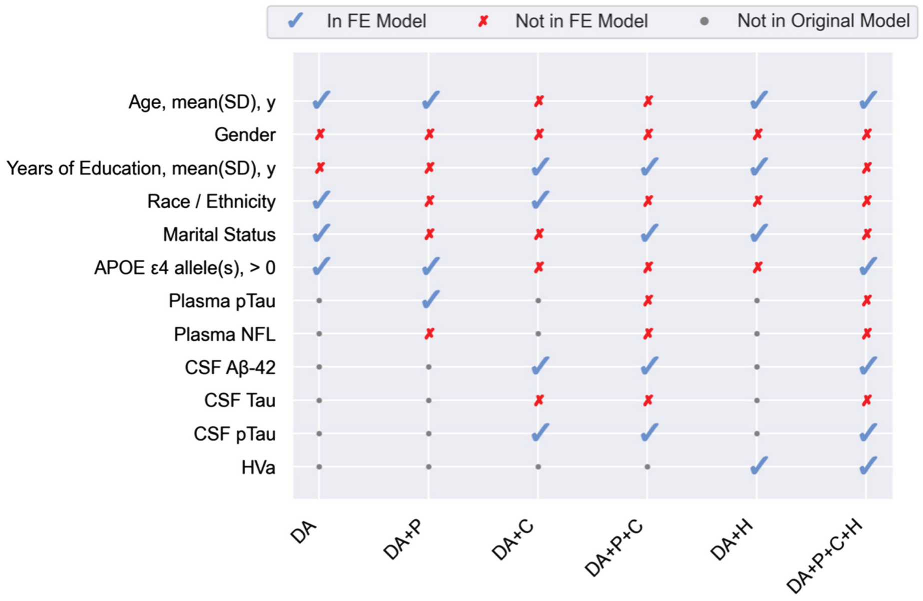
Chapter 3

Literature Review

Numerous prior studies have investigated the prediction of conversion from MCI to AD. However, the input features, methodological approach and observational timescales vary considerably across literatures. Additionally, the predictive potential of plasma biomarkers remains relatively less explored in this context despite their emerging relevance in neurodegenerative research. Therefore, a review was conducted to identify studies that align closely with the objectives of this research. The review is structured into two areas: An evaluation of the classification problem in terms of the predictive features used as inputs to ML models, and an examination of the algorithms utilised to predict MCI progression.

3.1 Model Input

The literature reviewed explored a range of predictors employed in constructing ML models for forecasting MCI to AD progression. In ML, such predictors are known as independent variables and serve to estimate the outcome of a dependent variable, which in this case is disease progression. A key focus across studies was the evaluation of CSF and plasma biomarkers as predictive features, analysing whether they can achieve performance comparable to other established biomarkers like MRI and PET scans. Most studies adopted a multimodal approach, incorporating demographic data, cognitive and neuropsychological assessments, CSF and plasma biomarkers. One study conducted a systematic comparison of various feature combinations to isolate the predictive value of each data type with Figure 3.1 showing the different features used in each dataset. The findings indicated that integrating both CSF and BBMs significantly enhanced model performance, yielding a positive percentage value of 79.1% and negative percentage value of 86.9%, outperforming the models that excluded one or more of these modalities [24].



✓ In FE Model ✗ Not in FE Model ● Not in Original Model

	DA	DA+P	DA+C	DA+P+C	DA+H	DA+P+C+H
Age, mean(SD), y	✓	✓	✗	✗	✓	✓
Gender	✗	✗	✗	✗	✗	✗
Years of Education, mean(SD), y	✗	✗	✓	✓	✓	✗
Race / Ethnicity	✓	✗	✓	✗	✗	✗
Marital Status	✓	✗	✗	✓	✓	✗
APOE ε4 allele(s), > 0	✓	✓	✗	✗	✗	✓
Plasma pTau	●	✓	●	✗	●	✗
Plasma NFL	●	✗	●	✗	●	✗
CSF Aβ-42	●	●	✓	✓	●	✓
CSF Tau	●	●	✗	✗	●	✗
CSF pTau	●	●	✓	✓	●	✓
HVa	●	●	●	●	✓	✓

Figure 3.1: FE Model, Feature Engineered Model; DA, Demographics and APOE4; P, Plasma; C, CSF; H, Adjusted Hippocampal Volume [24].

Another study demonstrated that incorporating at least one biomarker alongside demographic variables and *APOE* $\epsilon4$ allele count significantly improves predictive accuracy, with plasma biomarkers performing comparably to CSF biomarkers which further underscores the practical advantages of plasma biomarkers given their lower cost and reduced invasiveness [24]. Furthermore, emerging research has highlighted the discriminative power of plasma biomarkers such as phosphorylated tau (p-tau) as having the ability to reliably differentiate cognitively normal individuals from those with AD, reinforcing their value as essential inputs for model development [25].

3.1.1 Missing Data

A common limitation observed across many studies was the presence of missing data which poses a threat to the robustness and validity of predictive models. Figure 3.2 illustrates the extent of missing data in one such study, highlighting the challenges this introduces in maintaining statistical rigour. For example, in a cohort of 963 individuals initially diagnosed with MCI, 229 had no 2-year follow up data. Among those with available follow up, additional predictor variables were missing. The study addressed this by excluding those participants, thereby further reducing the sample size and potentially diminishing the model's sensitivity and generalisability [24]. Alternative approaches have also been explored to mitigate this limitation within the ADNI dataset. Notably, MICE imputation has been shown to enhance statistical efficiency while preserving the validity of inferences [26], hence offering a more robust solution to handling incomplete data.

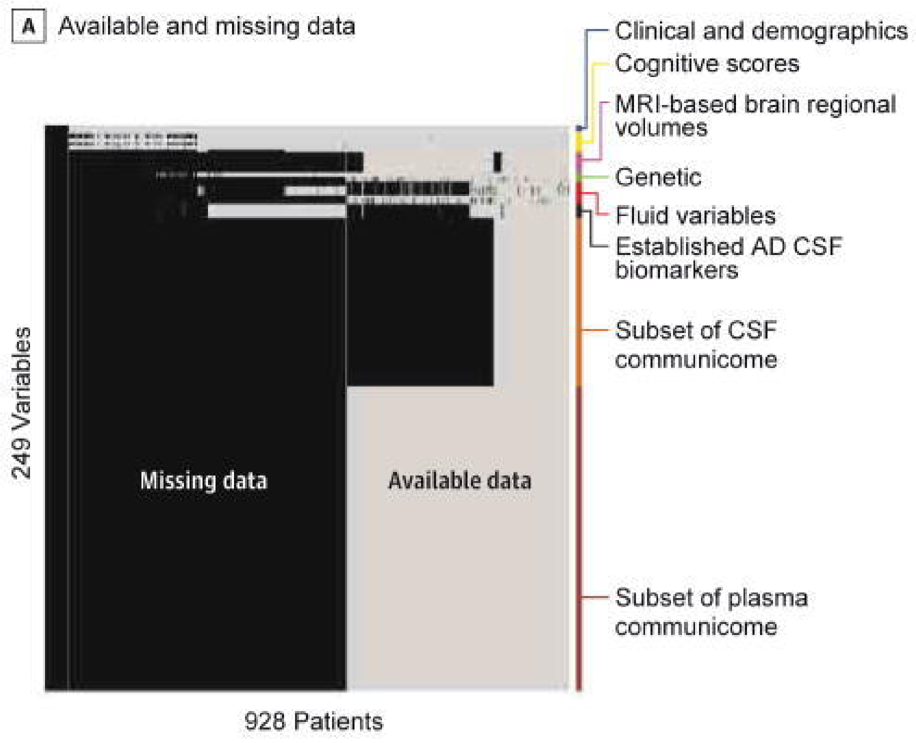


Figure 3.2: The quantity of missing data visualised for 928 participants with MCI [27].

3.1.2 Limitations

Some studies were constrained by the limited availability of specific plasma biomarkers within the ADNI dataset. For example, key biomarkers such as the $A\beta_{42}/A\beta_{40}$ ratio was not assessed in one investigation, hence the need for further research to evaluate their potential additive value in predicting disease progression [24]. In another study, the selection criteria led to a small cohort of only 94 MCI participants, primarily because of CSF multiplex data being unavailable for majority of the participants [27]. This limitation made it difficult to find another subset of patients as a validation set to confirm the studies findings. Consequently, future work should explore alternative validation strategies to assess the robustness and reproducibility of results.

3.2 Classification Methods

Various studies have investigated a range of ML models and configurations to develop classifiers capable of predicting the conversion from MCI to AD over different timescales. These efforts have focused on assessing model reliability, discriminative performance and clinical applicability.

3.2.1 Random Forest

One study explored disease progression over a 2-year follow up period using a Random Forest model, an ensemble ML algorithm known for its robustness and capacity to estimate feature importance. The model was trained on a dataset comprising demographic variables, APOE4 allele count, CSF and plasma biomarkers, and evaluated using 5-fold cross-validation, achieving a sensitivity of 62.2% and a specificity of 70% [24]. Hyperparameter tuning was performed within the cross-validation framework, highlighting the importance of optimising model parameters to enhance predictive performance, an approach that will also be adopted in our methodology.

3.2.2 Logistic Regression

One study investigated the role of CSF biomarkers in predicting the progression from MCI to AD using a Logistic Regression (LR) model, observing that it attributed to an 11% increase in classification accuracy from 74.1% to 85.1% [28], highlighting the strong prognostic value of CSF biomarkers in predicting disease progression from MCI to AD.

3.2.3 Support Vector Machine

Finally, another study constructed a Support Vector Machine (SVM) model that integrated imaging, CSF and blood biomarkers to classify individuals with MCI at risk of progression to AD, managing to achieve 95% ± 3% classification accuracy [29].

Chapter 4

Experimental Methods

4.1 Methods

This project was developed using Python 3.10.12 and leveraged several key libraries: Scikit-Learn for implementing ML algorithms, Pandas and NumPy for data manipulation and numerical analysis, and Matplotlib alongside Seaborn for data visualisation. All development was carried out locally using Visual Studio Code as the Integrated Development Environment (IDE), running Jupyter Notebooks on a machine equipped with an Intel(R) Core(TM) i5-8250U CPU and x86_64 architecture.

4.2 Data

This study utilised data from the Alzheimer's Disease Neuroimaging Initiative (ADNI) database which was accessed via the Laboratory of Neuro Imaging (LONI) repository (adni.loni.usc.edu) in October 2024. Initiated in 2004 as a public–private partnership under the leadership of Michael W. Weiner, ADNI is supported by the National Institute on Aging, the Foundation for the National Institutes of Health, the Alzheimer's Association, and multiple pharmaceutical and biotechnology partners. The primary objective of ADNI is to develop and validate sensitive and reliable biomarkers such as MRI, PET, CSF and plasma biomarkers for use in AD clinical trials [30]. The initiative also seeks to improve diagnostic accuracy and provide deeper insights into the progression of AD. Further information can be found at `www.adni-info.org`.

Participants were selected according to the ADNI eligibility criteria, designed to ensure cohort consistency and data quality. To be included in the study, subjects were required to meet all of the following [31]:

- Hachinski Ischemic Score of less than or equal to 4
- Stable use of permitted medications for at least four weeks prior to screening
- Geriatric Depression Scale score of less than 6
- Availability of a study partner with 10+ hours of contact per week, either in person or by telephone who could accompany the participant to clinical visits
- Adequate visual and auditory acuity for neuropsychological testing
- Good general health with no medical conditions that could interfere with study participation
- Minimum of six years of education or equivalent work history
- Fluency in either English or Spanish
- For female participants: postmenopausal for at least two years or surgically sterile
- Willingness and ability to complete a three-year imaging study
- Consent to DNA extraction for APOE $\epsilon 4$ genotyping
- Agreement to undergo blood and urine sampling for biomarker analysis;
- No contraindications to MRI
- Not currently enrolled in any other investigational studies

4.3 Model

All available features were used as inputs during the development of both individual models and the final ensemble model. Prior to model construction, the dataset was partitioned into an 80:20 split for training and testing respectively, while hyperparameter tuning was performed using 5-fold cross validation on the training subset to optimise performance and mitigate overfitting. The dataset comprised 783 participants, of whom 590 remained clinically stable with MCI throughout the two-year period and 193 progressed to a diagnosis of AD.

4.4 Considerations

While the hyperparameter tuning phase involved the exploration of a wide range of configurable parameters and value combinations, the exhaustive evaluation of all possible configurations is computationally infeasible. Therefore, it is not possible to definitively claim that the optimal model configuration has been achieved. Despite efforts to preserve data integrity through rigorous oversampling techniques and the imputation of missing values, a substantial proportion of the dataset used in model training was synthetically generated. This synthetic data, although necessary to address class imbalance and sparsity, may diverge from real world distributions, thus potentially affecting the realism of the model. As such, the aim of this research is not to assert definitive superiority but to propose an architecture that demonstrates meaningful potential in predicting the progression from MCI to AD using CSF and plasma biomarkers. Furthermore, while the ADNI dataset offers a source of neuroimaging and clinical data, it is not derived from a population based study. Consequently, its demographic and clinical characteristics may not fully represent the heterogeneity observed in wider and more diverse populations. This highlights the importance of validating the model's generalisability across independent and heterogeneous cohorts to ensure genuine clinical applicability.

Chapter 5

Preprocessing

5.1 Data Exploration

The primary dataset used in this project was ADNIMERGE which is a collation of the key data about participants from the ADNI1, ADNI2, ADNI3, and ADNIGO phases. The plasma biomarker data had to be obtained separately from each of these studies from the ADNI repository and then merged into the other dataset.

For this study, we used only a subset of the predictors in the ADNIMERGE dataset as we were mainly focusing on the usefulness of CSF and plasma biomarkers in the prediction of the conversion of MCI to AD over a two-year duration. One set of features used were the demographic information of the participants such as their age, gender, years of education, marital status and racial category. The clinical variable used was the Functional Activities Questionnaire (FAQ), a 10 item collateral-report scale with 4 ordinal responses within each item, with the following possible categories [32]:

0. Normal
1. Has difficulty but does by self
2. Requires assistance
3. Dependent

The neuropsychological tests included were Mini-Mental State Examination (MMSE), all available variations of the Rey Auditory Verbal Learning Test (RAVLT), the Cognitive Subscale Alzheimer's Disease Assessment Scale and the modified Preclinical Alzheimer's Cognitive Composite (mPACC) score. The Mini-Mental State Examination is a 30-item questionnaire for evaluating cognitive domains including orientation, memory, attention and language, providing a global assessment of cognitive functions [33]. The Rey Auditory Verbal

Learning Test is a verbal memory assessment in which participants are asked to recall words from a presented list to evaluate multiple facets of their memory, including immediate recall, learning rate, forgetting, and percent forgetting [34]. The Alzheimer's Disease Assessment Scale is a comprehensive cognitive and behavioural evaluation tool for measuring cognitive impairments in areas such as recall, recognition, and praxis, the score of each being summed to get a score between 0 and 70 [35]. Our study specifically focuses on three components: ADAS11, ADAS13 and ADASQ4. mPACCdigit is a modified version of the Preclinical Alzheimer's Cognitive Composite (PACC) which combines global recognition, episodic memory and timed exective function. The mPACCdigit includes the Digit Symbol Substitution Test (DSST) while mPACCtrailsB uses a log-transformed Trails B score as a proxy for DSST [36]. The abbreviations of some of the features in the dataset are listed in Table 5.1.

Abbreviation	Description
DX	Diagnosis
PTGENDER	Sex
PTMARRY	Marital Status
PTRACCAT	Racial Category
APOE4	Number of APOE4 alleles
MMSE	Mini-Mental State Examination
RAVLT_immediate	Rey Auditory Verbal Learning Test Immediate Score
RAVLT_learning	Rey Auditory Verbal Learning Test Learning Score
RAVLT_forgetting	Rey Auditory Verbal Learning Test Forgetting Score
RAVLT_perc_forgetting	Rey Auditory Verbal Learning Test Percentage Forgetting Score
ADAS11	The Cognitive Subscale Alzheimer's Disease Assessment Scale (11 tasks)
ADAS13	The Cognitive Subscale Alzheimer's Disease Assessment Scale (13 tasks)
ADASQ4	Task 4 Of The Cognitive Subscale Alzheimer's Disease Assessment Scale (Delayed Word Recall)
mPACCdigit	modified Preclinical Alzheimer's Cognitive Composite with Digit Symbol Substitution
mPACCtrailsB	modified Preclinical Alzheimer's Cognitive Composite with Trails B
FAQ	Functional Activities Questionnaire

Table 5.1: Abbreviations of some of the input features in the dataset.

The Cerebrospinal Fluid (CSF) biomarkers employed in this study include amyloid-beta 42 ($A\beta_{42}$), total tau (t-tau), and phosphorylated tau (p-tau). Plasma biomarkers that were included were the $A\beta_{42/40}$ ratio, Neurofilament Light Chain (NfL) and phosphorylated tau (p-tau), all of which are considered key indicators of neurodegenerative processes associated with AD and its prodromal stages.

In addition to the biomarkers, other features were also incorporated that are associated with AD progression. This includes the presence of the APOE $\epsilon 4$ allele count and the baseline clinical diagnosis at the initial visit with the participants used in our study being in either early MCI (EMCI) or late MCI (LMCI) at this stage based on their performance on the Logical Memory II subtest of the Memory Scale [37].

Our dataset included the values of these variables both at the baseline visit where the participants were diagnosed with MCI and these values at the most latest reading within the two year threshold. In total, our model integrated 41 predictive features, of which 37 are continuous variables and the remaining 4 are categorical. We used data from 783 patients, all of whom had a baseline diagnosis of MCI. 590 of these participants remained in the MCI stage throughout the two-year period while the remaining 193 transitioned into AD during this period. Table 5.2 provides a summary of the demographic and clinical characteristics of all the participants included in the study at their last readings after performing imputation on the dataset while Figure 5.1 shows the distribution of all the categorical features in the dataset.

		All Participants	Stable MCI	AD
	Count	783	590	193
	Age, Mean (SD)	73.04 (7.45)	72.78 (7.48)	73.82 (7.35)
Demographics	Education, Mean (SD)	15.98 (2.75)	16.01 (2.75)	15.90 (2.76)
	Married (%)	78.29	76.10	84.97
	Male/Female	467/316	348/242	119/74
	RAVLT Immediate	30.75 (11.46)	33.53 (11.04)	22.27 (8.11)
	RAVLT Learning	3.35 (2.61)	3.82 (2.64)	1.91 (1.90)
	RAVLT Forgetting	4.79 (2.39)	4.85 (2.52)	4.60 (1.96)
	RAVLT Percent Forgetting	72.86 (32.71)	67.13 (33.72)	90.39 (21.44)
Neuropsychological	ADAS Q4	6.30 (2.75)	5.64 (2.68)	8.28 (1.84)
Tests: Mean (SD)	ADAS 11	12.93 (7.07)	11.11 (5.78)	18.50 (7.48)
	ADAS 13	20.23 (9.89)	17.61 (8.49)	28.23 (9.56)
	MMSE	26.00 (3.43)	26.79 (2.88)	23.56 (3.86)
	FAQ	6.80 (6.97)	4.55 (5.48)	13.68 (6.52)
	mPACCdigit	-8.42 (6.58)	-6.80 (5.91)	-13.40 (6.03)
	mPACCtrailsB	-8.05 (6.36)	-6.37 (5.54)	-13.21 (5.93)

Table 5.2: Participant overview at the two-year follow up from the initial visit after data imputation.

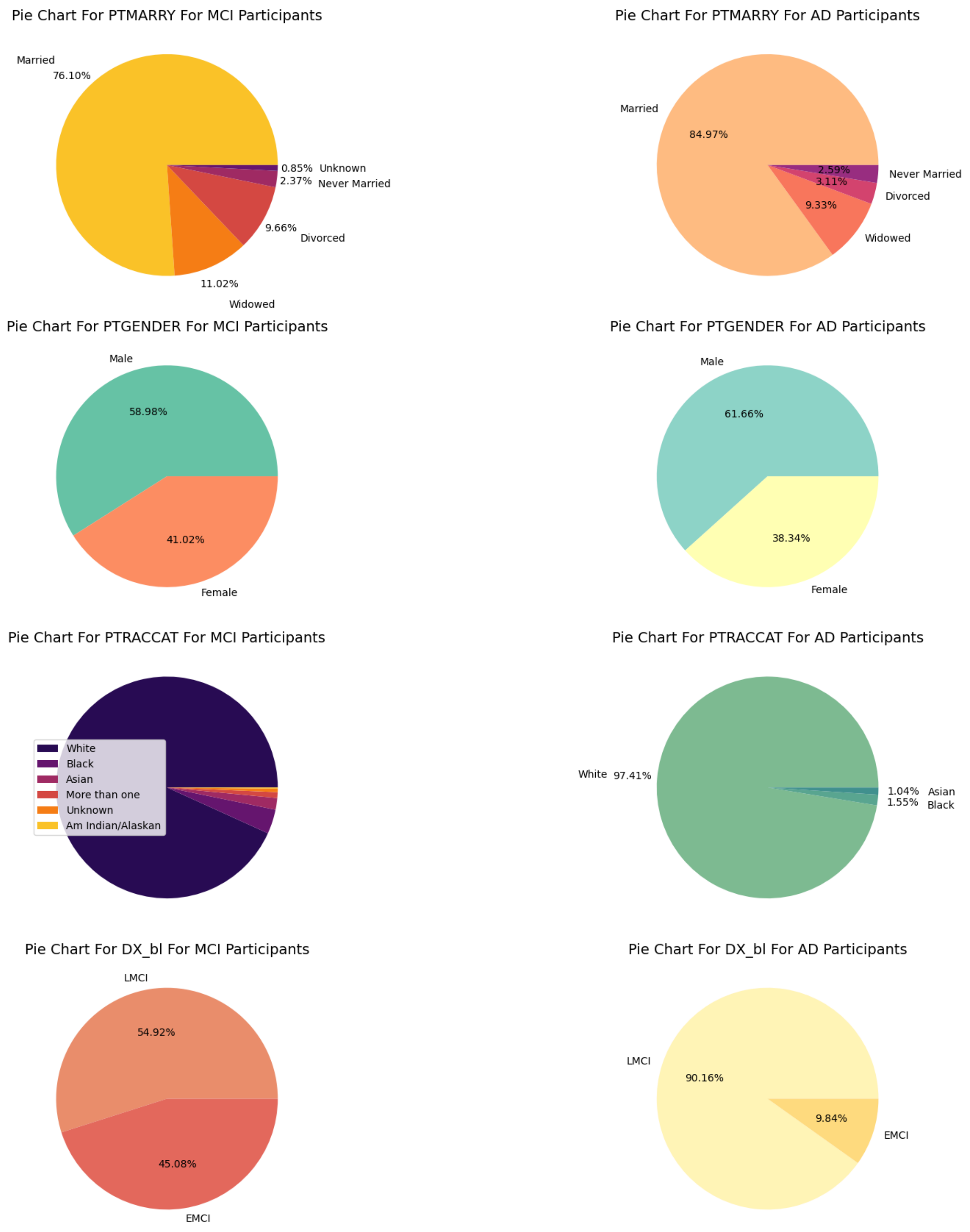


Figure 5.1: Pie charts of the categorical features in the dataset.

5.2 Missing Data

Given the multi-modal nature of the dataset, a significant portion of the data was missing, as illustrated in Figure 5.2. Due to the small dataset, discarding participants that have any missing data wasn't a feasible option as this would leave a very small number of participants. Instead, a comprehensive evaluation of various data imputation methods was conducted and compared for performance.

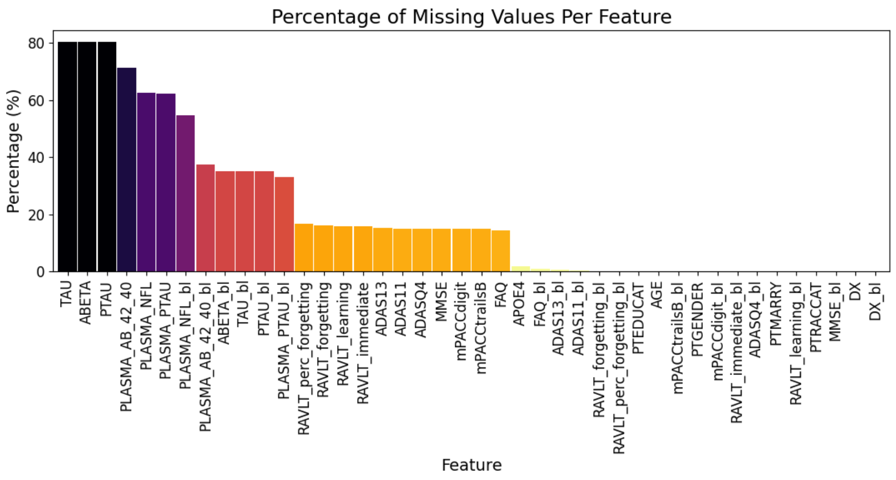


Figure 5.2: Percentage of missing data per feature in the dataset.

Initially, we generated a sample dataset comprising only the participants with complete values for each feature, which was only 13 participants. We then intentionally masked 20% of the data in this set to simulate missing values (NaN) to compare the performance of the imputation methods in predicting these values as compared to the true values. We applied K-Nearest Neighbours (KNN) imputation to predict the missing values, assessing the model's accuracy by computing the Mean Absolute Error (MAE) and Mean Squared Error (MSE) for each K value ranging from 1 to 10. The results, as shown in Figure 5.3, indicated that the 2-NN configuration yielded the best performance with a MAE of 2.34 and a MSE of 205.19. Subsequently, we tested Multiple Imputation by Chained Equations (MICE) with a Random Forest imputation model (MICE Forest) as an alternative approach. Using the same sampled dataset with missing values, the MAE and MSE was evaluated across various combinations of iteration, ranging from 3 to 5, and dataset splits, ranging from 3 to 10, to the MICE Forest model to identify the optimal configuration. MICE Forest outperformed KNN imputation, as illustrated in Figure 5.4, yielding superior results when compared to the true values. The optimal number of iterations for running the MICE Forest model was found to be 4 with the data split into 4 sets, with this configuration achieving a MAE of 1.98 and a MSE of 103.62. Based on the results obtained, MICE Forest imputation was applied to the complete dataset.

Finally, a comparative analysis of the statistical metrics was performed for each feature that initially had more than 30% missing data before and after imputation. This allowed us to evaluate whether the data distribution of those features were range bound and aligned after imputing the data. Figures 5.5 and 5.6 show the key statistics of these features before and after imputation respectively.

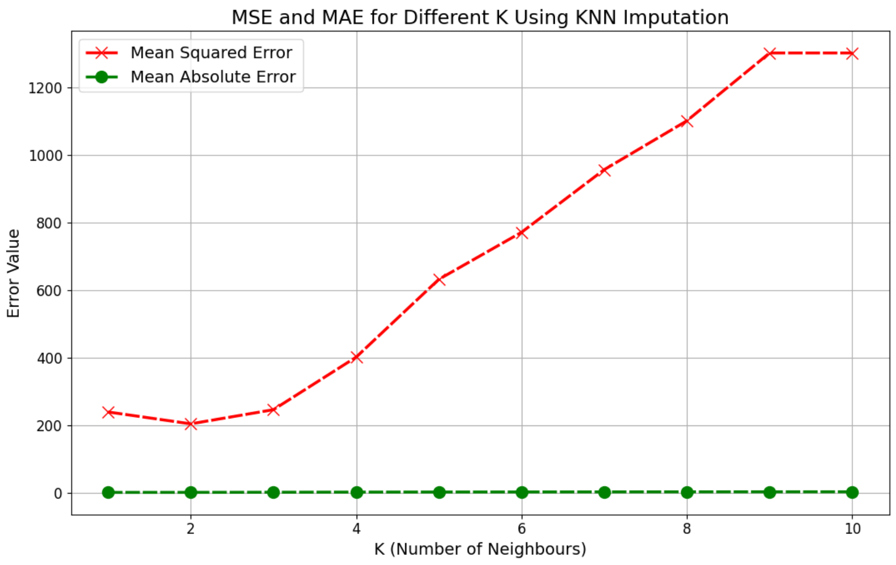


Figure 5.3: The Mean Squared Error and Mean Absolute Error for varying K-values in KNN imputation.

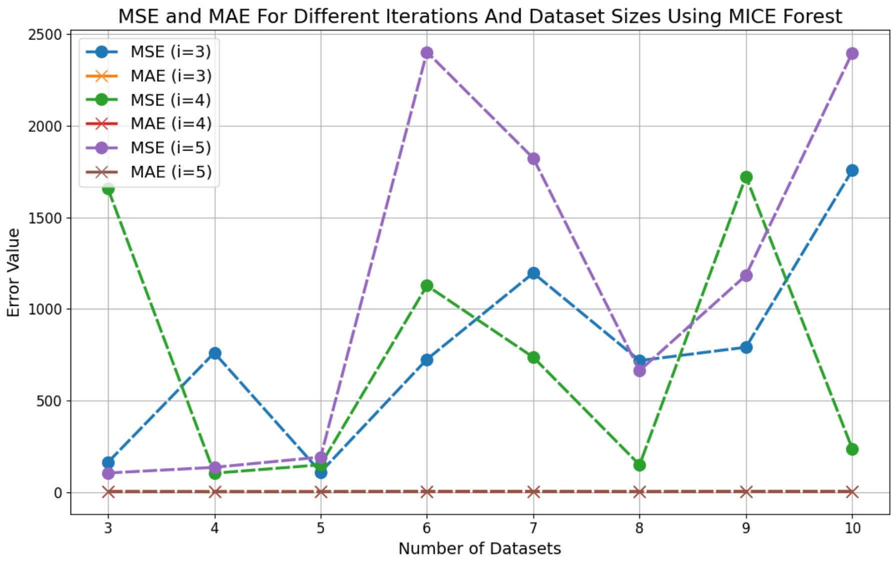


Figure 5.4: The Mean Squared Error and Mean Absolute Error across different iteration and dataset split configurations for MICE Forest imputation.

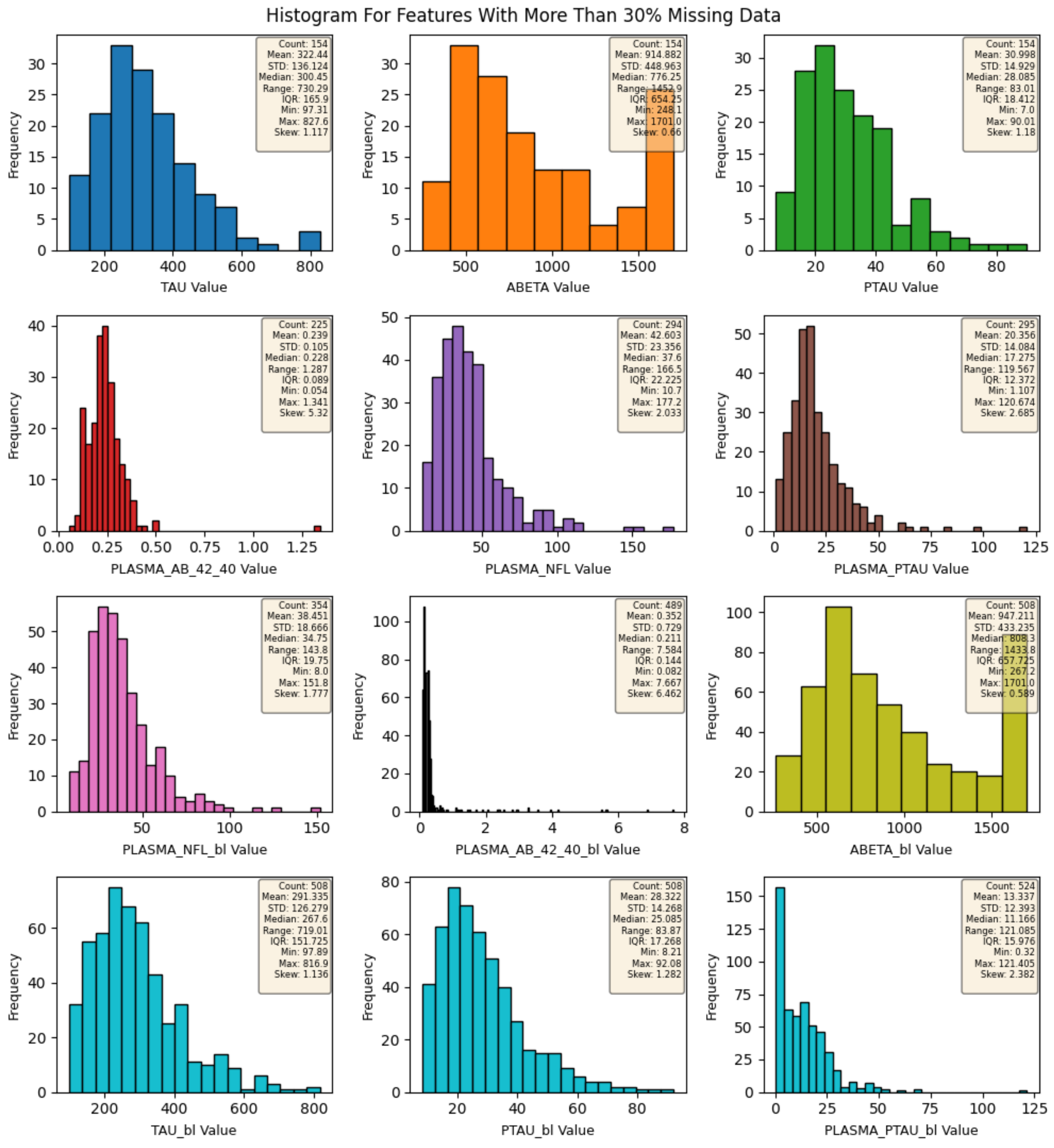


Figure 5.5: Histogram and statistics of the features before imputation which have more than 30% data missing.

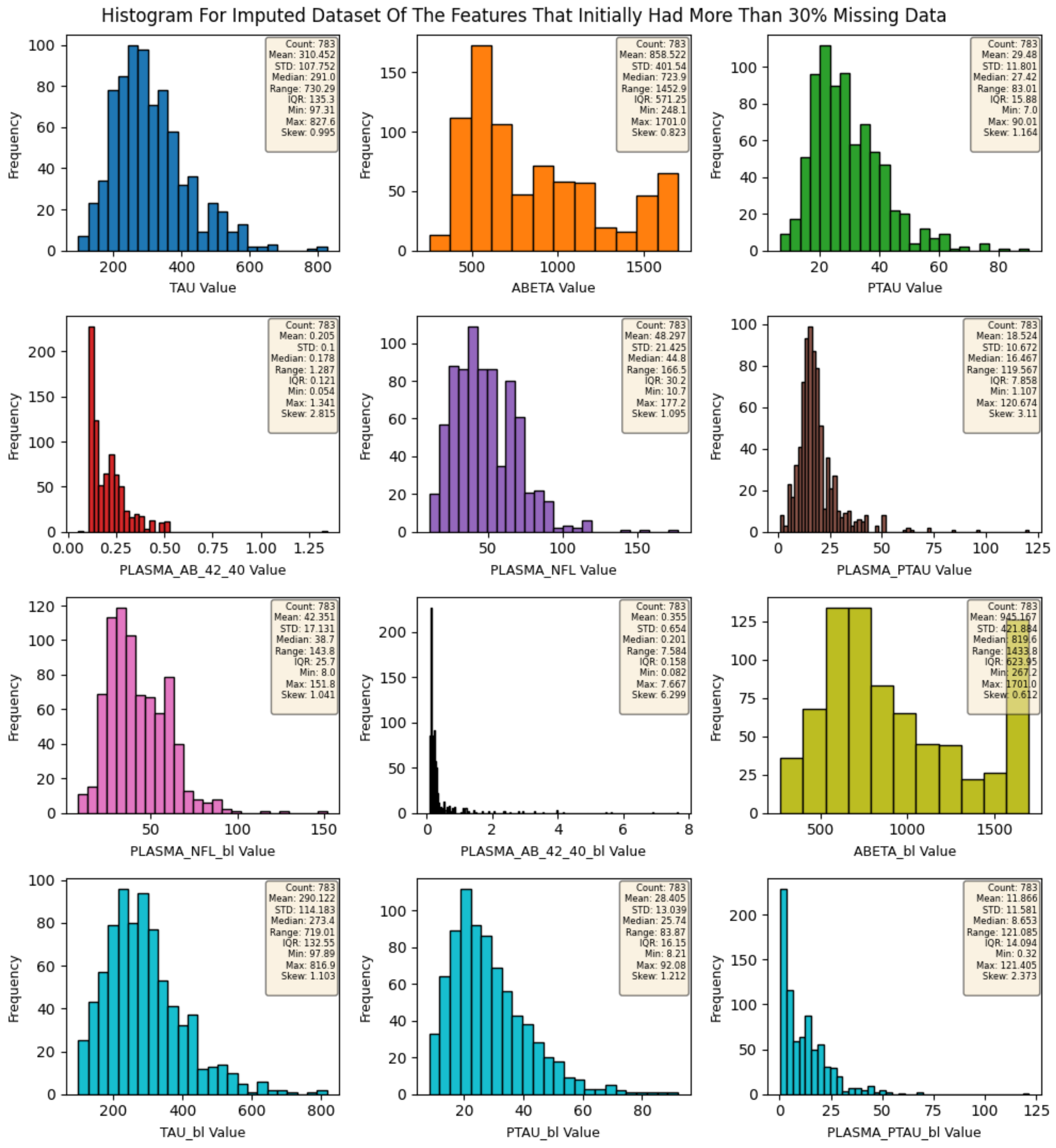


Figure 5.6: Histogram and statistics of the features after imputation which initially had more than 30% data missing.

5.3 Class Imbalance

Class imbalance represents a significant challenge in classification tasks as it can lead models to disproportionately favour the majority class, resulting in poor generalisation and biased predictions against the minority class [38]. This occurs because the model receives insufficient examples from the under-represented class which impairs its ability to learn meaningful patterns associated with that group. In the context of our dataset, 590 out of 783 participants remained in MCI over the two-year period while 193 progressed to AD, reflecting a pronounced imbalance towards one class. Training models on such skewed data risks overfitting to the dominant class ('Stay in MCI'). Therefore, it was essential to explore methods to mitigate the imbalance and ensure the model could learn discriminative features from both classes to enhance its ability to generalise beyond the training data.

5.3.1 Undersampling

One strategy considered was undersampling which involves reducing the number of instances in the majority class to match the size of the minority class. While effective in balancing class distributions, this method wasn't suitable for our study due to the limited number of participants who converted to AD ($N = 193$). Reducing the majority class to this size would substantially diminish the overall dataset and risk the loss of valuable information while introducing sampling bias. Such a reduction could compromise the model's ability to generalise, hence negatively impacting its predictive performance. Undersampling is more appropriate in scenarios involving large datasets where the majority class is sufficiently represented such that discarding samples does not result in significant information loss. Furthermore, it can also improve computational efficiency by reducing storage and training time. An example of random undersampling is illustrated in Figure 5.7 with the majority class downsampled to achieve class balance.

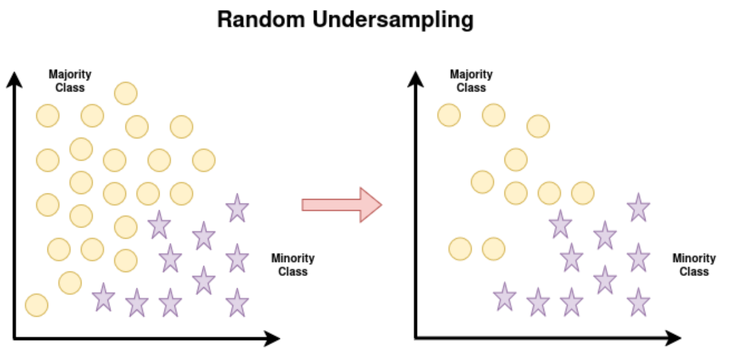


Figure 5.7: Example of random undersampling.

5.3.2 Oversampling

Oversampling is a common strategy to address class imbalance by augmenting the minority class to match the size of the majority class, either through duplication or the generation of synthetic samples. This helps mitigate model bias towards the majority class, potentially enhancing classification accuracy and improving generalisability to unseen data. However, this also has its limitations as synthetic data may fail to capture the true distribution of the minority class which may introduce artificial patterns that do not reflect the underlying biological or clinical reality, leading to overfitting and misleading predictions. Given the class imbalance present in our dataset, oversampling was the more appropriate measure. Consequently, we explored several oversampling methods including Random Oversampling, Synthetic Minority Oversampling Technique (SMOTE), and Adaptive Synthetic Sampling (ADASYN).

Random

Random Oversampling addresses class imbalance by replicating existing samples from the minority class. While this technique is straightforward and effective in increasing minority representation, it poses a risk of overfitting due to the duplication of identical instances. A visualisation of this method is shown in Figure 5.8.

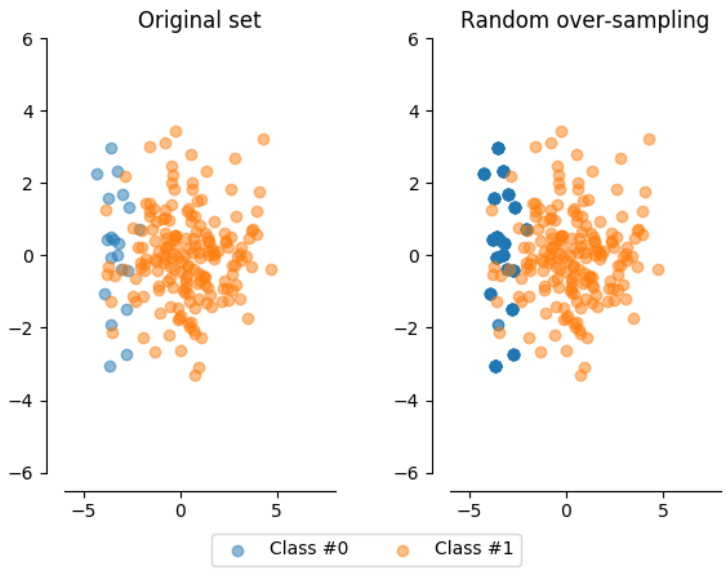


Figure 5.8: Example of random oversampling [39].

SMOTE

Synthetic Minority Oversampling Technique (SMOTE) enhances the diversity of the minority class by generating synthetic samples through linear interpolation. The high level steps of the algorithm are as follows:

1. Identify the KNN of a given minority sample.
2. For each neighbour, compute the difference vector between the sample and its neighbour.
3. Multiply this vector by a random number in the range [0, 1] to scale the distance
4. Add the resulting vector to the original sample to generate a synthetic example within the feature space.
5. Repeat this process for multiple neighbours as specified by the user.

ADASYN

Adaptive Synthetic Sampling (ADASYN) builds upon the principles of SMOTE by generating synthetic data adaptively, focusing more on minority class examples that are harder to learn. This targeted approach increases classifier performance in regions where class imbalance is most pronounced. The algorithm operates as follows:

1. Compute the degree of class imbalance:

   $$d = \frac{m_{min}}{m_{maj}} \tag{5.1}$$

   where m_{min} and m_{max} are the number of minority and majority class samples respectively. If d falls below a predefined threshold, the algorithm proceeds.

2. Calculate the total number of synthetic samples to be generated:

   $$G = \beta(m_{maj} - m_{min}) \tag{5.2}$$

   where β defines the desired balance level (e.g., $\beta = 1$ for full balance).

3. For each minority sample, identify its KNN and compute:

   $$r_i = \frac{\text{Number of majority samples among the } K \text{ neigbours}}{K} \tag{5.3}$$

   This quantifies the local learning difficulty where a higher r_i indicates a more challenging region due to the dominance of majority samples.

4. Normalise the r_i values to obtain:

$$\hat{r}_i = \frac{r_i}{\sum r_i} \tag{5.4}$$

5. Determine the number of synthetic samples G_i to generate for each minority sample:

$$G_i = G\hat{r}_i \tag{5.5}$$

6. For each G_i, generate synthetic examples by selecting two random minority samples, x_i and x_j, within the neighbourhood and interpolating:

$$x_i + \lambda(x_j - x_i) \tag{5.6}$$

   where λ is a random scalar in the range [0, 1].

5.3.3 Results

After experimenting with all three oversampling techniques, ADASYN emerged as the most effective approach for generating synthetic data to address class imbalance. Unlike simpler methods, ADASYN adaptively focuses on generating synthetic samples for minority class instances that are harder to classify which enhances the model's ability to learn from complex patterns in the data. To further assess the quality of oversampling, dimensionality reduction to two principal components was performed, allowing for a visual comparison of the resulting distributions. As illustrated in Figure 5.9, the samples generated by ADASYN aligned more closely to the structure of the original data compared to those produced by SMOTE.

To avoid the risk of data leakage, oversampling was applied exclusively to the training set as this could lead to biased performance metrics due to the model inadvertently learning from test data. This separation helped with preserving the integrity of the model evaluation phase. The final training set comprised 940 samples, with 469 participants remaining with a diagnosis of MCI over the two-year period, and 471 progressing from MCI to AD.

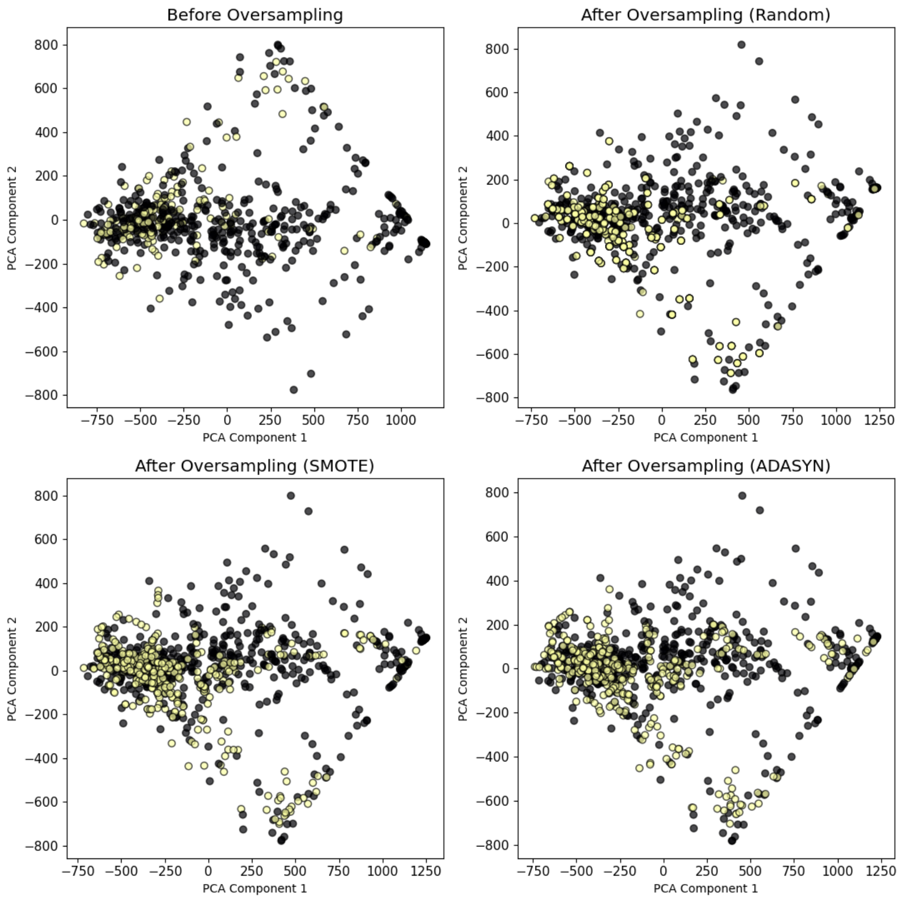


Figure 5.9: Visualisation of the different oversampling methods in a 2-dimensional space.

5.4 Further Preprocessing

Additional preprocessing steps applied to the dataset included:

- Encoding non-numerical categorical variables into numerical representations to enable model interpretability and computation. This included features such as gender, marital status, racial background, and clinical diagnosis (AD or MCI).
- Converting some CSF biomarker values that were initially represented as strings into their corresponding numerical formats to ensure consistency and compatibility.

- Restricting the dataset to a two-year observational window and retaining only the biomarker data from each individual's baseline visit and their final recorded visit within this period.

5.5 Feature Selection

To evaluate the predictive contribution of each feature, we conducted a manual feature selection process to identify critical features and eliminate redundancy. The dataset was partitioned into several subsets to allow for comparative analysis during the evaluation phase. This was useful for determining the individual and combined utility of different data types in influencing the model output. The baseline group comprised demographic, clinical, and neuropsychological assessments, along with the participants initial diagnosis. Two extended subsets were then created: one incorporating CSF biomarkers and the other integrating plasma biomarkers. Finally, a comprehensive dataset that encompassed all available features was constructed. A summary of each dataset configuration and its respective feature composition is provided in Table 7.1.

Chapter 6

Model Construction

The ensemble model was constructed by training and optimising the hyperparameters of six individual models. Hyperparameter tuning was conducted using a grid search approach with 5-fold cross validation, applied to the oversampled training set, while a separate testing set was used to evaluate the performance of each hyperparameter configuration. For each model, the hyperparameter combination yielding the highest classification accuracy was selected. Various ensemble configurations were then constructed using these optimised models, and the ensemble achieving the highest overall accuracy was chosen as the final solution. This section provides a detailed account of this process and the associated experiments, while the specific roles of the optimised hyperparameters were previously outlined in Section 2.2.3.

6.1 Developing K-Nearest Neighbour Model

6.1.1 Hyperparameters

For the KNN model, the hyperparameters tuned were the number of nearest neighbours (`n_neighbours`) and the weighting scheme (`weights`). The values explored for these parameters are summarised in Table 6.1.

Hyperparameter	Values
`n_neighbours`	$1, 2, \ldots, 15$
`weights`	`uniform`, `distance`

Table 6.1: Hyperparameter values tested for the K-Nearest Neighbours model

The mean classification accuracy across folds computed separately for uniform and distance-based weighting is visualised in Figure 6.1.

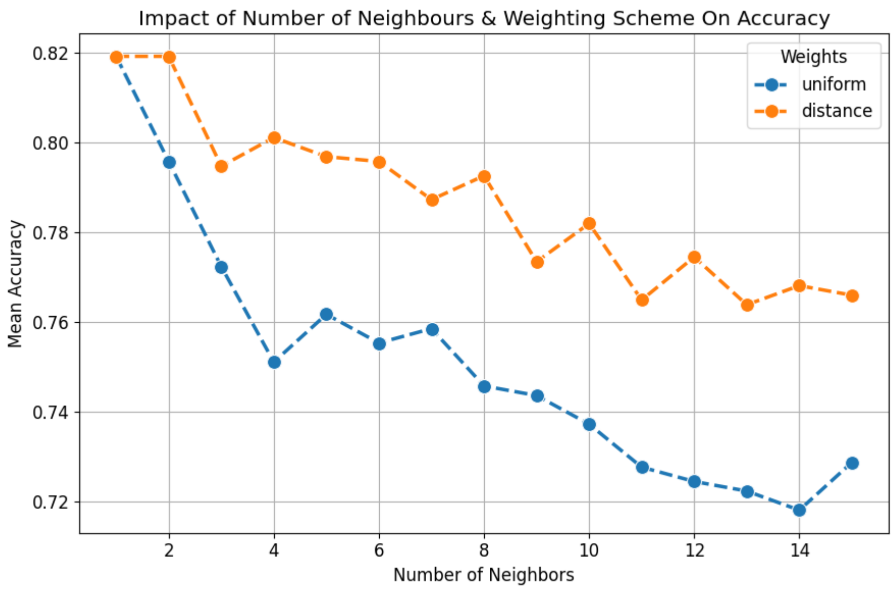


Figure 6.1: Mean classification accuracy across folds for different values of K and weighting schemes.

6.1.2 Results

As shown in Figure 6.1, the optimal configuration was found to be `n_neighbours` = 1 with distance-based weighting, yielding a mean accuracy of 0.82. Additionally, it is evident that the distance-weighted scheme performed equally or better than the uniform method for each value of K explored. These results were interesting as selecting a single nearest neighbour is often associated with overfitting as the model directly adopts the label of the closest training data point. Such a K value is often associated with an increase in sensitivity to noise and outliers, potentially leading to highly variable decision boundaries. However, in this case, the high accuracy could suggest that the data distribution may naturally favour a lower K value.

6.2 Developing Logistic Regression Model

6.2.1 Hyperparameters

The LR model was optimised by tuning the regularisation parameter (C) and the choice of optimisation algorithm (`solver`). The values tested are summarised in Table 6.2.

Hyperparameter	Values
`C`	0.001, 0.01, 0.1, 1, 10
`solver`	`lbfgs`, `liblinear`, `newton-cg`, `newton-cholesky`, `sag`, `saga`

Table 6.2: Hyperparameter values tested for the Logistic Regression model

The effect of varying C and the choice of `solver` on model performance is illustrated in Figure 6.2. The graph suggests that the optimal hyperparameter configuration for this model was able to achieve a marginally superior accuracy as compared to the KNN model.

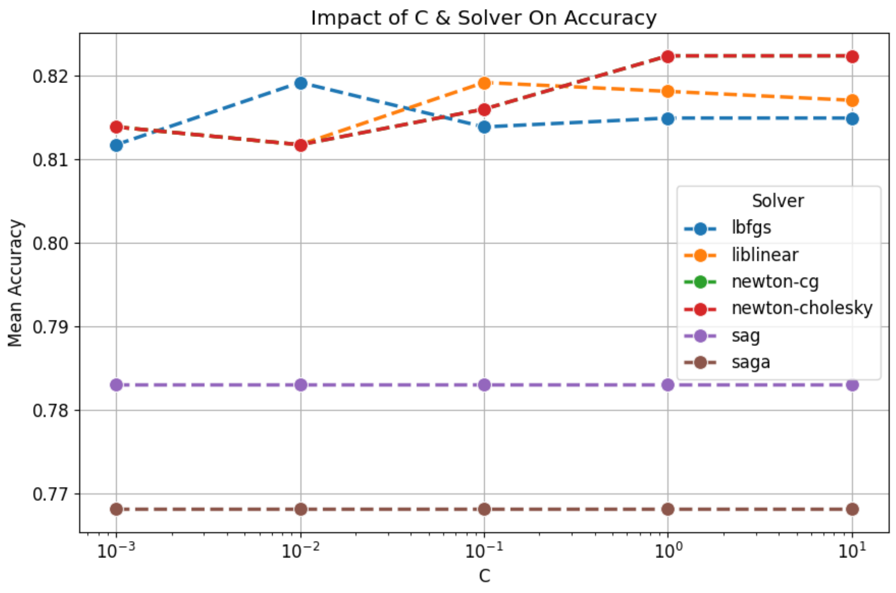


Figure 6.2: Mean classification accuracy across folds for different values of C and optimisation methods.

6.2.2 Results

The results show that the highest mean classification accuracy (0.822) was achieved for $C = 1$ and $C = 10$ with the `newton-cg` and `newton-cholesky` solvers. Notably, these two solvers exhibited identical performance across all tested values of C, leading to overlapping lines in Figure 6.2. This consistency suggests that both solvers effectively optimise the model under these conditions, potentially due to their shared reliance on second order optimisation techniques.

6.3 Developing Support Vector Machine Model

6.3.1 Hyperparameters

The hyperparameters tuned for the SVM model were the regularisation parameter (C) and the kernel function (`kernel`). The search space for these parameters is summarised in Table 6.3.

Hyperparameter	Values
`C`	0.001, 0.01, 0.1, 1, 10
`kernel`	`linear`, `poly`, `rbf`, `sigmoid`

Table 6.3: Hyperparameter values tested for the Support Vector Machine model

The influence of C on model performance for each kernel function is illustrated in Figure 6.3. The optimal hyperparameter configuration for this model resulted in superior performance compared to the previously evaluated models, achieving the highest mean accuracy across folds.

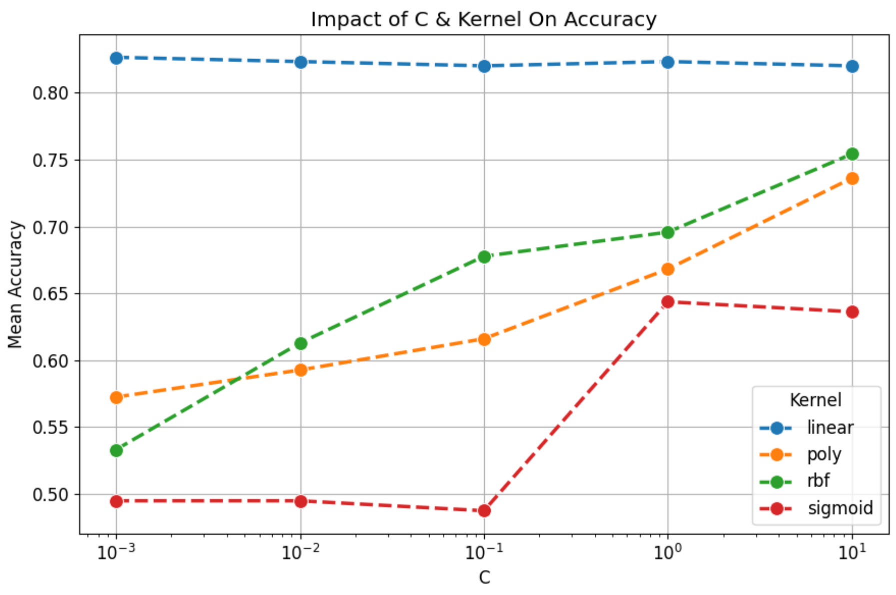


Figure 6.3: Mean classification accuracy across folds for different values of C and kernel functions.

6.3.2 Results

The findings demonstrate that the highest mean accuracy (0.827) was achieved with a `linear` kernel and $C = 0.001$. Figure 6.3 highlights the significant influence of the kernel function on model performance with the `linear` kernel consistently outperforming the others, suggesting that kernel selection plays a crucial role in classification accuracy than fine tuning the regularisation parameter alone.

6.4 Developing Multilayer Perceptron Model

6.4.1 Hyperparameters

The performance of the MLP model was optimised by tuning the activation function and hidden layer size. The specific values that were tested for the model are presented in Table 6.4.

Hyperparameter	Values
`activation`	`relu`, `identity`, `logistic`, `tanh`
`hidden_layer_sizes`	$(50,)$, $(100,)$, $(50, 50)$

Table 6.4: Hyperparameter values tested for the Multilayer Perceptron model

The impact of these hyperparameters on model performance is illustrated in Figure 6.4. Among all models tested so far, the MLP achieved the highest mean accuracy across folds, outperforming the alternative architectures above.

6.4.2 Results

The results highlight that the `logistic` activation function consistently outperforms other activation functions across all hidden layer configurations. This can be attributed to its suitability for binary classification tasks, such as the one presented in this research, as it maps real-valued inputs to a range between 0 and 1 using the function:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{6.1}$$

Figure 6.4 further highlights the relationship between activation function and hidden layer size. Notably, a single hidden layer of size $(100,)$ yielded the best performance with the `logistic` and `tanh` activation functions, while performing worse with `relu` and `identity`. This suggests that both hyperparameters must be carefully tuned to the dataset to obtain optimal results. The best performing configuration was a `logistic` activation with a hidden layer of size $(100,)$, achieving a mean classification accuracy of 0.878 across the 5 folds.

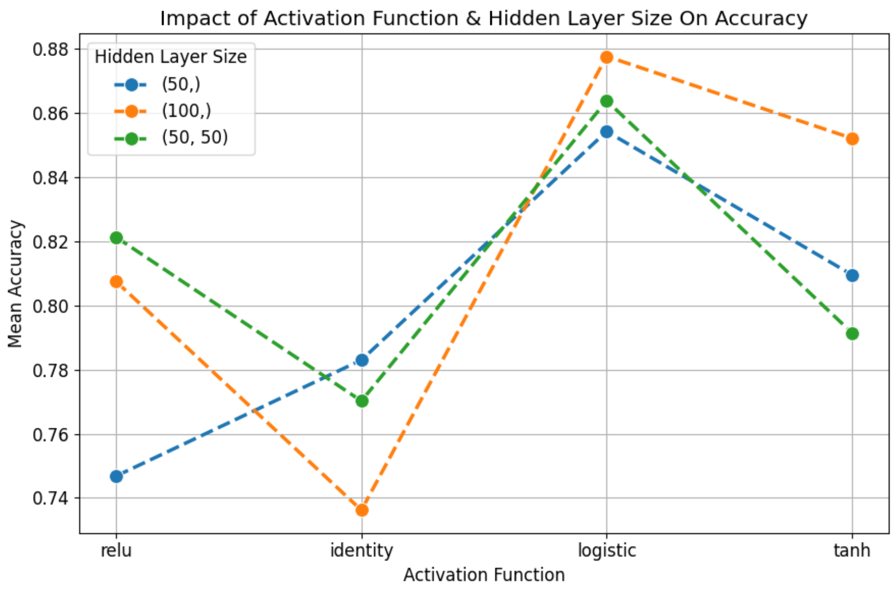


Figure 6.4: Mean classification accuracy across folds for different activation functions and hidden layer sizes.

6.5 Developing Random Forest Model

6.5.1 Hyperparameters

The key hyperparameters optimised for the Random Forest model were the number of decision trees (`n_estimators`) and the maximum depth of each individual tree (`max_depth`). These hyperparameters directly influence the model's complexity, generalisation, and computational efficiency. The values explored during tuning are presented in Table 6.5.

Hyperparameter	Values
`n_estimators`	50, 80, 110, 140, 170, 200
`max_depth`	10, 20, 30, 40, 50

Table 6.5: Hyperparameter values tested for the Random Forest model

The impact of these hyperparameters on model performance is illustrated in Figure 6.5. The Random Forest classifier consistently achieved high accuracy across all configurations examined and performed significantly better than those tested above.

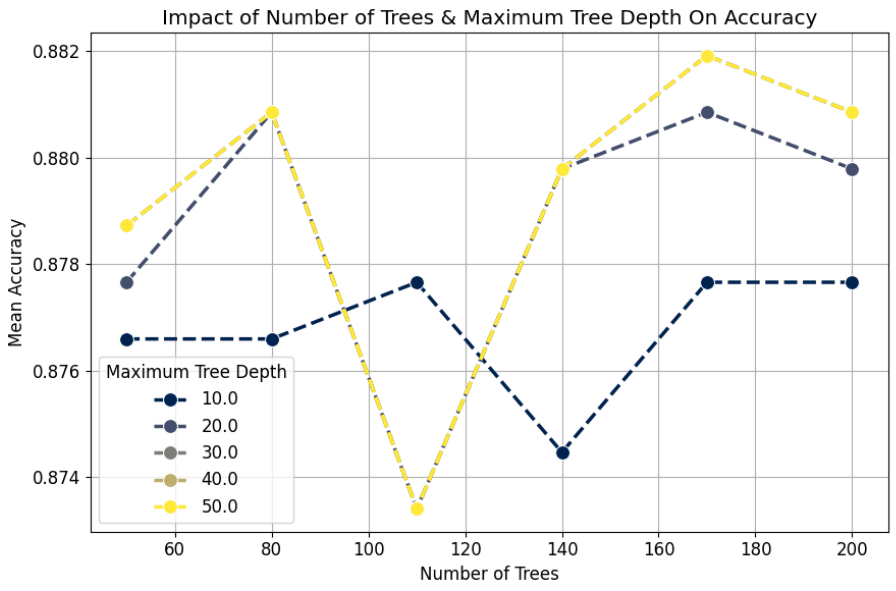


Figure 6.5: Mean classification accuracy across folds for different number of decision trees and maximum tree depth values.

6.5.2 Results

The results indicate that increasing the number of trees beyond a certain point can diminish performance. As shown in Figure 6.5, mean accuracy declined when `n_estimators` increased from 180 to 200, likely due to overfitting or increased variance. Despite this, the Random Forest model demonstrated strong robustness with accuracy tightly bounded between 0.87 and 0.89 across all tested configurations. This is particularly noteworthy given the dataset's limitations with the inclusion of imputed and oversampled data. The optimal hyperparameter configuration yielded the highest mean classification accuracy of 0.882, which was achieved with `n_estimators` set to 170 and `max_depth` values of 30, 40, and 50. The overlapping accuracy curves in Figure 6.5 suggest that performance plateaued at `max_depth` = 30, indicating that deeper trees did not provide additional improvement.

6.6 Developing Decision Tree Model

6.6.1 Hyperparameters

The final model explored in this study was the Decision Tree, with key hyperparameters tested, including the `criterion`, `max_depth`, and `max_features`, with the respective values outlined in Table 6.6.

Hyperparameter	Values
`criterion`	`gini`, `entropy`, `log_loss`
`max_depth`	5, 10, 15, 20, 25, 30
`max_features`	`auto`, `sqrt`, `log2`

Table 6.6: Hyperparameter values tested for the Decision Tree model

The performance of each hyperparameter combination was visualised in a 3D space as shown in Figure 6.6. While the Decision Tree classifier exhibited peak performance slightly below that of the MLP and SVM, it outperformed both the KNN and LR models.

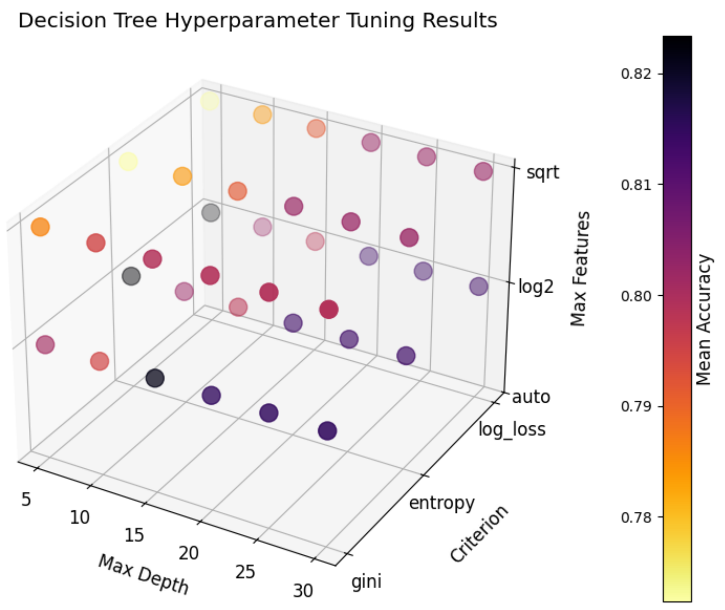


Figure 6.6: Mean classification accuracy across folds for various hyperparameter combinations.

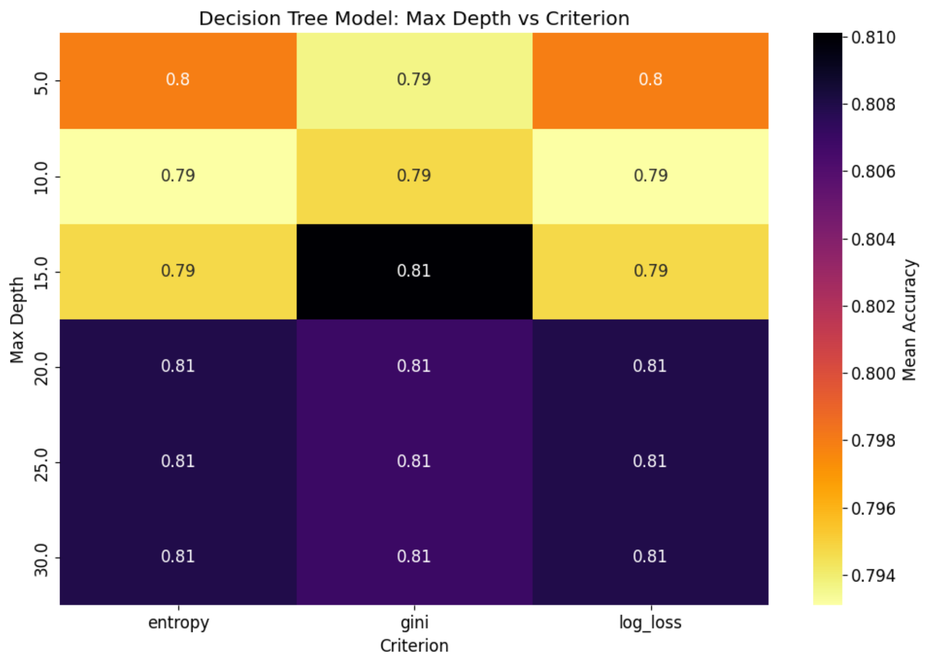


Figure 6.7: Heatmap of mean classification accuracies across folds for different combinations of maximum depth and criterion.

6.6.2 Results

The analysis in Figure 6.6 illustrates that increasing the `max_depth` of the tree generally improves classification accuracy as the model becomes more complex. However, the highest accuracy of 0.823 was achieved with a `max_depth` of 5, `max_features` set to `log2`, and `criterion` as `entropy`. The entropy criterion, defined as:

$$-\sum_{i=1}^{n} p_i \log_2(p_i) \tag{6.2}$$

where p_i represents the probability of selecting a sample from class i quantifies the uncertainty or impurity of a dataset. This accounts for the full distribution of class probabilities, making it particularly sensitive to information in minority classes, an important factor in this study given the dataset's skew towards participants with stable MCI over the two-year period. The ability to capture subtle variations in the data is crucial for generalisation and improving model performance in the presence of such class imbalance.

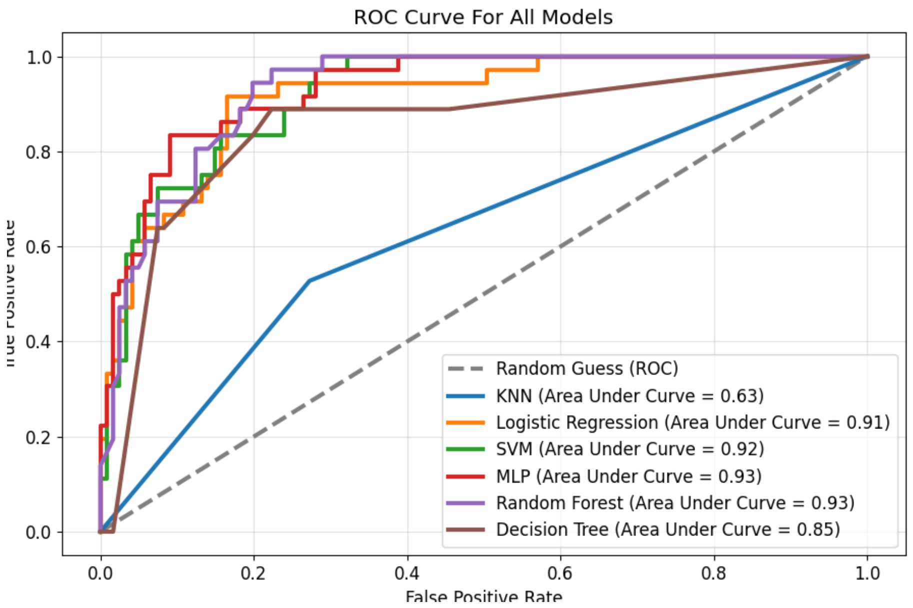


Figure 6.8: ROC curve for each of the models with their optimal hyperparameter configuration.

Figure 6.8 illustrates the predictive performance of the all the models, with most demonstrating strong classification ability. However, KNN achieved an AUC of 0.63 which indicates limited but non-negligible predictive power, performing only marginally better than chance. In contrast, MLP and Random Forest exhibited exceptional performance with both attaining an AUC of 0.93. These results suggest that these models effectively discriminated between classes and generalised well as seen by this performance on the unseen dataset.

6.7 Developing Ensemble Model

Following hyperparameter tuning of the six individual models, we proceeded with constructing the ensemble model to enhance classification accuracy. Ensemble learning mitigates model-specific errors by aggregating predictions which improves generalisation and reduces overfitting. To determine the optimal ensemble configuration, we evaluated all possible permutations of the six models, each using its tuned hyperparameters, employing both stacking and voting classifiers. Each ensemble was trained on the oversampled dataset and evaluated on the unseen test set, with the model yielding the highest classification accuracy selected as the final classifier.

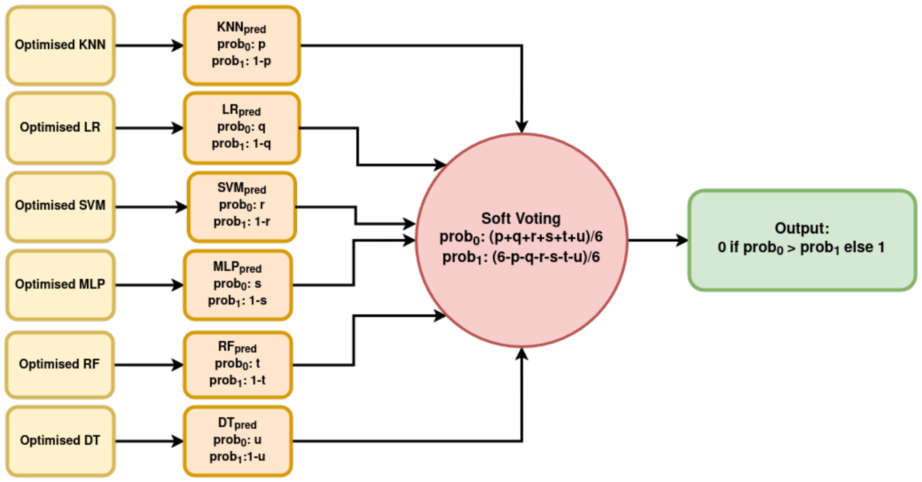

Optimised KNN
Optimised LR
Optimised SVM
Optimised MLP
Optimised RF
Optimised DT
KNN_{pred} $prob_0$: p $prob_1$: 1-p
LR_{pred} $prob_0$: q $prob_1$: 1-q
SVM_{pred} $prob_0$: r $prob_1$: 1-r
MLP_{pred} $prob_0$: s $prob_1$: 1-s
RF_{pred} $prob_0$: t $prob_1$: 1-t
DT_{pred} $prob_0$: u $prob_1$:1-u
Soft Voting
$prob_0$: (p+q+r+s+t+u)/6
$prob_1$: (6-p-q-r-s-t-u)/6
Output:
0 if $prob_0$ > $prob_1$ else 1

Figure 6.9: A soft voting ensemble model built using all 6 models.

Voting-based ensemble methods combine predictions from multiple models. In hard voting, the majority class prediction is selected, whereas in soft voting, the predicted class probabilities are averaged and the final class determined by the highest probability. We adopted the soft voting approach in this study, with its structure illustrated in Figure 6.9. In contrast, stacking employs a hierarchical approach, where the predictions from the base models serve as input features for the **meta-model**. In our implementation, logistic regression was chosen as the final estimator due to its probabilistic output, interpretability, and effectiveness in integrating diverse model predictions. The stacking architecture is depicted in Figure 6.10 from the perspective of all 6 base models combination.

Table 6.7 presents the classification accuracy of the different ensemble configurations on the test set using both a stacking and voting approach. The highest accuracy of 0.873 was achieved using a stacking classifier comprising a Decision Tree model and a LR model, with LR also chosen as the **meta-model**.

Table 6.7: The classification accuracy of all ensemble model combinations and classification techniques

Model	Stacking Accuracy	(Soft) Voting Accuracy
KNN	0.682	0.682
LR	0.834	0.834
SVM	0.834	0.834
MLP	0.834	0.809
RF	0.841	0.841

Model	Stacking Accuracy	(Soft) Voting Accuracy
DT	0.803	0.803
KNN, LR	0.815	0.682
KNN, SVM	0.803	0.682
KNN, MLP	0.803	0.682
KNN, RF	0.847	0.682
KNN, DT	0.809	0.764
LR, SVM	0.834	0.834
LR, MLP	0.828	0.822
LR, RF	0.841	0.834
LR, DT	0.873	0.847
SVM, MLP	0.828	0.815
SVM, RF	0.841	0.828
SVM, DT	0.815	0.822
MLP, RF	0.860	0.860
MLP, DT	0.847	0.822
RF, DT	0.841	0.828
KNN, LR, SVM	0.803	0.815
KNN, LR, MLP	0.803	0.822
KNN, LR, RF	0.847	0.828
KNN, LR, DT	0.815	0.803
KNN, SVM, MLP	0.796	0.809
KNN, SVM, RF	0.847	0.815
KNN, SVM, DT	0.796	0.803
KNN, MLP, RF	0.834	0.815
KNN, MLP, DT	0.790	0.803
KNN, RF, DT	0.847	0.803
LR, SVM, MLP	0.828	0.821
LR, SVM, RF	0.841	0.834
LR, SVM, DT	0.854	0.860
LR, MLP, RF	0.860	0.834
LR, MLP, DT	0.847	0.847
LR, RF, DT	0.841	0.847
SVM, MLP, RF	0.872	0.834
SVM, MLP, DT	0.847	0.828
SVM, RF, DT	0.841	0.828
MLP, RF, DT	0.860	0.815
KNN, LR, SVM, MLP	0.803	0.828

Model	Stacking Accuracy	(Soft) Voting Accuracy
KNN, LR, SVM, RF	0.847	0.822
KNN, LR, SVM, DT	0.803	0.809
KNN, LR, MLP, RF	0.834	0.834
KNN, LR, MLP, DT	0.790	0.803
KNN, LR, RF, DT	0.847	0.822
KNN, SVM, MLP, RF	0.841	0.828
KNN, SVM, MLP, DT	0.796	0.803
KNN, SVM, RF, DT	0.847	0.822
KNN, MLP, RF, DT	0.834	0.822
LR, SVM, MLP, RF	0.860	0.834
LR, SVM, MLP, DT	0.847	0.841
LR, SVM, RF, DT	0.841	0.841
LR, MLP, RF, DT	0.860	0.854
SVM, MLP, RF, DT	0.872	0.841
KNN, LR, SVM, MLP, RF	0.841	0.834
KNN, LR, SVM, MLP, DT	0.796	0.809
KNN, LR, SVM, RF, DT	0.847	0.822
KNN, LR, MLP, RF, DT	0.828	0.822
KNN, SVM, MLP, RF, DT	0.841	0.822
LR, SVM, MLP, RF, DT	0.860	0.841
KNN, LR, SVM, MLP, RF, DT	0.841	0.822

6.8 Model Construction Results

The model that achieved the highest accuracy was a stacking ensemble comprising two base models, each trained independently, with LR serving as the **meta-model**. The base models were Logistic Regression and a Decision Tree classifier. The LR model was optimised with a regularisation parameter $C = 1$ using the `newton-cg` solver. For the Decision Tree, the optimal hyperparameters were an `entropy` criterion, a maximum depth of 5, and feature selection based on `log2`. The final estimator had the same hyperparameter setting as the base LR model. The structure of this ensemble is visualised in Figure 6.11.

This model demonstrated strong predictive performance, misclassifying only 20 out of 157 test samples (12.7%), as shown in Figure 6.12. Furthermore, the ROC curve (Figure 6.13) yielded an AUC score of 0.92, indicating the model has strong predictive power and robust generalisability in distinguishing between both classes.

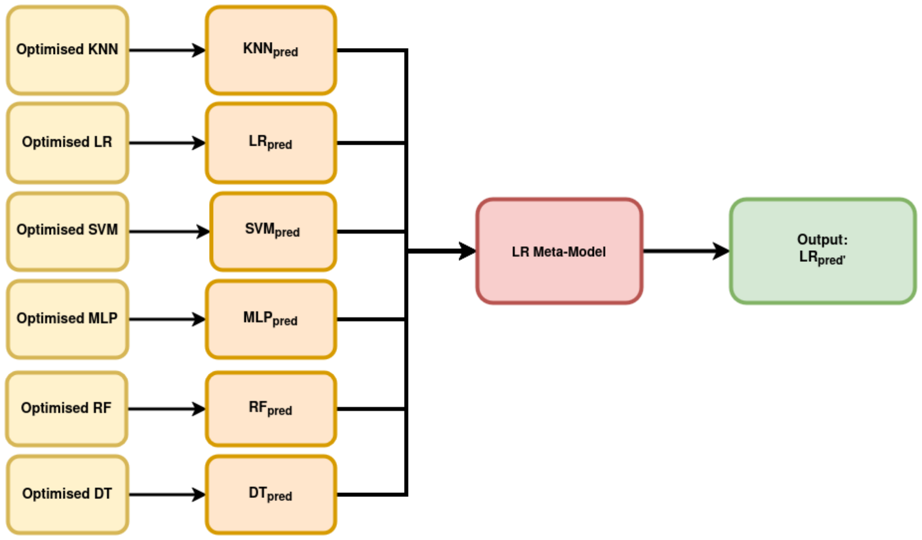


Figure 6.10: A stacking ensemble model built using all 6 models.

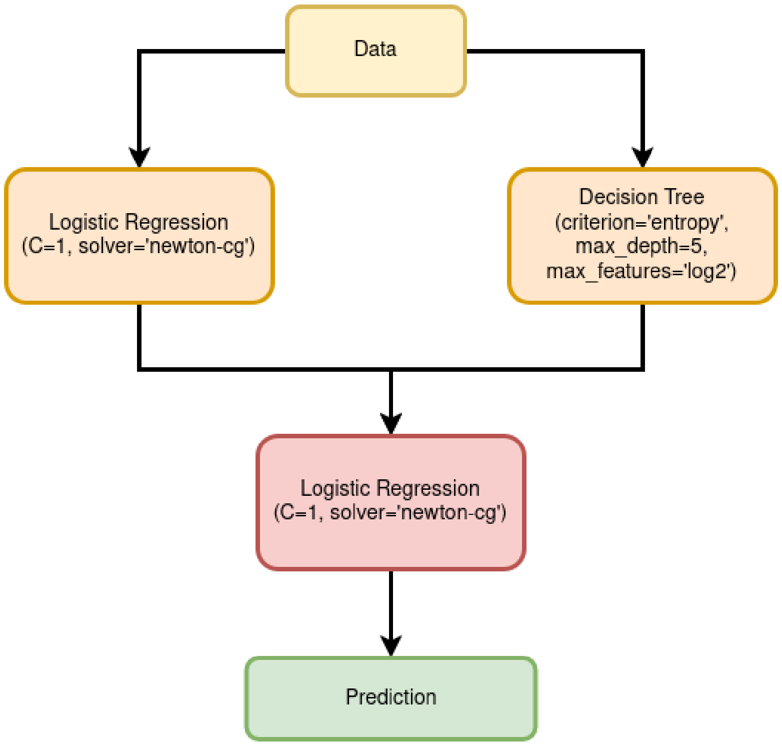

Data
Logistic Regression (C=1, solver='newton-cg')
Decision Tree (criterion='entropy', max_depth=5, max_features='log2')
Logistic Regression (C=1, solver='newton-cg')
Prediction

Figure 6.11: The structure of the final ensemble model.

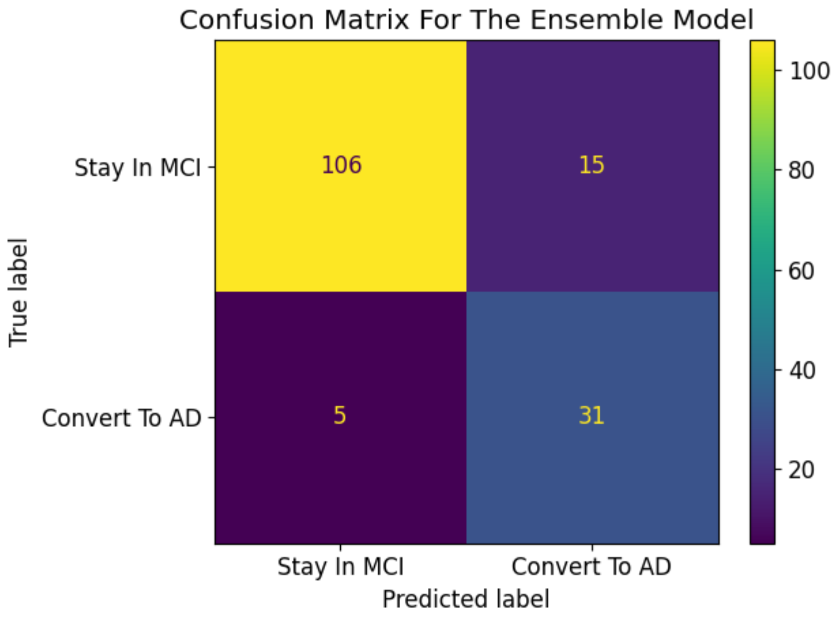


Figure 6.12: The confusion matrix of the best performing ensemble model.

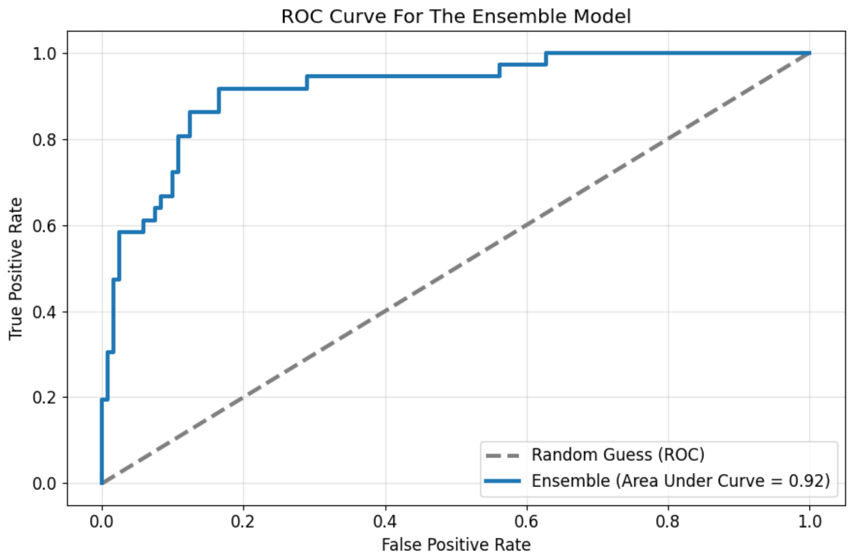


Figure 6.13: The ROC curve of the best performing ensemble model.

Chapter 7

Results and Discussions

7.1 Model Evaluation Results

To evaluate the final ensemble model obtained after training and optimisation, a stratified K-fold cross-validation approach was employed. The dataset was partitioned into training and testing subsets, where the ensemble model was fitted to the training set and then used to predict the corresponding testing set. This process was iterated across all the datasets tabulated in Table 7.1.

The ensemble model demonstrated consistent performance across all the datasets mentioned in Table 7.1, with minimal variation in mean and standard deviations. Table 7.2 summarises the key performance metrics of the model on each dataset. Among the evaluated configurations, the model trained on the base dataset with plasma biomarkers achieved the best overall performance, having a superior accuracy, precision, recall and F1-score, outperforming all other dataset permutations.

Datasets	Included Features
Base	Age, Gender, Education, Marital Status, Racial Category, APOE $\epsilon 4$, MMSE, RAVLT-I, RAVLT-L, RAVLT-F, RAVLT-PF, ADAS11, ADAS13, ADASQ4, mPACCdigit, mPACCtrailsB, FAQ, Diagnosis (EMCI/LMCI)
Base + CSF	Base + ABETA, PTAU, TAU
Base + Plasma	Base + Aβ42/40-Plasma, NFL-Plasma, PTAU-Plasma
Full Dataset	Base + CSF + Plasma

Table 7.1: The different datasets used for ensemble model evaluation

Dataset	Accuracy	Precision	Recall	F1 Score	AUC
Base	0.81 ± 0.02	0.68 ± 0.06	0.46 ± 0.06	0.55 ± 0.05	0.88 ± 0.01
Base + CSF	0.82 ± 0.02	0.69 ± 0.10	0.52 ± 0.06	0.59 ± 0.05	0.88 ± 0.02
Base + Plasma	0.83 ± 0.02	0.71 ± 0.06	0.52 ± 0.07	0.59 ± 0.05	0.88 ± 0.02
Full Dataset	0.82 ± 0.02	0.70 ± 0.04	0.46 ± 0.10	0.55 ± 0.06	0.88 ± 0.02

Table 7.2: Mean and standard deviation of each metric in relation to the performance of the ensemble model on each dataset

7.1.1 Full Dataset

In comparison, the full dataset demonstrated a marginally lower F1-score, recall, and accuracy. Contrarily, the model achieved a consistent AUC score with the results of the other datasets as illustrated in Figure 7.1. However, this model achieved strong predictive performance with a high accuracy of 0.82, suggesting it performs well in forecasting conversion in MCI patients. Additionally, this model yielded a low standard deviation in accuracy and precision, indicating greater robustness with the predictions more tightly distributed around the mean. Such clustering suggests a higher reliability in the output of the model which is particularly important in clinical applications where consistency in predictions is vital.

7.1.2 Base + CSF Dataset

The inclusion of the Cerebrospinal Fluid (CSF) biomarker into the base dataset resulted in mixed performance outcomes. While there was a marginal improvement in all of the performance metrics, the standard deviation in precision also increased from 0.06 to 0.10. A potential cause for this may be due to the high proportion of missing values for key CSF biomarkers (ABETA, PTAU, and TAU) in the raw dataset. Consequently, majority of these values used to train the model were synthetically imputed using ADASYN, as explained in Section 5.2, which may have introduced a potential source of error that likely compromised the model's predictive performance.

7.1.3 Misclassification

Furthermore, the highest misclassification rates across all dataset combinations occurred in the prediction of individuals who converted from MCI to AD in reality but were predicted as stable MCI after the 2-year period, with $\approx 53.8\%$ of samples misclassified, as visualised in the confusion matrix in Figure 7.2. This discrepancy is likely attributable to the highly skewed class distribution within the datasets, with the under-representation of individuals who convert to AD over the two years. This potentially hindered the model's ability to learn well the underlying patterns associated with MCI to AD progression, hence emphasising the need for more advanced techniques to address this class imbalance in future research.

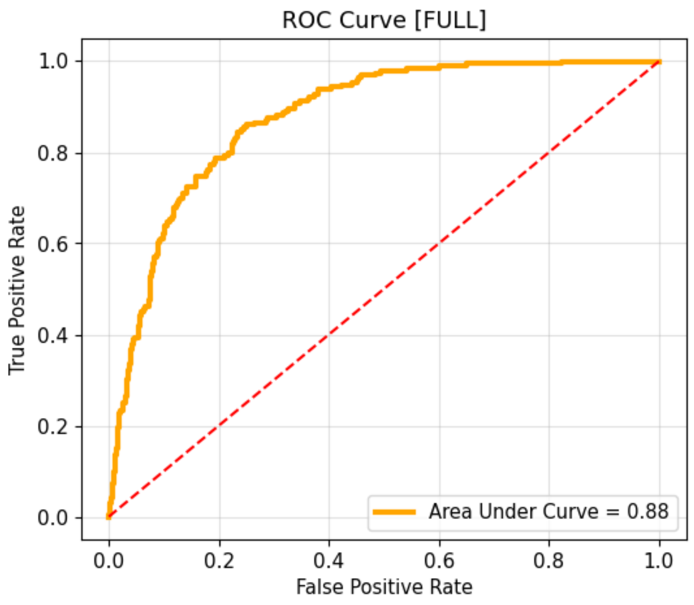


Figure 7.1: The ROC curve of the ensemble model when tested on the full dataset.

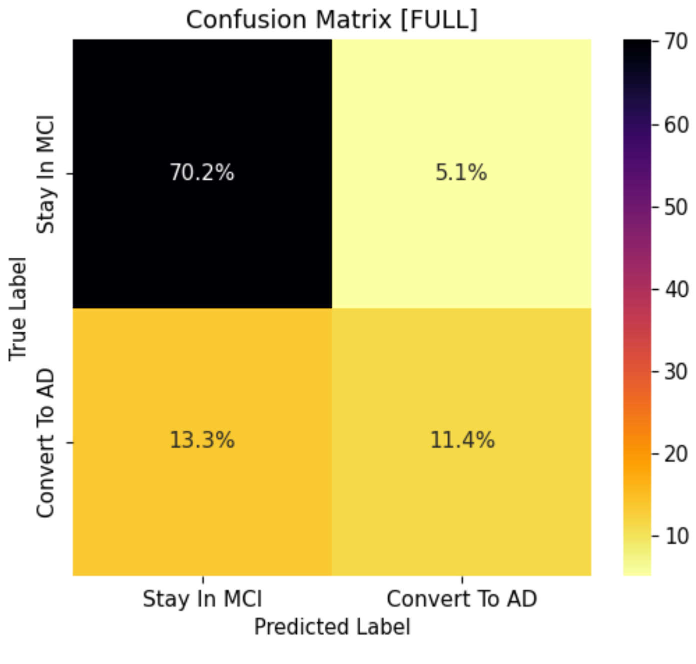


Figure 7.2: The confusion matrix for the prediction of MCI to AD conversion by the ensemble model on the full dataset.

7.2 Feature Importance Analysis

7.2.1 Mutual Information-Based Feature Importance

The evaluation phase of the ensemble model was concluded with an analysis of feature importance in determining whether participants transitioned from MCI to AD or remained in stable MCI over the two-year period. Mutual information-based feature selection was used to assess the dependency between each feature and the model output. This method quantifies the amount of information a feature provides about the target variable by capturing both linear and non-linear relationships. Specifically, it measures the reduction in uncertainty about the target variable given the knowledge of a particular feature, thereby identifying the features that are most informative. It is computed using the mutual information between a feature X and the target variable Y, defined as [40]:

$$I(X;Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) \log \frac{p(x,y)}{p(x)p(y)} \tag{7.1}$$

where $p(x,y)$ is the joint probability distribution of the feature and the target, $p(x)$ and $p(y)$ the marginal probability distributions of the feature and the target respectively. The analysis yielded compelling results, with baseline neuropsychological and cognitive assessment features such as Functional Activities Questionnaire (FAQ), ADAS13, mPACCtrailsB and RAVLT Immediate emerging as the most informative predictors of conversion to AD, as illustrated in Figure 7.3.

This finding underscores the crucial role that cognitive and neuropsychological tests play in the early identification and monitoring of disease progression. Alternatively, demographic variables such as age, gender, racial category and education level exhibited the lowest mutual information scores of 0, suggesting their limited contribution to the predictive performance of the ensemble model, implying that although demographic factors may assist in offering context, they are insufficient as stand-alone predictors for AD progression.

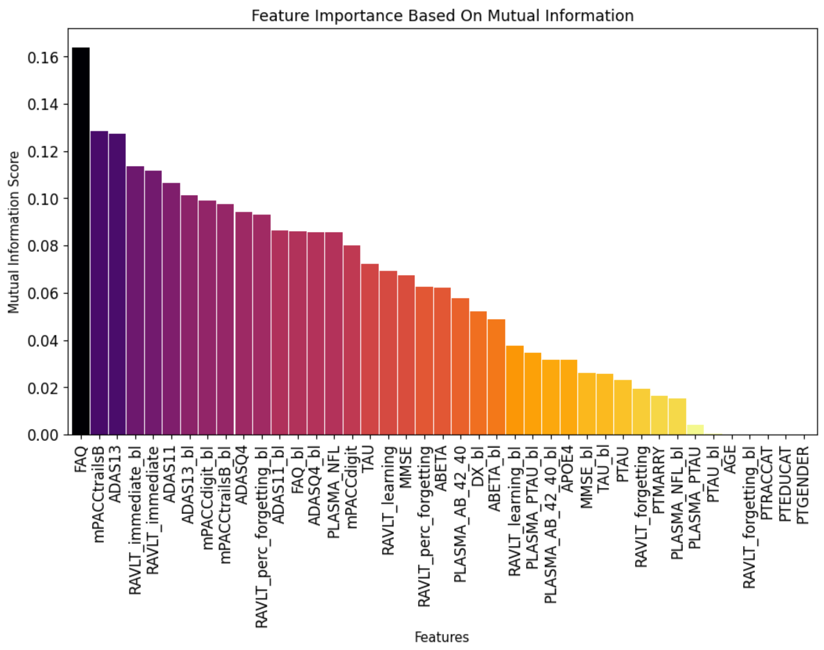


Figure 7.3: Mutual information-based feature importance analysis of the ensemble model on the full dataset.

7.2.2 Permutation Feature Importance

Another method explored for feature evaluation was permutation feature importance, an approach that quantifies the decrease in a model's performance when the values of a particular feature are randomly permutated. This method disrupts the relationship between the feature and the target variable, hence providing insight into a feature's contribution to the model's predictive performance. This is computed using the following equation:

$$I_j = S_{original} - S^j_{permuted} \tag{7.2}$$

where $S_{original}$ is the model's performance (accuracy) on the original data, $S^j_{permuted}$ is the performance of the model after randomly shuffling the values of the j-th feature, and I_j denoting the importance of feature j. Similar to the results from mutual information-based feature selection, this analysis also highlighted the vital role that cognitive and neuropsychological assessments hold in determining the model output with features such as FAQ, mPACCtrailsB, RAVLT Immediate, and ADAS13 again emerging as some of the most influential. Additionally, some plasma and CSF biomarkers such as NfL and TAU demonstrated significant importance, as shown in Figure 7.4.

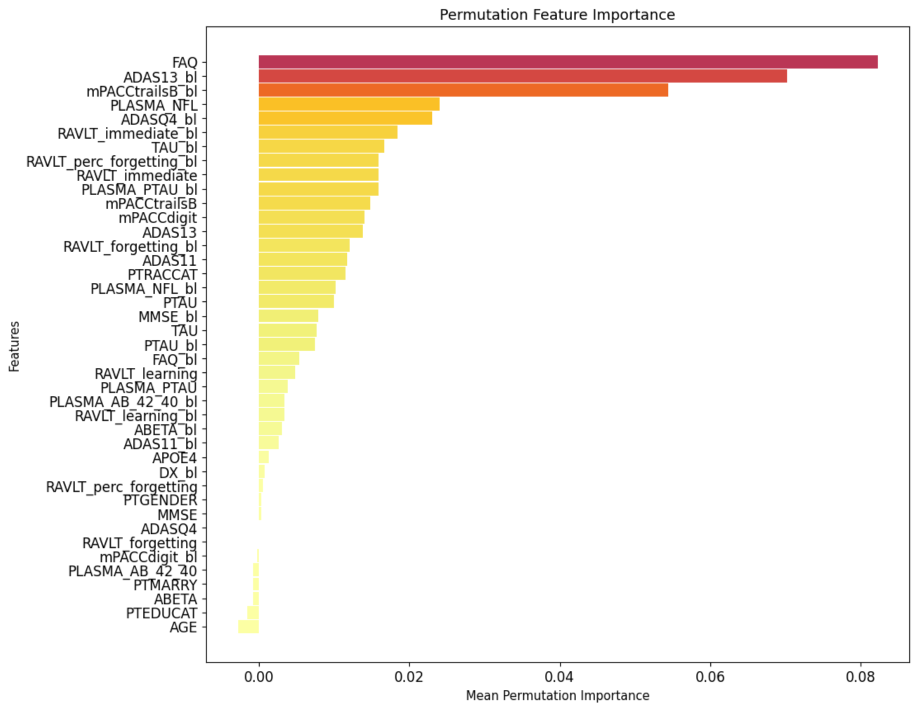


Figure 7.4: Mean permutation importance score of each feature in the dataset.

However, marital status, education level and gender exhibited negative mean permutation feature scores, implying that shuffling these features' values led to an improvement in model performance rather than a decline. This result could arise from many factors such as these features introducing noise or irrelevant information to the model, the model overfitting to specific patterns in the training data associated with these features, or simply reflect random variations in model performance or noise within the evaluation metric as the negative scores were marginal and close to 0.

7.3 Discussion

7.3.1 Discussion of Results

The goal of the research was to assess the predictive capability of plasma and CSF biomarkers in conjunction with socio-demographic and neuropsychological tests as a cost-effective alternative to predict MCI to AD progression within 2 years of the base diagnosis, using data collected from the ADNI study. This was achieved with an accuracy of 0.82 and precision of 0.70, suggesting strong performance given the limited data and relatively short time-frame examined.

One of the primary challenges in constructing AD prediction models is the significant class imbalance between participants who remain in stable MCI and those who progress to AD. This issue is particularly pronounced in studies with short observation windows such as the two-year period explored in this study. While the majority of older adults with MCI develop dementia within five years [41], the shorter period we studied contributed to a skewed dataset with most participants remaining in the stable MCI category. To address this imbalance, we experimented with several oversampling methods including random oversampling, Synthetic Minority Oversampling Technique (SMOTE), and Adaptive Synthetic Sampling (ADASYN). Among these, ADASYN produced synthetic data that most closely aligned with the original dataset, making it the preferred method for balancing the classes. Another challenge encountered was the large proportion of missing data, particularly for CSF biomarkers. Missing data necessitated the use of imputation methods to generate values to fit into the dataset. We evaluated two imputation strategies, K-Nearest Neighbours and Multiple Imputation by Chained Equations (MICE) Forest, through rigorous testing on a small sample set, and used the optimal results on the complete dataset. This experiment revealed that MICE Forest significantly outperformed KNN imputation for our dataset as it achieved a lower MAE and MSE.

We evaluated the performance of various ML classification models on the imputed and over-sampled dataset, ultimately adopting an ensemble model approach after exhaustively testing all possible model permutations. This ensemble model demonstrated robust predictive capabilities across all the datasets. Notably, feature importance analysis revealed that even simple, non-invasive data such as clinical evaluations and neuropsychological assessments contribute substantially to the model's predictive performance, which is an encouraging finding as such data is inexpensive and straightforward to collect in clinical settings.

7.3.2 Libraries Used For Implementation

The `Sklearn` library in Python was particularly useful for the development of the models, `GridSearchCV` for hyperparameter tuning, `pandas` and `matplotlib` for data pre-processing and visualisation respectively.

7.3.3 Evaluation of Objectives

Below is a summary of each of the objectives set out at the start of the project and how they were achieved:

1. **Data Collection and Preprocessing:** All the relevant data was retrieved from the ADNI repository and systematically collated. Preprocessing techniques such as imputation to handle missing values and oversampling to address class imbalances were applied, as well as filtering the

dataset to retain only the most relevant features within the specified time window of two-years.

2. **Model Implementation and Training:** A comprehensive evaluation of multiple classification algorithms including KNN, MLP, SVM, LR, Decision Tree and Random Forest was performed to determine their effectiveness in predicting disease progression. Hyperparameter tuning was performed for each model to identify the optimal configuration that maximised predictive accuracy. Finally, all the permutations of each of the models with their optimal hyperparameters were explored to construct an ensemble model that enhances robustness and overall performance.

3. **Ensemble Model Evaluation:** The predictive capability of the ensemble model was tested using different dataset combinations. This approach enabled an in-depth assessment of the contribution of individual biomarker categories, providing an insight into their influence on model performance.

4. **Input Feature Evaluation:** A detailed feature importance analysis was conducted to identify the most influential predictors in determining the model's outcome, and features with minimal impact on classification performance were also identified.

7.3.4 Reflection

Overall, the planning, development, and organisation of the project went very well, with strong time management and a well-defined vision of the required tasks which allowed for a systematic workflow. The bulk of the development phase was spent on data preprocessing, largely due to the scale of the ADNI dataset. Significant effort was required to extract the plasma biomarker data and integrate it into the ADNIMERGE dataset whilst also addressing key challenges such as class imbalance and missing data, both of which have critical implications on the final model to be generated.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

In this study, we demonstrated that ML models leveraging diverse feature sets can effectively predict MCI to AD conversion. We highlighted the potential of achieving strong predictive performance using CSF and blood-based biomarkers alone without the need of costly imaging modalities such as MRI and PET scans. This approach is particularly advantageous given the resource constraints prevalent in many healthcare systems, especially in developing nations where access to imaging biomarkers is financially prohibitive and challenging. By demonstrating the utility of CSF and BBMs, this project provides evidence to support their viability as affordable and less intensive alternatives for predicting AD progression.

The ensemble model developed in this study consisted of a stacking classifier comprising a Logistic Regression model and a Decision Tree classifier. The input dataset included 41 features spanning socio-demographic variables, clinical assessments, CSF and blood-based biomarkers. The dataset was imputed to fill in missing values using MICE Forest and over-sampled using ADASYN to address the class imbalance. The model's performance was evaluated using 4-fold stratified cross-validation, and demonstrated strong predictive accuracy across all datasets. While the model achieved good precision, the recall and F1 scores were comparatively moderate which highlights areas for potential improvement in future work.

Despite producing an ensemble model that achieved high predictive accuracy, it is essential to acknowledge the limitations in this research, especially in regards to the dataset. The ADNI dataset, while comprehensive, is not a population-based study hence its demographic and clinical characteristics may not be reflective of the heterogeneity observed in a larger population. Therefore, the generalisability of the model must be rigorously tested on diverse cohorts to ensure robust clinical applicability.

8.2 Future Work

A key target of future research should be to address the significant class imbalance observed in this study. To improve the predictive capability of plasma and CSF biomarkers, future studies should consider extending the observation period to 4–6 years as a longer time-frame would likely yield a more balanced dataset which would reduce the skewness observed in this study. Additionally, the feature importance evaluation employed in this research could be refined by eliminating the features that had negligible influence on the target variable and instead prioritise only the most informative features which may lead to a more efficient and performant model. We could also explore the use of more classification models in the ensemble to analyse the impact it has on performance, or explore deep learning approaches as they may offer superior performance by capturing complex non-linear relationships between features. However, this would also increase computational complexity and demand an increase in resources, hence the need to carefully consider the trade-off between complexity and performance.

Bibliography

[1] E. Bomasang-Layno and R. Bronsther. "Diagnosis and Treatment of Alzheimer's Disease: An Update". In: *Delaware Journal of Public Health* 7.4 (Sept. 2021), pp. 74–85. DOI: 10.32481/djph.2021.09.009.

[2] "2023 Alzheimer's disease facts and figures". In: *Alzheimer's Dementia* 19.4 (Apr. 2023). Epub 2023 Mar 14, pp. 1598–1695. DOI: 10.1002/alz.13016.

[3] F Gonzalez-Ortiz et al. "Plasma phospho-tau in Alzheimer's disease: towards diagnostic and therapeutic trial applications". In: *Molecular Neurodegeneration* 18.1 (Mar. 2023), p. 18. DOI: 10.1186/s13024-023-00605-8.

[4] Michael P. Murphy and III LeVine Harry. "Alzheimer's disease and the amyloid-beta peptide". In: *Journal of Alzheimer's Disease* 19.1 (2010), pp. 311–323. DOI: 10.3233/JAD-2010-1221.

[5] M. V. F. Silva, C. d. M. G. Loures, L. C. V. Alves, et al. "Alzheimer's disease: risk factors and potentially protective measures". In: *Journal of Biomedical Science* 26.1 (2019), p. 33. DOI: 10.1186/s12929-019-0524-y. URL: https://doi.org/10.1186/s12929-019-0524-y.

[6] Frank Jessen et al. "The characterisation of subjective cognitive decline". In: *Lancet Neurology* 19.3 (Mar. 2020), pp. 271–278. DOI: 10.1016/S1474-4422(19)30368-0. URL: https://doi.org/10.1016/S1474-4422(19)30368-0.

[7] P. Chen, H. Cai, W. Bai, et al. "Global prevalence of mild cognitive impairment among older adults living in nursing homes: a meta-analysis and systematic review of epidemiological surveys". In: *Translational Psychiatry* 13 (2023), p. 88. DOI: 10.1038/s41398-023-02361-1. URL: https://doi.org/10.1038/s41398-023-02361-1.

[8] Zana Breijyeh and Rafik Karaman. "Comprehensive Review on Alzheimer's Disease: Causes and Treatment". In: *Molecules* 25.24 (Dec. 2020), p. 5789. DOI: 10.3390/molecules25245789.

[9] Ronald Petersen and Michael W. Weiner. *ADNI-1 Protocol.* Visited: 2025-01-25. Sept. 2008. URL: https://adni.loni.usc.edu/wp-content/themes/freshnews-dev-v2/documents/clinical/ADNI-1_Protocol.pdf.

[10] Giovanni B. Frisoni et al. "The clinical use of structural MRI in Alzheimer disease". In: *Nature Reviews Neurology* 6 (2010), pp. 67–77. DOI: 10.1038/nrneurol.2009.215. URL: https://doi.org/10.1038/nrneurol.2009.215.

[11] Agneta Nordberg et al. "The use of PET in Alzheimer disease". In: *Nature Reviews Neurology* 6.2 (Feb. 2010), pp. 78–87. DOI: 10.1038/nrneurol.2009.217.

[12] Kaj Blennow and Henrik Zetterberg. "Cerebrospinal Fluid Biomarkers for Alzheimer's Disease". In: *Journal Name* 1 (Jan. 2009), pp. 413–417.

[13] M. V. Pais, O. V. Forlenza, and B. S. Diniz. "Plasma Biomarkers of Alzheimer's Disease: A Review of Available Assays, Recent Developments, and Implications for Clinical Practice". In: *Journal of Alzheimer's Disease Reports* 7.1 (May 2023), pp. 355–380. DOI: 10.3233/ADR-230029.

[14] J. A. Nichols, H. W. Herbert Chan, and M. A. B. Baker. "Machine learning: applications of artificial intelligence to imaging and diagnosis". In: *Biophysical Reviews* 11.1 (Feb. 2019), pp. 111–118. DOI: 10.1007/s12551-018-0449-9. URL: https://doi.org/10.1007/s12551-018-0449-9.

[15] IBM. *Machine Learning - IBM Think.* Accessed: 2025-02-02. 2025. URL: https://www.ibm.com/think/topics/machine-learning.

[16] Syed Tahir Hussain Rizvi et al. "Analysis of Machine Learning Based Imputation of Missing Data". In: *Cybernetics and Systems* (2023), pp. 1–15. DOI: 10.1080/01969722.2023.2247257. eprint: https://doi.org/10.1080/01969722.2023.2247257. URL: https://doi.org/10.1080/01969722.2023.2247257.

[17] Z. Zhang. "Introduction to machine learning: k-nearest neighbors". In: *Annals of Translational Medicine* 4.11 (June 2016), p. 218. DOI: 10.21037/atm.2016.03.37.

[18] S. Huang et al. "Applications of Support Vector Machine (SVM) Learning in Cancer Genomics". In: *Cancer Genomics Proteomics* 15.1 (Jan. 2018), pp. 41–51. DOI: 10.21873/cgp.20063.

[19] R.Y. Choi et al. "Introduction to Machine Learning, Neural Networks, and Deep Learning". In: *Translational Vision Science Technology* 9.2 (Feb. 2020), p. 14. DOI: 10.1167/tvst.9.2.14.

[20] ScienceDirect Topics. *Multilayer Perceptron.* Accessed: 2025-02-25. 2025. URL: https://www.sciencedirect.com/topics/computer-science/multilayer-perceptron.

[21] Leo Breiman. "Random Forests". In: *Machine Learning* 45.1 (2001), pp. 5–32. ISSN: 1573-0565. DOI: 10.1023/A:1010933404324. URL: https://doi.org/10.1023/A:1010933404324.

[22] Lior Rokach and Oded Maimon. "Decision Trees". In: *Data Mining and Knowledge Discovery Handbook*. Ed. by Oded Maimon and Lior Rokach. Boston, MA: Springer US, 2005, pp. 165–192. ISBN: 978-0-387-25465-4. DOI: 10.1007/0-387-25465-X_9.

[23] Deniz Gunay. *Random Forest*. Accessed: 2025-03-27. 2025. URL: https://medium.com/@denizgunay/random-forest-af5bde5d7e1e.

[24] Bhargav T. Nallapu et al. "Plasma Biomarkers as Predictors of Progression to Dementia in Individuals with Mild Cognitive Impairment". In: *Journal of Alzheimer's Disease* 98.1 (2024). PMID: 38393899, pp. 231–246. DOI: 10.3233/JAD-230620. eprint: https://journals.sagepub.com/doi/pdf/10.3233/JAD-230620. URL: https://journals.sagepub.com/doi/abs/10.3233/JAD-230620.

[25] Thomas K. Karikari, Anne L. Benedet, Nicholas J. Ashton, et al. "Diagnostic performance and prediction of clinical progression of plasma phospho-tau181 in the Alzheimer's Disease Neuroimaging Initiative". In: *Molecular Psychiatry* 26 (2021), pp. 429–442. DOI: 10.1038/s41380-020-00923-z. URL: https://doi.org/10.1038/s41380-020-00923-z.

[26] G. Chandrasekaran, S. X. Xie, and Alzheimer's Disease Neuroimaging Initiative. "Improving Regression Analysis with Imputation in a Longitudinal Study of Alzheimer's Disease". In: *Journal of Alzheimer's Disease* 99.1 (2024), pp. 263–277. DOI: 10.3233/JAD-231047.

[27] Benoît Lehallier et al. "Combined Plasma and Cerebrospinal Fluid Signature for the Prediction of Midterm Progression From Mild Cognitive Impairment to Alzheimer Disease". In: *JAMA Neurology* 73.2 (Feb. 2016). Epub 2015 Dec 14, pp. 203–212. DOI: 10.1001/jamaneurol.2015.3135.

[28] R. L. H. Handels et al. "Predicting progression to dementia in persons with mild cognitive impairment using cerebrospinal fluid markers". In: *Alzheimer's & Dementia* 13.8 (Aug. 2017). Epub 2017 Feb 17, pp. 903–912. ISSN: 1552-5260. DOI: 10.1016/j.jalz.2016.12.015.

[29] Paolo Maria Rossini, Fabio Miraglia, and Fabrizio Vecchio. "Early dementia diagnosis, MCI-to-dementia risk prediction, and the role of machine learning methods for feature extraction from integrated biomarkers, in particular for EEG signal analysis". In: *Alzheimer's & Dementia* 18.12 (Dec. 2022). Epub 2022 Apr 7, pp. 2699–2706. ISSN: 1552-5260. DOI: 10.1002/alz.12645.

[30] Michael W. Weiner and Dallas P. Veitch. "Introduction to special issue: Overview of Alzheimer's Disease Neuroimaging Initiative". In: *Alzheimer's & Dementia* 11.7 (July 2015), pp. 730–733. DOI: 10.1016/j.jalz.2015.05.007.

[31] Ronald C. Petersen et al. "Alzheimer's Disease Neuroimaging Initiative (ADNI): clinical characterization". In: *Neurology* 74.3 (Jan. 2010). Epub 2009 Dec 30, pp. 201–209. ISSN: 1526-632X. DOI: 10.1212/WNL.0b013e3181cb3e25.

[32] David Andrés González et al. "Comprehensive Evaluation of the Functional Activities Questionnaire (FAQ) and Its Reliability and Validity". In: *Assessment* 29.4 (2022). PMID: 33543638, pp. 748–763. DOI: 10.1177/1073191121991215. eprint: https://doi.org/10.1177/1073191121991215. URL: https://doi.org/10.1177/1073191121991215.

[33] Ingrid Arevalo-Rodriguez et al. "Mini-Mental State Examination (MMSE) for the early detection of dementia in people with mild cognitive impairment (MCI)". In: *Cochrane Database of Systematic Reviews* 7.7 (July 2021), p. CD010783. DOI: 10.1002/14651858.CD010783.pub3.

[34] Elaheh Moradi et al. "Rey's Auditory Verbal Learning Test scores can be predicted from whole brain MRI in Alzheimer's disease". In: *NeuroImage: Clinical* 13 (Dec. 2016), pp. 415–427. DOI: 10.1016/j.nicl.2016.12.011.

[35] Jacqueline K. Kueper, Mark Speechley, and Manuel Montero-Odasso. "The Alzheimer's Disease Assessment Scale-Cognitive Subscale (ADAS-Cog): Modifications and Responsiveness in Pre-Dementia Populations. A Narrative Review". In: *Journal of Alzheimer's Disease* 63.2 (2018), pp. 423–444. DOI: 10.3233/JAD-170991.

[36] Q. Gao et al. "Utility of Polygenic Risk Scoring to Predict Cognitive Impairment as Measured by Preclinical Alzheimer Cognitive Composite Score". In: *JAR Life* 11 (Feb. 2022), pp. 1–8. DOI: 10.14283/jarlife.2022.1.

[37] Sheng-Yuan Lin et al. "The Clinical Course of Early and Late Mild Cognitive Impairment". In: *Frontiers in Neurology* 13 (May 2022), p. 685636. DOI: 10.3389/fneur.2022.685636. URL: https://doi.org/10.3389/fneur.2022.685636.

[38] Amalia Luque et al. "The impact of class imbalance in classification performance metrics based on the binary confusion matrix". In: *Pattern Recognition* 91 (2019), pp. 216–231. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2019.02.023. URL: https://www.sciencedirect.com/science/article/pii/S0031320319300950.

[39] Guillaume Lemaître. *Random Over-Sampling Example — imbalanced-learn Documentation.* Accessed: 2025-04-10. URL: https://glemaitre.github.io/imbalanced-learn/auto_examples/over-sampling/plot_random_over_sampling.html.

[40] Hongfang Zhou, Xiqian Wang, and Rourou Zhu. "Feature selection based on mutual information with correlation coefficient". In: *Applied Intelligence* 52.5 (2022), pp. 5457–5474. ISSN: 1573-7497. DOI: 10.1007/s10489-021-02524-x. URL: https://doi.org/10.1007/s10489-021-02524-x.

[41] Y. Chen et al. "Progression from normal cognition to mild cognitive impairment in a diverse clinic-based and community-based elderly cohort". In: *Alzheimer's Dementia* 13.4 (Apr. 2017). Epub 2016 Aug 30, pp. 399–405. DOI: 10.1016/j.jalz.2016.07.151. URL: https://doi.org/10.1016/j.jalz.2016.07.151.

[42] Kyriaki G. Yiannopoulou and Stavros G. Papageorgiou. “Current and Future Treatments in Alzheimer Disease: An Update”. In: *Journal of Central Nervous System Disease* 12 (Feb. 2020), p. 1179573520907397. DOI: 10.1177/1179573520907397.

[43] National Health Service (NHS). *Alzheimer’s Disease - Treatment*. Accessed: 2025-01-26. URL: https://www.nhs.uk/conditions/alzheimers-disease/treatment/.

Abbreviations

AD Alzheimer's Disease. 1, 7, 11, 12, 14–17, 19, 25–27, 29–35, 42, 45, 46, 64–66, 68, 69, 71, 80

ADAS-Cog Alzheimer's Disease Assessment Scale-Cognitive. 16

ADASYN Adaptive Synthetic Sampling. 43–45, 64, 69, 71

ADNI Alzheimer's Disease Neuroimaging Initiative. 11, 12, 16, 27, 28, 30, 32, 33, 68–71

ANN Artificial Neural Network. 20, 21

AUC Area Under the Curve. 64

CDR Clinical Dementia Rating. 16

CSF Cerebrospinal Fluid. 1, 11, 12, 17, 26–30, 32–34, 46, 47, 64, 67–69, 71, 72

FAQ Functional Activities Questionnaire. 33, 34, 66, 67

IDE Integrated Development Environment. 30

KNN K-Nearest Neighbours. 5, 6, 19, 22, 37, 38, 44, 48, 50, 55, 57, 69, 70

LONI Laboratory of Neuro Imaging. 30

LR Logistic Regression. 5, 19, 22, 29, 49, 50, 55, 58, 60, 70, 71

MAE Mean Absolute Error. 6, 37–39, 69

MCI Mild Cognitive Impairment. 1, 6, 7, 11, 12, 14–16, 19, 25–29, 31–35, 42, 45, 46, 64–66, 68, 69, 71

MICE Multiple Imputation by Chained Equations. 6, 27, 37–39, 69, 71

ML Machine Learning. 11, 12, 17, 18, 20–22, 24, 26, 29, 30, 69, 71

MLP Multilayer Perceptron. 5, 6, 20, 21, 23, 52, 55, 57, 70

MMSE Mini-Mental State Examination. 16, 33, 34, 63

MRI Magnetic Resonance Imaging. 1, 11, 16, 26, 30, 31, 71

MSE Mean Squared Error. 6, 37–39, 69

NfL Neurofilament Light Chain. 17, 34, 67

PET Positron Emission Tomography. 1, 11, 16, 26, 30, 71

RAVLT Rey Auditory Verbal Learning Test. 16, 33, 34, 63, 66, 67

SCD Subjective Cognitive Decline. 14, 15

SMOTE Synthetic Minority Oversampling Technique. 43–45, 69

SVM Support Vector Machine. 5, 20, 23, 29, 51, 55, 70

Appendix A

Background

A.1 Treatment for Alzheimer's Disease

Currently, no medical cure exists for Alzheimer's Disease and disease-modifying strategies remain a focal point of ongoing research. However, certain pharmaceutical drugs such as N-methyl-D-aspartate (NMDA) receptor antagonists and Acetylcholinesterase (AChE) inhibitors have demonstrated effectiveness in reducing symptoms. These medications are useful for decelerating cognitive decline and managing behavioural disturbances which enables individuals to sustain functional independence for a longer duration despite their inability to arrest disease progression [42]. Other non-pharmacological approaches include cognitive stimulation therapy which aims to enhance memory retention and problem-solving abilities, and cognitive rehabilitation which focuses on helping individuals achieve personal goals and maintain autonomy in daily activities. Additionally, reminiscence therapy is another strategy which involves engaging individuals with past experiences through photographs and personal belongings to improve their mood and overall well-being [43].

Appendix B

Source Code

B.1 Common Utility Functions

```python
import matplotlib.pyplot as plt
import miceforest as mf
import numpy as np
import pandas as pd
from data_imputation import *
from sklearn.impute import KNNImputer
from sklearn.metrics import mean_squared_error, mean_absolute_error
from typing import Any, Callable

INPUT_DIR = "./raw_data"
OUTPUT_DIR = "./output_data"
DF_COLUMNS = ["RID", "VISCODE2"]
ADNIMERGE_COLUMNS = ["RID", "VISCODE"]


def create_bl_of_col(df: pd.DataFrame, column_name: str) -> pd.DataFrame:
    """
    Takes the baseline values of a specific column, creates a new
    column in the DataFrame with that column name post-fixed with
    `bl`, and inserts that bl value for each record for each `RID`.

    Args:
        pd.DataFrame: The input DataFrame containing `RID`,
                      `VISCODE`, and the specified column.
        str: The name of the column for which the bl values will be created.

    Returns:
        pd.DataFrame: A new DataFrame with an added column that
                      contains bl values for the specified column.
    """
    bl_name = f"{column_name}_bl"
    bl_values = df[df["VISCODE"] == "bl"][["RID", column_name]]
    df = df.merge(
        bl_values.rename(columns={column_name: bl_name}), on="RID", how="left"
    )
    for rid in df["RID"].unique():
        bl_value = df.loc[df["RID"] == rid, bl_name].iloc[0]
        if pd.notna(bl_value):
            df.loc[df["RID"] == rid, bl_name] = bl_value
    return df
```

```
def create_plasma_df(
    column_name: str, df_lst_with_map_f: list[tuple[pd.DataFrame, Callable]]
) -> pd.DataFrame:
    """
    Constructs a single DataFrame from a list of DataFrames
    with a corresponding mapping functions, creating rows
    with a specified column name, the `RID`, and `VISCODE`.

    Args:
        str: The name of the target column to be populated in the new DataFrame.
        list[tuple]:
            A list where each element is a tuple consisting of:
            - A DataFrame containing columns `RID` and `VISCODE2`.
            - A mapping function that transforms each
              row to create values for the target column.

    Returns:
        pd.DataFrame: A new DataFrame containing the columns `RID`, `VISCODE`,
                      and `column_name`, with duplicate and missing values
                      removed. The DataFrame is reset with a continuous index.
    """
    # Create a dataframe with these specific columns
    new_df = pd.DataFrame(columns=["RID", "VISCODE", column_name])
    new_df.drop(new_df.index, inplace=True)

    for df, map_f in df_lst_with_map_f:
        for i in range(df.shape[0]):
            # Catch any errors such as null values
            # or if the passed map function throws
            try:
                row = df.iloc[i]
                data = {
                    "RID": int(row.RID),
                    "VISCODE": row.VISCODE2,
                    # Apply the passed map function on each row
                    column_name: map_f(row),
                }
                new_df.loc[len(new_df.index)] = data
            except:
                pass

    # Drop duplicate, NaN rows and reset the index
    new_df.drop_duplicates(subset=ADNIMERGE_COLUMNS, inplace=True)
    new_df.dropna(inplace=True)
    new_df.reset_index(drop=True, inplace=True)
    return new_df


def df_of_csv(filename: str, input_dir: bool = True) -> pd.DataFrame:
    """
    Reads a CSV file into a DataFrame from the specified directory.

    Args:
        str: The name of the CSV file to read (without the `.csv` extension).
        bool: If True, find the file in the INPUT_DIR,
              else from OUTPUT_DIR. (Default is True)

    Returns:
        pd.DataFrame: The DataFrame containing the CSV data.
    """
    return pd.read_csv(f"{INPUT_DIR if input_dir else OUTPUT_DIR}/{filename}.csv")


def filter_n_years_from_bl(df: pd.DataFrame, n: int) -> pd.DataFrame:
    """
    Filters rows where `Years_bl` values are greater than or equal to n
    (inclusive), indicating the n-year mark from the baseline. Then,
    it removes duplicates within this subset, keeping only the first
    occurrence of each unique `RID` value.

    Args:
        pd.DataFrame: The DataFrame containing columns `Years_bl` and `RID`.
        int: Years from baseline to filter the data for.

    Returns:
        pd.DataFrame: A DataFrame filtered to rows with `Years_bl` values
                      greater than or equal to n (inclusive), , and with
                      duplicates removed based on `RID` (keeping only the
                      first occurrence of each 'RID').
    """
    # Filter the df for the records that are >= than n years from the bl
    n_years_from_bl = df[(df["Years_bl"] >= n)]
    # Return only the first record for each RID of the filtered df
    return n_years_from_bl.drop_duplicates(subset="RID", keep="first")


def cols_histogram_of_df(df: pd.DataFrame, title: str):
    """
    Generates and displays a set of histograms for each column in a DataFrame.

    Args:
        pd.DataFrame: The input DataFrame whose columns will be plotted as histograms.
        str: The title of the entire plot, displayed at the top.

    Returns:
        None
    """
    rows = int(np.ceil(len(df.columns) / 3))
    cols = 3
    fig, ax = plt.subplots(rows, cols, figsize=(10, 11))

    ax = ax.flatten()
    fig.suptitle(title)
    colors = plt.get_cmap("tab10")

    for i, col in enumerate(df.columns):
        ax[i].hist(
            df[col].astype(float),
            bins="auto",
            color=colors(i),
            edgecolor="black",
        )
        ax[i].set_xlabel(f"{col} Value", fontsize=9)
        ax[i].set_ylabel("Frequency", fontsize=9)
        ax[i].text(
            0.985,
            0.975,
            str_of_df_col_stats(df, col),
            transform=ax[i].transAxes,
            fontsize=6,
            verticalalignment="top",
            horizontalalignment="right",
            bbox=dict(
                boxstyle="round", facecolor="#F7E7C7", edgecolor="black", alpha=0.5
            ),
        )
    plt.tight_layout()
    plt.show()


def map_col_to_num(df: pd.DataFrame, column_name: str) -> pd.DataFrame:
    """
    Maps unique categorical values in a specified column of a DataFrame to integers.
    This is useful for encoding categorical data as numerical values.
    Args:
        pd.DataFrame: The input DataFrame containing the column to be mapped.
        str: The name of the column whose values should be mapped to integers.

    Returns:
        pd.DataFrame: A DataFrame with the specified column's values mapped to integers.
    """
    vals = df[column_name].unique()
    numerical_vals = list(range(len(vals)))
    df[column_name] = df[column_name].replace(vals, numerical_vals)
    print(f"{column_name}:\t{vals} -> {numerical_vals}")
    return df


def data_imputation(df: pd.DataFrame, imputation: Imputation_type) -> pd.DataFrame:
    """
    Imputes missing values in the input DataFrame using the specified imputation method.

    Args:
        pd.DataFrame: The input DataFrame containing missing values to be imputed.
        Imputation_type: The imputation method to be used for filling missing values.

    Returns:
        pd.DataFrame: A DataFrame with imputed values, containing no missing values.
    """
    match imputation:
        case Mice_forest(iterations, num_datasets):
            kernel = mf.ImputationKernel(df, num_datasets=num_datasets, random_state=0)
            kernel.mice(iterations=iterations)
            return kernel.complete_data()
        case Knn(n):
            return pd.DataFrame(
                KNNImputer(n_neighbors=n).fit_transform(df),
                columns=df.columns,
                index=df.index,
            )
        case _:
            return df


def print_no_of_rows_removed(df: pd.DataFrame, df1: pd.DataFrame) -> None:
    """
    Print the number & percentage of rows removed after filtering a DataFrame.

    Args:
        pd.DataFrame: The DataFrame before filtering.
        pd.DataFrame: The DataFrame after filtering.

    Returns:
        None
    """
    diff = df.shape[0] - df1.shape[0]
    percent = round((diff / df.shape[0]) * 100, 2)
    print(f"{diff} rows removed through data processing ({percent}% removed)")


def remove_float_values(n: Any) -> bool:
    """
    Checks if a value cannot be converted to a float, returning True
    if the conversion fails (indicating it's not a float-compatible value),
    and False if it succeeds (indicating it's float-compatible).

    Args:
        Any: The value to check for float compatibility.

    Returns:
        bool: True if the value cannot be converted to a float, otherwise False.
    """
    try:
        float(n)
    except ValueError:
        return True
    return False


def remove_rows_not_in_adnimerge(
    adnimerge_df: pd.DataFrame, df: pd.DataFrame, print_statistics: bool = True
) -> pd.DataFrame:
    """
    Filters a DataFrame by retaining only the rows with
    matching `VISCODE2` and `RID` in the ADNIMERGE dataset.

    Args:
        pd.DataFrame: The ADNIMERGE DataFrame.
        pd.DataFrame: The input DataFrame to be filtered.
        bool: Controls whether to print statistics about the filtering.

    Returns:
        pd.DataFrame: A filtered DataFrame with rows matching only
                      those also present in the ADNIMERGE dataset.
    """
    # Some df have the column labelled as VISCODE,
    # so to standardise it, we rename it to VISCODE2
    if "VISCODE" in df.columns and "VISCODE2" not in df.columns:
        df.rename(columns={"VISCODE": "VISCODE2"}, inplace=True)

    # Replace sc (screening) with bl (baseline) in the plasma df
    # since ADNIMERGE uses bl hence reduce noise in filtering
    df.loc[df["VISCODE2"] == "sc", "VISCODE2"] = "bl"

    filtered_df = df.merge(
        adnimerge_df,
        left_on=DF_COLUMNS,
        right_on=ADNIMERGE_COLUMNS,
        how="inner",
    )

    # Print some statistics about the filtering
    print_no_of_rows_removed(df, filtered_df) if print_statistics else None
    return filtered_df


def replace_nan_with_surrounding_matching_val(
    df: pd.DataFrame, column_name: str
) -> pd.DataFrame:
    """
    Fills NaN values in a specified column by matching values from adjacent rows
    when certain conditions are met.

    This function iterates through each row in the DataFrame, looking for
    NaN values in the specified column. If a NaN is found in a row where:
    - The `RID` value is the same in the row above, current, and the row below.
    - The values in the specified column are identical in the rows immediately
      above and below the NaN row.

    If found, replace the NaN value with the matching surrounding value.

    Args:
        pd.DataFrame: The input DataFrame containing the data with NaN values
                      to be replaced.
        str: The name of the column where NaN values should be filled.

    Returns:
        pd.DataFrame: The modified DataFrame with NaN values in the specified
                      columns replaced, if they met the above criteria.
    """
    for i in range(1, df.shape[0] - 1):
        # Check if the value in the specified column is NaN
        if pd.isna(df.loc[i, column_name]):
            # Check if the RID above and below match the current RID and
            # that the values in the above and below column are the same
            if (
                df.loc[i - 1, "RID"] == df.loc[i, "RID"] == df.loc[i + 1, "RID"]
                and df.loc[i - 1, column_name] == df.loc[i + 1, column_name]
            ):
                # Replace NaN with the above and below matching value
                df.loc[i, column_name] = df.loc[i - 1, column_name]
    return df


def compute_mse_and_mae(
    true_df: pd.DataFrame, predicted_df: pd.DataFrame
) -> tuple[float, float]:
    """
    Compute the Mean Squared Error (MSE) and Mean Absolute Error (MAE) between
    the true and predicted values stored in DataFrames.

    Args:
        pd.DataFrame: A DataFrame containing the true values.
        pd.DataFrame: A DataFrame containing the predicted values.

    Returns:
        (tuple[float, float]): A tuple containing the MSE and MAE, in that order.
            - The first value is the MSE (Mean Squared Error).
            - The second value is the MAE (Mean Absolute Error).
    """
    mse = mean_squared_error(true_df, predicted_df)
    mae = mean_absolute_error(true_df, predicted_df)
    return float(mse), float(mae)


def statistics_df_of_df(
    df: pd.DataFrame, exclusion_cols: list[str] = []
) -> pd.DataFrame:
    """
    Computes common statistical measures for the numeric columns in a DataFrame
    and returns a new DataFrame containing those statistics.

    This function calculates the following statistics for each numeric column:
    - Mean
    - Count (Sample Size)
    - Standard Deviation
    - Median
    - Range
    - Interquartile Range (IQR)
    - Minimum Value
    - Maximum Value
    - Skewness

    It drops any NaN values before performing the calculations, and
    optionally, specific columns can be excluded from the analysis.

    Args:
        pd.DataFrame: The input DataFrame containing the numerical data.

        list[str]: A list of column names to exclude from
                   the statistics calculation. (Default is [])

    Returns:
        pd.DataFrame: A DataFrame containing the calculated
                      statistics for each numerical column.
    """
    filtered_df = df.drop(columns=exclusion_cols)
    stats_df = pd.DataFrame(
        columns=[
            "Column",
            "Count",
            "Mean",
            "STD",
            "Median",
            "Range",
            "IQR",
            "Min",
            "Max",
            "Skew",
        ]
    )
    for column in filtered_df.columns:
        # Drop NaN values for the specific column
        col_data = filtered_df[column].dropna().astype(float)
        stats_df.loc[stats_df.shape[0]] = {
            "Column": column,
            "Count": len(col_data),
            "Mean": col_data.mean().round(3),
            "STD": col_data.std().round(3),
            "Median": col_data.median().round(3),
            "Range": (col_data.max() - col_data.min()).round(3),
            "IQR": (col_data.quantile(0.75) - col_data.quantile(0.25)).round(3),
            "Min": col_data.min().round(3),
            "Max": col_data.max().round(3),
            "Skew": col_data.skew().round(3),
        }
    return stats_df


def str_of_df_col_stats(df: pd.DataFrame, col_name: str) -> str:
    """
    Generates a string representation of the stats for a specific column in a DataFrame.

    Args:
        pd.DataFrame: The DataFrame for which column statistics are to be computed.
        str: The name of the column for which statistics should be extracted.

    Returns:
        str: A formatted string containing the statistics for the specified column.
             Each statistic presented in the form of "statistic_name: statistic_value".
    """
    statistics_df = statistics_df_of_df(df)
    stats = statistics_df[statistics_df["Column"] == col_name]
    filtered_stats = stats.drop(columns=["Column"]).to_dict()
    text = ""
    for key, val in filtered_stats.items():
        key_2 = next(iter(val))
        text += f"{key}: {val[key_2]}\n"
    return text
```

B.2 Data Imputation Classes

```
from dataclasses import dataclass
from typing import Union


@dataclass
class Knn:
    """
    Represents a K-Nearest Neighbors (KNN) imputer configuration.
    Attributes:
        n (int): The number of nearest neighbors to use in imputation.
    """

    n: int


@dataclass
class Mice_forest:
    """
    Represents a MICE (Multiple Imputation by Chained Equations) forest
    imputer configuration.

    Attributes:
        iterations (int): The number of iterations for the MICE process.
        num_datasets (int): The number of imputed datasets to generate,
                            providing multiple complete data sets for
                            further analysis or aggregation.
    """

    iterations: int
    num_datasets: int


Imputation_type = Union[Knn, Mice_forest]
```

B.3 Model Training & Evaluation Utility Functions

```
import numpy as np
import pandas as pd
import random
import seaborn as sns
from enum import Enum
from imblearn.over_sampling import ADASYN, RandomOverSampler, SMOTE
from itertools import combinations
from matplotlib import pyplot as plt
from sklearn.calibration import CalibratedClassifierCV, calibration_curve
from sklearn.ensemble import (
    RandomForestClassifier,
    StackingClassifier,
    VotingClassifier,
)
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import (
    accuracy_score,
    auc,
    confusion_matrix,
    f1_score,
    make_scorer,
    mean_absolute_error,
    mean_squared_error,
    precision_score,
    recall_score,
    roc_auc_score,
    roc_curve,
)
from sklearn.model_selection import GridSearchCV, StratifiedKFold, train_test_split
from sklearn.neighbors import KNeighborsClassifier
from sklearn.neural_network import MLPClassifier
```

```python
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier


class Over_samplers(Enum):
    RANDOM = "Random"
    SMOTE = "SMOTE"
    ADASYN = "ADASYN"


def create_ensemble_model(
    cv_results: dict,
    X_train: pd.DataFrame,
    X_test: pd.DataFrame,
    Y_train: pd.Series,
    Y_test: pd.Series,
) -> dict:
    """
    Create and evaluate ensemble models (StackingClassifier and VotingClassifier)
    using the best estimators and returns the best ensemble model.

    Args:
        dict: The results from the `tune_hyperparameters` function.
        pd.DataFrame: The training data features.
        pd.DataFrame: The testing data features.
        pd.Series: The training data labels.
        pd.Series: The testing data labels.

    Returns:
        dict: A dictionary containing the information of the best ensemble model:
            - "type" (str): ("StackingClassifier" or "VotingClassifier").
            - "model" (object): The best ensemble model.
            - "accuracy" (float): The accuracy of the best ensemble model.
            - "estimators" (list): The base estimators used in the ensemble.
            - "predicted_labels" (ndarray): Predicted labels for the test data.
            - "predicted_prob" (ndarray): Predicted probabilities for the test set.
            - "metrics" (dict): A dictionary of performance metrics.
    """
    model_accuracy = lambda prediction: accuracy_score(Y_test, prediction)
    predicted_prob = lambda model: (
        model.predict_proba(X_test)[:, 1] if hasattr(model, "predict_proba") else None
    )
    compute_metrics = lambda y_true, y_pred: {
        "Accuracy": accuracy_score(y_true, y_pred),
        "Error Rate": 1 - accuracy_score(y_true, y_pred),
        "F1-Score": f1_score(y_true, y_pred, average="weighted"),
        "Precision": precision_score(y_true, y_pred, average="weighted"),
        "Recall": recall_score(y_true, y_pred, average="weighted"),
        "Mean Squared Error": mean_squared_error(y_true, y_pred),
        "Mean Absolute Error": mean_absolute_error(y_true, y_pred),
    }

    # Get the best estimators from the cv_results dictionary
    estimators = [
        (name, results["best_estimator"]) for name, results in cv_results.items()
    ]

    # Variables to store of the best model
    best_model = None
    best_accuracy = 0
    best_estimators = None
    best_predicted_labels = None
    best_predicted_prob = None
    best_metrics = None

    # Iterate through all combinations of estimators
    for i in range(1, len(estimators) + 1):
        for comb in combinations(estimators, i):
            comb = list(comb)

            # Perform StackingClassification
            stacking_classifier = StackingClassifier(
                estimators=comb,
                final_estimator=LogisticRegression(
                    **cv_results["Logistic Regression"]["best_params"]
                ),
                cv=5,
                n_jobs=-1,
            )
            stacking_classifier.fit(X_train, Y_train)
            stacking_pred = stacking_classifier.predict(X_test)
            stacking_accuracy = model_accuracy(stacking_pred)
            stacking_prob = predicted_prob(stacking_classifier)

            # Compare if this is the best model
            if stacking_accuracy > best_accuracy:
                best_model = stacking_classifier
                best_accuracy = stacking_accuracy
                best_estimators = comb
                best_predicted_labels = stacking_pred
                best_predicted_prob = stacking_prob
                best_metrics = compute_metrics(Y_test, stacking_pred)

            # Perform VotingClassification
            voting_classifier = VotingClassifier(
                estimators=comb, voting="soft", n_jobs=-1
            )
            voting_classifier.fit(X_train, Y_train)
            voting_pred = voting_classifier.predict(X_test)
            voting_accuracy = model_accuracy(voting_pred)
            voting_prob = predicted_prob(voting_classifier)

            # Compare if this is the best model
            if voting_accuracy > best_accuracy:
                best_model = voting_classifier
                best_accuracy = voting_accuracy
                best_estimators = comb
                best_predicted_labels = voting_pred
                best_predicted_prob = voting_prob
                best_metrics = compute_metrics(Y_test, voting_pred)

    # Return the best model, its accuracy, and other details
    return {
        "model": best_model,
        "accuracy": best_accuracy,
        "estimators": best_estimators,
        "predicted_labels": best_predicted_labels,
        "predicted_prob": best_predicted_prob,
        "metrics": best_metrics,
    }


def evaluate_ensemble_model(
    ensemble_model: StackingClassifier | VotingClassifier,
    df: pd.DataFrame,
    label_col: str,
    n_splits: int = 4,
    random_state: int | None = 42,
) -> dict:
    """
    Evaluate the performance of an ensemble model using k-fold cross-validation.

    Args:
        (StackingClassifier | VotingClassifier): The ensemble model to be evaluated.
        pd.DataFrame: The dataset containing features and the target label column.
        label_col str: The name of the column containing the target labels.
        (int, optional): The number of folds for stratified k-fold CV. (Default is 4)
        (int | None, optional): Random seed for reproducibility. (Default is 42)

    Returns:
        dict: A dictionary containing the following evaluation statistics:
            - Accuracy: Tuple of mean and std of accuracy scores across folds.
            - F1-Score: Tuple of mean and std of F1-scores across folds.
            - Precision: Tuple of mean and std of precision scores across folds.
            - Recall: Tuple of mean and std of recall scores across folds.
            - AUC: Tuple of mean and std of AUC scores across folds.
            - Confusion Matrix Normalized cumulative confusion matrix across all folds.
            - True Labels List of true labels from all folds.
            - Predicted Probabilities: List of predicted probabilities for
                                      the positive class from all folds.
    """
    X = df.drop(columns=label_col)
    Y = df[label_col]
    k_fold = StratifiedKFold(n_splits=n_splits, shuffle=True, random_state=random_state)
    (
        accuracy_scores,
        auc_scores,
        all_probabilities,
        all_true_labels,
        f1_scores,
        precision_scores,
        recall_scores,
    ) = (
        [],
        [],
        [],
        [],
        [],
        [],
        [],
    )
    cumulative_confusion_matrix = np.zeros((2, 2), dtype=int)
    for train_idx, test_idx in k_fold.split(X, Y):
        X_train, X_test = X.iloc[train_idx], X.iloc[test_idx]
        Y_train, Y_test = Y.iloc[train_idx], Y.iloc[test_idx]
        ensemble_model.fit(X_train, Y_train)
        Y_pred = ensemble_model.predict(X_test)
        Y_prob = ensemble_model.predict_proba(X_test)[:, 1]

        # Calculate metrics for the current fold
        accuracy_scores.append(accuracy_score(Y_test, Y_pred))
        auc_scores.append(roc_auc_score(Y_test, Y_prob))
        all_true_labels.extend(Y_test)
        all_probabilities.extend(Y_prob)
        f1_scores.append(f1_score(Y_test, Y_pred))
        precision_scores.append(precision_score(Y_test, Y_pred))
        recall_scores.append(recall_score(Y_test, Y_pred))
        cumulative_confusion_matrix += confusion_matrix(Y_test, Y_pred)

    return {
        "Accuracy": (np.mean(accuracy_scores), np.std(accuracy_scores)),
        "F1-Score": (np.mean(f1_scores), np.std(f1_scores)),
        "Precision": (np.mean(precision_scores), np.std(precision_scores)),
        "Recall": (np.mean(recall_scores), np.std(recall_scores)),
        "AUC": (np.mean(auc_scores), np.std(auc_scores)),
        "Confusion Matrix": cumulative_confusion_matrix
        / np.sum(cumulative_confusion_matrix),
        "True Labels": all_true_labels,
        "Predicted Probabilities": all_probabilities,
    }

def evaluate_predictor_accuracy(
    ensemble_model: StackingClassifier | VotingClassifier,
    X_test: pd.DataFrame,
    Y_test: pd.Series,
) -> pd.DataFrame:
    """
    Evaluate the accuracy of each individual predictor (base estimator)
    in the ensemble model and represent the results as a DataFrame.

    Args:
        (StackingClassifier | VotingClassifier): The ensemble model.
        pd.DataFrame: The testing data features.
        pd.Series: The testing data labels.

    Returns:
        pd.DataFrame: A DataFrame with each predictor's name and its accuracy score.

    Raises:
        ValueError: If an unsupported ensemble model is passed.
    """
    # Get the base estimator depending on the ensemble type
    if isinstance(ensemble_model, VotingClassifier):
        base_estimators = ensemble_model.estimators
    elif isinstance(ensemble_model, StackingClassifier):
        base_estimators = ensemble_model.named_estimators_.items()
    else:
        raise ValueError("Unsupported ensemble model type.")

    feature_accuracies = []
    for name, estimator in base_estimators:
        Y_pred = estimator.predict(X_test)
        accuracy = accuracy_score(Y_test, Y_pred)
        feature_accuracies.append({"Estimator": name, "Accuracy": accuracy})

    return pd.DataFrame(feature_accuracies)


def plot_ensemble_evaluation(results: dict, data_name: str, n_bins: int = 10) -> None:
    """
    Function to plot the evaluation metrics and visualizations for an ensemble model.

    Args:
        dict: The evaluation results dictionary containing metrics and confusion matrix.
        str: The name of the data (e.g., 'BASE') to be displayed in the plot titles.
        int: Number of bins to discretize the [0, 1] interval for the calibration curve.
             (Default is 10)

    Returns:
        None
    """
    # Create subplots with 2 rows and 2 columns
    fig, ((ax1, ax2), (ax3, ax4)) = plt.subplots(2, 2, figsize=(12, 10))

    colors = ["red", "blue", "skyblue", "orange", "green", "yellow", "black"]

    # Plot the confusion matrix on the first subplot (ax1)
    sns.heatmap(
        results["Confusion Matrix"] * 100,
        annot=True,
        fmt=".1f",
        cmap="inferno_r",
        xticklabels=["Stay In MCI", "Convert To AD"],
        yticklabels=["Stay In MCI", "Convert To AD"],
        annot_kws={"size": 11},
        ax=ax1,
    )
    for text in ax1.texts:
        text.set_text(f"{text.get_text()}%")
    ax1.set_title(f"Confusion Matrix [{data_name}]")
    ax1.set_xlabel("Predicted Label")
    ax1.set_ylabel("True Label")
```

```python
ax1.xaxis.set_tick_params(labelsize=11)
ax1.yaxis.set_tick_params(labelsize=11)

# Plot the ROC curve on the second subplot (ax2)
true_labels = results["True Labels"]
predicted_probabilities = results["Predicted Probabilities"]
fpr, tpr, _ = roc_curve(true_labels, predicted_probabilities)
roc_auc = auc(fpr, tpr)
ax2.plot(
    fpr,
    tpr,
    color=random.choice(colors),
    label=f"Area Under Curve = {roc_auc:.2f}",
    linewidth=3,
)
ax2.plot([0, 1], [0, 1], color=random.choice(colors), linestyle="--")
ax2.set_title(f"ROC Curve [{data_name}]")
ax2.set_xlabel("False Positive Rate")
ax2.set_ylabel("True Positive Rate")
ax2.legend(loc="lower right")
ax2.grid(alpha=0.375)
ax2.xaxis.set_tick_params(labelsize=11)
ax2.yaxis.set_tick_params(labelsize=11)

# Plot the metrics bar chart on ax3
metrics = ["Accuracy", "F1-Score", "Precision", "Recall"]
mean_vals = [results[metric][0] for metric in metrics]
std_vals = [results[metric][1] for metric in metrics]
bars = ax3.bar(
    metrics, mean_vals, yerr=std_vals, capsize=5, color=random.choice(colors)
)
ax3.set_ylim(0, 1)
ax3.set_xlabel("Metrics")
ax3.set_ylabel("Score")
ax3.set_title(f"Evaluation Metrics with Standard Deviations [{data_name}]")
legend_labels = [
    f"{metric}: {mean:.2f} ± {std:.2f}"
    for metric, mean, std in zip(metrics, mean_vals, std_vals)
]
ax3.legend(bars, legend_labels, loc="lower right")
ax3.xaxis.set_tick_params(labelsize=11)
ax3.yaxis.set_tick_params(labelsize=11)

# Plot the calibration curve on ax4
fraction_of_positives, mean_predicted_value = calibration_curve(
    results["True Labels"],
    results["Predicted Probabilities"],
    n_bins=10,
)
ax4.plot(
    mean_predicted_value,
    fraction_of_positives,
    marker="o",
    label="Calibration Curve",
    color=random.choice(colors),
    linewidth=3,
)
ax4.plot(
    [0, 1],
    [0, 1],
    linestyle="--",
    label="Perfectly Calibrated",
    color=random.choice(colors),
    linewidth=3,
)
ax4.set_title(f"Calibration Curve For Ensemble Model [{data_name}]")
ax4.set_xlabel("Mean Predicted Value")
ax4.set_ylabel("Fraction Of Positives")
```

```
    ax4.legend()
    ax4.grid(alpha=0.375)
    ax4.xaxis.set_tick_params(labelsize=11)
    ax4.yaxis.set_tick_params(labelsize=11)

    plt.subplots_adjust(wspace=0.3, hspace=0.4)
    plt.tight_layout()
    plt.show()


def pretty_print_ensemble_model_results(data_set_name: str, results: dict) -> None:
    """
    Prints the results of the model evaluation in a nicely formatted way.

    Args:
        str: The name of the CSV that the model was applied on.
        dict: The output of the `evaluate_ensemble_model` function.

    Returns:
        None
    """
    results_copy = results.copy()
    results_copy.pop("Confusion Matrix")
    results_copy.pop("True Labels")
    results_copy.pop("Predicted Probabilities")
    print(f"Results for the {data_set_name} dataset using the ensemble model -")
    for idx, (key, value) in enumerate(results_copy.items(), start=1):
        tabs = f"\t\t" if key == "AUC" else f"\t"
        print(f"{idx}. {key}:{tabs} {round(value[0], 2)} ± {round(value[1], 2)}")


def split_data(
    df: pd.DataFrame,
    label_col: str,
    test_size: float = 0.2,
    random_state: int | None = 42,
) -> tuple[pd.DataFrame, pd.DataFrame, pd.Series, pd.Series]:
    """
    Split a DataFrame into training and testing data/labels.

    Args:
        pd.DataFrame: The DataFrame to split.
        str: Name of the column to use as labels.
        float: The percentage of the dataset to use in the test split. (Default is 0.2)
        (int | None, optional): Random seed for reproducibility. (Default is 42)

    Returns:
        (X_train, X_test, Y_train, Y_test): The data and labels split into
                                            training and testing datasets.
    """

    # Remove the column that will be used as the label
    X = df.drop(columns=label_col)
    Y = df[label_col]

    # Split the dataset into train/test
    X_train, X_test, Y_train, Y_test = train_test_split(
        X, Y, test_size=test_size, random_state=random_state
    )
    return X_train, X_test, Y_train, Y_test


def over_sample_data(
    method: Over_samplers,
    X_train: pd.DataFrame,
    Y_train: pd.Series,
    random_state: int | None = 42,
) -> tuple[pd.DataFrame, pd.Series]:
    """
    Performs oversampling on the training data using the specified method.

    Args:
        Over_samplers: The oversampling method to use. Must be one of:
            - Over_samplers.ADASYN: Adaptive Synthetic Sampling.
            - Over_samplers.RANDOM: Random Oversampling.
            - Over_samplers.SMOTE:  Synthetic Minority Oversampling Technique.
        pd.DataFrame: The training feature data.
        pd.Series: The training target labels.
        (int | None, optional): Random seed for reproducibility. (Default is 42)

    Returns:
        (tuple[pd.DataFrame, pd.Series]): A tuple containing:
            - pd.DataFrame: The resampled feature data.
            - pd.Series: The resampled target labels.

    Raises:
        ValueError: If an unsupported oversampling method is provided.
    """
    match method:
        case Over_samplers.ADASYN:
            model = ADASYN(random_state=random_state)
        case Over_samplers.RANDOM:
            model = RandomOverSampler(random_state=random_state)
        case Over_samplers.SMOTE:
            model = SMOTE(random_state=random_state)
        case _:
            raise ValueError("Unsupported Oversampling Method")
    X_train_resampled, Y_train_resampled = model.fit_resample(X_train, Y_train)
    return X_train_resampled, Y_train_resampled


def tune_hyperparameters(
    X_train: pd.DataFrame,
    X_test: pd.DataFrame,
    Y_train: pd.Series,
    Y_test: pd.Series,
    random_state: int | None = 42,
) -> dict:
    """
    Perform hyperparameter tuning for multiple
    machine learning models using GridSearchCV.

    Args:
        pd.DataFrame: The training data features.
        pd.DataFrame: The testing data features.
        pd.Series: The training data labels.
        pd.Series: The testing data labels.
        (int | None, optional): Random seed for reproducibility. (Default is 42)

    Returns:
        dict: A dictionary containing the following information for each model:
            - "cv_results" (dict): Detailed results from GridSearchCV.
            - "best_params" (dict): Best hyperparameter combination found.
            - "best_f1_score" (float): Best F1 score achieved during CV.
            - "best_accuracy" (float): Accuracy score of the best estimator on test data.
            - "best_estimator" (object): The best trained model for each classifier.
            - "predicted_labels" (ndarray): Predicted labels for the test data.
            - "predicted_prob" (ndarray): Predicted probabilities for the test set.
    """
    # [0.001, 0.01, 0.1, 1, 10, 10]
    c = [10**i for i in range(-3, 2)]
    model_with_params = {
        "KNN": {
            "Model": KNeighborsClassifier(),
            "Parameters": {
                "n_neighbors": list(range(1, 16)),
                "weights": ["uniform", "distance"],
            },
        },
        "Logistic Regression": {
            "Model": LogisticRegression(random_state=random_state),
            "Parameters": {
                "C": c,
                "solver": [
                    "lbfgs",
                    "liblinear",
                    "newton-cg",
                    "newton-cholesky",
                    "sag",
                    "saga",
                ],
            },
        },
        "SVM": {
            "Model": SVC(random_state=random_state),
            "Parameters": {
                "C": c,
                "kernel": ["linear", "poly", "rbf", "sigmoid"],
            },
        },
        "MLP": {
            "Model": MLPClassifier(max_iter=500, random_state=random_state),
            "Parameters": {
                "activation": ["relu", "identity", "logistic", "tanh"],
                "hidden_layer_sizes": [(50,), (100,), (50, 50)],
            },
        },
        "Random Forest": {
            "Model": RandomForestClassifier(random_state=random_state),
            "Parameters": {
                "n_estimators": list(range(50, 201, 30)),
                "max_depth": [None] + list(range(10, 51, 10)),
            },
        },
        "Decision Tree": {
            "Model": DecisionTreeClassifier(random_state=random_state),
            "Parameters": {
                "criterion": ["gini", "entropy", "log_loss"],
                "max_depth": [None] + list(range(5, 31, 5)),
                "max_features": ["auto", "sqrt", "log2"],
            },
        },
    }

    scoring = {
        "f1_score": "f1_weighted",
        "accuracy": make_scorer(accuracy_score),
        "precision": "precision_weighted",
        "recall": "recall_weighted",
    }

    # Apply GridSearchCV for each model
    cv_results = {}
    CV = 5
    for name, model_with_params in model_with_params.items():
        grid = GridSearchCV(
            model_with_params["Model"],
            model_with_params["Parameters"],
            cv=CV,
            scoring=scoring,
            refit="f1_score",
            n_jobs=-1,
            verbose=False,
        )
        grid.fit(X_train, Y_train)
        best_model = grid.best_estimator_

        # Special handling for SVM, wrap with CalibratedClassifierCV
        if name == "SVM":
            best_model = CalibratedClassifierCV(best_model, cv=CV)
            best_model.fit(X_train, Y_train)

        Y_pred = best_model.predict(X_test)

        # Get probabilities or decision scores
        if hasattr(best_model, "predict_proba"):
            Y_prob = best_model.predict_proba(X_test)[:, 1]
        elif hasattr(best_model, "decision_function"):
            Y_prob = best_model.decision_function(X_test)
        else:
            Y_prob = None

        cv_results[name] = {
            "cv_results": grid.cv_results_,
            "best_params": grid.best_params_,
            "best_f1_score": grid.best_score_,
            "best_accuracy": accuracy_score(Y_test, Y_pred),
            "best_estimator": best_model,
            "predicted_labels": Y_pred,
            "predicted_prob": Y_prob,
        }
    return cv_results
```

B.4 Plasma Data Collection

```
#!/usr/bin/env python
# coding: utf-8

# # Plasma Data Collection From ADNI
#

# In[1]:


import os
import pandas as pd
from adni_utils import *


# In[2]:


# Create directory where all the results will be written to in this whole project
if not os.path.exists(OUTPUT_DIR):
    os.makedirs(OUTPUT_DIR)

adnimerge_rid_and_viscode_df = df_of_csv("ADNIMERGE")[ADNIMERGE_COLUMNS]

drop_rows_not_in_adnimerge = lambda df: remove_rows_not_in_adnimerge(
    adnimerge_rid_and_viscode_df, df
)


# ### Neurofilament Light Chain (NfL)
#
# This protein is part of the structure of a neuron and when neurons dies, NfL is
# released into the blood and CSF. Elevated NfL levels indicate potential damage
# to neurons, which can be related to many neurological conditions. In the context
# of AD, raised NfL levels suggest ongoing neurodegeneration, with plasma NfL
# quantity higher in MCI and AD dementia stages.
#

# In[3]:


blennow_nfl_df = df_of_csv("BLENNOW_PLASMA_NFL")
filtered_blennow_nfl_df = drop_rows_not_in_adnimerge(blennow_nfl_df)
filtered_blennow_nfl_df


# In[4]:


adx_df = df_of_csv("PLASMA_PROJECT_ADX")
filtered_adx_df = drop_rows_not_in_adnimerge(adx_df)
filtered_adx_df


# In[5]:


# Define a map function which takes a row and returns the NFL value for each df
nfl_dfs_with_map_f = [
    (filtered_blennow_nfl_df, lambda row: row.PLASMA_NFL),
    (filtered_adx_df, lambda row: row.NF_LIGHT),
]

nfl_df = create_plasma_df("PLASMA_NFL", nfl_dfs_with_map_f)
nfl_df


# ### Phosphorylated Tau (p-tau)
#
# High levels of p-tau is a strong indicator of the presence of AD, as this is
# linked to the formation of tangles in the brain, a primary cause of this disease.
#

# In[6]:


blennow_ptau_df = df_of_csv("BLENNOW_PLASMA_TAU")
filtered_blennow_ptau_df = drop_rows_not_in_adnimerge(blennow_ptau_df)
filtered_blennow_ptau_df


# In[7]:


fnih_ptau_df = df_of_csv("FNIH_PLASMA_PTAU")
filtered_fnih_ptau_df = drop_rows_not_in_adnimerge(fnih_ptau_df)
filtered_fnih_ptau_df


# In[8]:


ugot_ptau_df = df_of_csv("UGOT_PTAU181")
filtered_ugot_ptau_df = drop_rows_not_in_adnimerge(ugot_ptau_df)
filtered_ugot_ptau_df


# In[9]:
```

```
# Define a map function which takes a row and returns the ptau value for each df
ptau_dfs_with_map_f = [
    (filtered_blennow_ptau_df, lambda row: row.PLASMATAU),
    (filtered_fnih_ptau_df, lambda row: row.PTAU_181),
    (filtered_ugot_ptau_df, lambda row: row.PLASMAPTAU181),
]

ptau_181_df = create_plasma_df("PLASMA_PTAU", ptau_dfs_with_map_f)
ptau_181_df


# ### Amyloid- (A42/40)
#
# An imbalance between the production and clearance of A peptides, particularly
# A42, leads to amyloid plaque formation in the brain, a hallmark of AD.
# Therefore, the A42/40 ratio is particularly lower in patients with AD.
#

# In[10]:


araclon_ab_df = df_of_csv("ARACLON_ABETA")
filtered_araclon_ab_df = drop_rows_not_in_adnimerge(araclon_ab_df)
filtered_araclon_ab_df


# In[11]:


bateman_ab_df = df_of_csv("BATEMAN_LAB")
filtered_bateman_ab_df = drop_rows_not_in_adnimerge(bateman_ab_df)
filtered_bateman_ab_df


# In[12]:


shimadzu_ab_df = df_of_csv("SHIMADZU_ABETA")
filtered_shimadzu_ab_df = drop_rows_not_in_adnimerge(shimadzu_ab_df)
filtered_shimadzu_ab_df


# In[13]:


ugot_ab_df = df_of_csv("UGOT_ABETA")
filtered_ugot_ab_df = drop_rows_not_in_adnimerge(ugot_ab_df)

# There are a few rows in this dataset that
# without numerical values so we filter them out
condition = filtered_ugot_ab_df["AB_1_42"] != "-"
condition_2 = filtered_ugot_ab_df["AB_1_40"] != "-"

filtered_ugot_ab_df = filtered_ugot_ab_df[condition & condition_2]
filtered_ugot_ab_df


# In[14]:


upenn_ab_df = df_of_csv("UPENN_ABETA")
filtered_upenn_ab_df = drop_rows_not_in_adnimerge(upenn_ab_df)
filtered_upenn_ab_df


# In[15]:
```

```
uw_ab_df = df_of_csv("UW_ABETA")
filtered_uw_ab_df = drop_rows_not_in_adnimerge(uw_ab_df)
filtered_uw_ab_df


# In[16]:


# Compute the AB42/40 ratio given the AB42 and AB40 value
ab_ratio = lambda ab_42, ab_40: ab_42 / ab_40

# Define a map function which takes a row and returns the ab42/40 ratio for each df
ab_dfs_with_map_f = [
    (filtered_adx_df, lambda row: ab_ratio(row.ABETA42, row.ABETA40)),
    (filtered_araclon_ab_df, lambda row: ab_ratio(row.TP42, row.TP40)),
    (filtered_bateman_ab_df, lambda row: row.Abeta_4240),
    (filtered_shimadzu_ab_df, lambda row: ab_ratio(row.AB1_42, row.AB1_40)),
    (filtered_ugot_ab_df, lambda row: ab_ratio(row.AB_1_42, row.AB_1_40)),
    (filtered_upenn_ab_df, lambda row: ab_ratio(row.AB42, row.AB40)),
    (filtered_uw_ab_df, lambda row: ab_ratio(row.ABETA42, row.ABETA40)),
]

ab_42_40_df = create_plasma_df("PLASMA_AB_42_40", ab_dfs_with_map_f)
ab_42_40_df


# In[17]:


# Merge all three constructed df together, on the RID
# and VISCODE value, into a single df with an outer join
HOW = "outer"
nfl_plus_ptau_df = pd.merge(nfl_df, ptau_181_df, on=ADNIMERGE_COLUMNS, how=HOW)
plasma_df = pd.merge(nfl_plus_ptau_df, ab_42_40_df, on=ADNIMERGE_COLUMNS, how=HOW)
plasma_df


# In[18]:


# Export the plasma df
plasma_df.to_csv(f"{OUTPUT_DIR}/PLASMA_DATA.csv", encoding="utf-8", index=False)

# Print some statistics about the number of NaN
# values in the new df for each plasma type
print(plasma_df.count(), end="\n\n")
plasma_df_rows = plasma_df.shape[0]
for column in ["NFL", "PTAU", "AB_42_40"]:
    column_name = f"PLASMA_{column}"
    na_rows = plasma_df[column_name].isna().sum()
    percentage = round((na_rows / plasma_df_rows) * 100, 2)
    print(
        f"{na_rows} out of {plasma_df_rows} values are missing for {column_name} ({percentage}% missing)"
    )
```

B.5 Data Preprocessing

```
#!/usr/bin/env python
# coding: utf-8

# # Pre-Processing Of ADNIMERGE & Plasma Dataset
```

```
#

# ### Import The Required Libraries
#

# In[1]:


import matplotlib.pyplot as plt, numpy as np, pandas as pd
from adni_utils import *
from functools import reduce


# ### Load The ADNIMERGE Dataset
#

# In[2]:


adnimerge_df = df_of_csv("ADNIMERGE")
adnimerge_df


# ### Load The Plasma Dataset
#

# In[3]:


plasma_df = df_of_csv("PLASMA_DATA", input_dir=False)
plasma_df


# ### Find The Different Types Of Diagnosis
#

# In[4]:


# All the different values that are present in the dataset for diagnosis
print(f"DX_bl: \t{adnimerge_df['DX_bl'].unique()}")
print(f"DX: \t{adnimerge_df['DX'].unique()}")


# ### Filter For Rows With MCI As Baseline Diagnosis
#

# In[5]:


# Filter out all the records that don't have MCI as the baseline
# diagnosis as we aren't interested in this data for this study
baseline_mci_df = adnimerge_df[
    adnimerge_df["DX_bl"].str.contains("EMCI|LMCI", regex=True, case=False, na=False)
]
baseline_mci_df


# ### Merge The ADNIMERGE And Plasma Dataset
#

# In[6]:


# These are the essential columns we are concerned with as part of the modelling
columns = [
    "RID",
    "VISCODE",
    "DX_bl",
    "DX",
    "Years_bl",
    "AGE",
    "PTGENDER",
    "PTEDUCAT",
    "PTMARRY",
    "PTRACCAT",
    "APOE4",
    "ABETA_bl",
    "ABETA",
    "TAU_bl",
    "TAU",
    "PTAU_bl",
    "PTAU",
    "PLASMA_NFL_bl",
    "PLASMA_NFL",
    "PLASMA_PTAU_bl",
    "PLASMA_PTAU",
    "PLASMA_AB_42_40_bl",
    "PLASMA_AB_42_40",
    "MMSE_bl",
    "MMSE",
    "RAVLT_immediate_bl",
    "RAVLT_immediate",
    "RAVLT_learning_bl",
    "RAVLT_learning",
    "RAVLT_forgetting_bl",
    "RAVLT_forgetting",
    "RAVLT_perc_forgetting_bl",
    "RAVLT_perc_forgetting",
    "ADAS11_bl",
    "ADAS11",
    "ADAS13_bl",
    "ADAS13",
    "ADASQ4_bl",
    "ADASQ4",
    "mPACCdigit_bl",
    "mPACCdigit",
    "mPACCtrailsB_bl",
    "mPACCtrailsB",
    "FAQ_bl",
    "FAQ",
]

# Merge the ADNIMERGE dataset with the plasma dataset on RID and VISCODE
df = baseline_mci_df.merge(plasma_df, on=["RID", "VISCODE"], how="left").sort_values(
    by=["RID", "Years_bl"]
)

# Add a bl column for each of the plasma columns into the merged dataset
columns_to_process = ["PLASMA_NFL", "PLASMA_PTAU", "PLASMA_AB_42_40"]
df = reduce(create_bl_of_col, columns_to_process, df)[columns]

pd.set_option("display.max_columns", None)
df


# In[7]:


# As we can see in the record for RID 4, DX is NaN for one of the columns
df[df["RID"] == 4]


# In[8]:
```

```
# Replace NaN for DX with an actual value, if the value
# of DX for the above and below rows are the same
df = replace_nan_with_surrounding_matching_val(df, "DX")

# Remove any rows from the dataset that still have NaN in their diagnosis
# since we can't determine what they were diagnosed with in that visit
df = df.dropna(subset="DX")

df[df["RID"] == 4]


# In[9]:


# An example record of the plasma data merged into the ADNIMERGE dataset
# Some of the values in the table are non-numerical like ABETA_bl (>1700)
df[df["RID"] == 2018]


# ### Replace String Values For CSF Data With Numerical Values
#

# In[10]:


# Get all the columns with mixed data types so we can format the data up for them
mixed_types = df.columns[df.apply(lambda col: col.map(type).nunique() > 1)]

# Find the exact non-numerical values and the corresponding column
print("These are the following strings to change to numerical values -")
for column in mixed_types:
    column_data = df[column].unique().tolist()
    filtered = filter(remove_float_values, column_data)
    print(f"{column}:", list(filtered))


# In[11]:


# Define a replacement map that replaces the strings with a numerical
# value for some of the fields that have mixed types in their columns
replacement_map = {
    "ABETA": {">1700": 1701},
    "TAU": {">1300": 1301},
    "PTAU": {">120": 121, "<8": 7},
    "ABETA_bl": {">1700": 1701},
    "TAU_bl": {">1300": 1301},
    "PTAU_bl": {">120": 121},
}

df = df.replace(replacement_map)


# In[12]:


# Check the conversion from strings to numerical values has worked as expected
df[df["RID"] == 2018]


# ### Filter For Those Who Remain In MCI 2 Years From Their Baseline
#

# In[13]:


# Filtered df with those who begin with MCI and remain
# in MCI for the duration of the study (non-converters)
```

```
remain_mci = df[df["DX"].str.contains("MCI", regex=True, case=False, na=False)]
remain_mci


# In[14]:


# Get only the first record for each of non-converter that is 2 years after bl
remain_mci_two_years = filter_n_years_from_bl(remain_mci, 2)
remain_mci_two_years


# ### Filter For Those Who Convert To AD Within 2 Years From Their Baseline
#

# In[15]:


# Filtered df with those who begin with MCI and convert to AD during the study
convert_ad = df[df["DX"].str.contains("Dementia", regex=True, case=False, na=False)]
convert_ad


# In[16]:


# Filter the converters df, getting the records for those who convert within
# 2 years from the baseline visit, keeping the most recent record for each RID
convert_ad_df = convert_ad[convert_ad["Years_bl"] <= 2].drop_duplicates(
    subset="RID", keep="last"
)
convert_ad_df


# In[17]:


# Get RIDs of all those who converted within 2 years to exclude from the next df
convert_ad_within_two_years_rid = convert_ad_df["RID"].unique()

# Remove the data for all those RIDs who converted within 2 years from bl
filtered_convert_ad_df = convert_ad[
    (~convert_ad["RID"].isin(convert_ad_within_two_years_rid))
]

# Get those who converted 2 years after their bl visit, these will be non-
# converters for our study as they converted after the set 2 year threshold
convert_ad_after_two_years = filter_n_years_from_bl(filtered_convert_ad_df, 2)

# Replace those who converted after 2 years
# diagnosis from Dementia to MCI for consistency
convert_ad_after_two_years["DX"] = convert_ad_after_two_years["DX"].replace(
    "Dementia", "MCI"
)

convert_ad_after_two_years


# ### Create The Non-Conversion Dataset
#

# In[18]:


# Create the non-converters df by combining those who remained
# in MCI over the 2 year period and those who converted to AD
# after the two year period from the baseline
remain_mci_df = (
    pd.concat([remain_mci_two_years, convert_ad_after_two_years], ignore_index=True)
    .sort_values(by=["RID", "Years_bl"])
    .drop_duplicates(subset="RID", keep="first")
)
remain_mci_df


# ### Get Statistical Measurements For Both Converters And Non-Converters Datasets
#

# In[19]:


# Remove the following columns from the statistical analysis as these are non-numerical
exclusion_cols = [
    "RID",
    "DX_bl",
    "DX",
    "PTGENDER",
    "PTMARRY",
    "PTRACCAT",
    "VISCODE",
    "Years_bl",
]


# In[20]:


# Non-converters statistics
statistics_df_of_df(remain_mci_df, exclusion_cols)


# In[21]:


# MCI to AD converters statistics
statistics_df_of_df(convert_ad_df, exclusion_cols)


# ### Combine Both Into A Single Dataset
#

# In[22]:


merged_df = pd.concat([remain_mci_df, convert_ad_df])
merged_df.drop_duplicates(subset="RID", keep="last", inplace=True)

# We can drop RID, VISCODE and Years_bl since we
# no longer need this data for further processing
merged_df.drop(columns=["Years_bl", "RID", "VISCODE"], inplace=True)

# Reset Index
merged_df.reset_index(drop=True, inplace=True)
merged_df


# ### Replace Categorical Data With Numerical Values
#

# In[23]:


# These columns are non-numerical so we need to
# find their values so we can cast them to numbers
cols = ["PTGENDER", "PTMARRY", "PTRACCAT", "DX_bl", "DX"]
for col in cols:
    print(f"{col}:\t{merged_df[col].unique()}")


# In[24]:


# Map these columns to numerical values
pd.set_option("future.no_silent_downcasting", True)
merged_df = reduce(map_col_to_num, cols, merged_df)
merged_df


# ### Visualise The Amount Of Missing Data And The Statistics Of The Columns With The Most Missing Data
#

# In[25]:


plt.rcParams.update({"font.size": 14})

null_count_percent = (
    merged_df.isnull().sum().sort_values(ascending=False) / merged_df.shape[0]
) * 100
norm = plt.Normalize(null_count_percent.min(), null_count_percent.max())
cmap = plt.cm.inferno_r
scalable_mapper = plt.cm.ScalarMappable(cmap=cmap, norm=norm)
scalable_mapper.set_array([])

null_count_percent.plot.bar(
    color=(cmap(norm(null_count_percent.values))),
    width=0.95,
    fontsize=8,
    xlabel="Variables",
    ylabel="Percentage (%)",
    title="Percentage of Missing Values Per Column",
    figsize=(11, 6),
)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()


# In[26]:


plt.rcParams.update({"font.size": 10})

# Find the names of the columns that have more than 30% of the data missing
missing_more_than = 30
column_names = null_count_percent[null_count_percent > missing_more_than].index

# Create a new df of those specific columns
missing_data_df = merged_df[column_names]

# Plot the histograms for all those columns and the
# numerical statistics calculated above visually
cols_histogram_of_df(
    missing_data_df,
    f"Histogram For Variables With More Than {missing_more_than}% Missing Data",
)


# ### Data Imputation
#

# In[27]:
```

```
# Convert numerical values that are objects into float64 to be able to apply imputation
for col in merged_df.select_dtypes(include=["object"]).columns:
    merged_df[col] = pd.to_numeric(merged_df[col])


# #### Setup The Dataset And Mask For The Sampling Dataset
#

# In[28]:


# Setup a seed for reproducible results
SEED = 42
np.random.seed(SEED)

# Create a sample of the df with only those rows with no NaN values
sample_data = merged_df.dropna().sample(frac=0.8, random_state=SEED)

# Replace around 20% of the data in the df with NaN values
sample_nan = sample_data.mask(np.random.rand(*sample_data.shape) < 0.2)
sample_nan


# #### Perform KNN Imputation From K = 1 To K = 10
#

# In[29]:


# Compute the mse and mae for each K-value from
# K=1 to K=10 to see the imputation performance
knn_results = {}
for n in range(1, 11):
    knn_sample_imputed = data_imputation(sample_nan, Knn(n))
    mse, mae = compute_mse_and_mae(sample_data, knn_sample_imputed)
    knn_results[n] = {"DF": knn_sample_imputed, "MSE": float(mse), "MAE": float(mae)}


# #### Plot The MAE And MSE For Each Imputed Dataset For Each K
#

# In[30]:


plt.rcParams.update({"font.size": 14})

# Plot the MAE and MSE results from the imputation on a plot for each K
plt.figure(figsize=(11, 7))
plt.plot(
    knn_results.keys(),
    [v["MSE"] for v in knn_results.values()],
    color="red",
    label="Mean Squared Error",
    linestyle=(0, (5, 1)),
    marker="x",
)
plt.plot(
    knn_results.keys(),
    [v["MAE"] for v in knn_results.values()],
    color="green",
    label="Mean Absolute Error",
    linestyle=(0, (5, 1)),
    marker="o",
)
plt.xlabel("K (Number of Neighbours)")
plt.ylabel("Error Value")
plt.title("MSE and MAE for Different K Using KNN Imputation")
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.show()


# In[31]:


# K=2 has the best performance when performing imputation so look into its exact values
knn_res = knn_results[2]
print("2-NN Imputation Performance Results -")
print(f"Mean Squared Error:\t{knn_res['MSE']}\nMean Absolute Error:\t{knn_res['MAE']}")


# #### Perform MICE Forest Imputation For Iterations 3-5 With Each Iteration Having Datasets Of Size 3-10
#

# In[32]:


# Compute the mse and mae for MICE forest data imputation on the
# sample dataset for iterations from 3 to 5 and 3 to 10 datasets
# for each iteration to find the optimal parameters for imputation
mice_forest_results = {}
for i in range(3, 6):
    mice_forest_results[i] = {}
    for k in range(3, 11):
        mice_forest_sample_imputed = data_imputation(
            sample_nan.reset_index(drop=True), Mice_forest(iterations=i, num_datasets=k)
        )
        mse, mae = compute_mse_and_mae(sample_data, mice_forest_sample_imputed)
        mice_forest_results[i][k] = {
            "DF": mice_forest_results,
            "MSE": float(mse),
            "MAE": float(mae),
        }


# #### Plot The MAE And MSE For Each Dataset And Iteration
#

# In[33]:


# Plot the MAE and MSE results from the imputation for each iteration and dataset size
plt.figure(figsize=(11, 7))
for i in range(3, 6):
    plt.plot(
        range(3, 11),
        [mice_forest_results[i][k]["MSE"] for k in range(3, 11)],
        label=f"MSE (i={i})",
        marker="o",
        linestyle=(0, (5, 1)),
    )
    plt.plot(
        range(3, 11),
        [mice_forest_results[i][k]["MAE"] for k in range(3, 11)],
        label=f"MAE (i={i})",
        marker="x",
        linestyle=(0, (5, 1)),
    )
plt.title("MSE and MAE For Different Iterations And Dataset Sizes Using MICE Forest")
plt.xlabel("Number of Datasets")
plt.ylabel("Error Value")
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.show()


# In[34]:


# Iterations=4 and Datasets=4 has the best performance
# when performing imputation so look into its exact values
mice_res = mice_forest_results[4][4]
print("4 Iterations & 4 Datasets MICE Forest Imputation Performance Results -")
print(
    f"Mean Squared Error:\t{mice_res['MSE']}\nMean Absolute Error:\t{mice_res['MAE']}"
)


# ### Impute The Full Dataset Using MICE Forest
#
# As per the imputation sampling done above, we can gather that MICE Forest is better suited as the imputation method for the data
#

# In[35]:


# Use MICE forest to impute the whole dataset
mice_forest_df_imputed = data_imputation(
    merged_df, Mice_forest(iterations=4, num_datasets=4)
)
mice_forest_df_imputed


# ### Visualise The New Statistics For Columns That Had The Most Missing Data After Performing MICE Forest Imputation
#

# In[36]:


plt.rcParams.update({"font.size": 10})

# Plot the histograms of the imputed dataset for all those columns
# that initially had more than 30% missing data in the original dataset
cols_histogram_of_df(
    mice_forest_df_imputed[column_names],
    f"Histogram For Imputed Dataset Of The Variables That Initially Had More Than {missing_more_than}% Missing Data",
)


# ### Get The CSF And Plasma Column Names
#

# In[37]:


# Function that takes a list of column names and appends
# to that list the matching baseline column name
add_bl_version_to_lst = lambda x: x + [f"{y}_bl" for y in x]
csf_cols = add_bl_version_to_lst(["ABETA", "PTAU", "TAU"])
plasma_cols = add_bl_version_to_lst(["PLASMA_AB_42_40", "PLASMA_NFL", "PLASMA_PTAU"])

print(f"Plasma Columns:\t{plasma_cols}")
print(f"CSF Columns:\t{csf_cols}")


# ### Export The Processed Datasets As A CSV
#

# In[38]:
```

```python
datasets = [
    # Remove the CSF and plasma data from the imputed df
    (mice_forest_df_imputed.drop(columns=(csf_cols + plasma_cols)), "BASE_DATASET"),
    # Remove the plasma data from the imputed df
    ((mice_forest_df_imputed.drop(columns=plasma_cols), "BASE_WITH_CSF_DATASET")),
    # Remove the CSF data from the imputed df
    ((mice_forest_df_imputed.drop(columns=csf_cols), "BASE_WITH_PLASMA_DATASET")),
    # Export the fully imputed df
    (mice_forest_df_imputed, "FULL_DATASET"),
]

# Export each of these datasets
for df, name in datasets:
    df.to_csv(f"{OUTPUT_DIR}/{name}.csv", encoding="utf-8", index=False)
```

B.6 Model Training & Hyperparameter Optimisation

```python
#!/usr/bin/env python
# coding: utf-8

# # ML Model Training
#

# In[1]:


import matplotlib.pyplot as plt, numpy as np, pandas as pd
from adni_utils import df_of_csv, OUTPUT_DIR
from modelling_evaluation_utils import *
from pickle import dump
from sklearn.decomposition import PCA
from sklearn.metrics import (
    confusion_matrix,
    ConfusionMatrixDisplay,
    roc_curve,
    roc_auc_score,
)


# ### Load The Processed Dataset
#

# In[2]:


df = df_of_csv("FULL_DATASET", input_dir=False)
df


# ### Split The Data & Labels Into Training And Testing Sets
#

# In[3]:


X_train, X_test, Y_train, Y_test = split_data(df, "DX")


# ### Check The Class Balance Between Converters And Non-Converters
```

```
#

# In[4]:


non_converters = (df[df["DX"] == 0]).shape[0]
converters = df.shape[0] - non_converters
print(f"Count Of Data For MCI -> MCI:\t{non_converters}")
print(f"Count Of Data For MCI -> AD:\t{converters}")


# ### Perform Oversampling Using Different Methods And Visualise The Results
#

# In[5]:


SEED = 42
over_samplers = [method for method in Over_samplers]

# Initialize PCA for dimensionality reduction of the dataset to a 2D space
pca = PCA(n_components=2, random_state=SEED)

# Create subplots
num_methods = round((len(over_samplers) + 1) / 2)
fig, ax = plt.subplots(num_methods, 2, figsize=(11, 11))
ax = ax.flatten()

# Function to plot a scatter plot for the given dataset
plot = lambda idx, x, y: ax[idx].scatter(
    x[:, 0], x[:, 1], c=y, cmap="inferno", edgecolors="black", alpha=0.7
)

plt.rcParams.update({"font.size": 12})
# Plot original sample
X_train_2D = pca.fit_transform(X_train)
plot(0, X_train_2D, Y_train)
ax[0].set_title(f"Before Oversampling")
ax[0].set_xlabel("PCA Component 1")
ax[0].set_ylabel("PCA Component 2")
ax[0].xaxis.set_tick_params(labelsize=11)
ax[0].yaxis.set_tick_params(labelsize=11)

for idx, method in enumerate(over_samplers):
    # Apply oversampling
    X_train_resampled, Y_train_resampled = over_sample_data(
        method=method,
        X_train=X_train,
        Y_train=Y_train,
    )

    # Plot resampled data
    X_train_resampled_2D = pca.fit_transform(X_train_resampled)
    plot(idx + 1, X_train_resampled_2D, Y_train_resampled)
    ax[idx + 1].set_title(f"After Oversampling ({method.value})")
    ax[idx + 1].set_xlabel("PCA Component 1")
    ax[idx + 1].set_ylabel("PCA Component 2")
    ax[idx + 1].xaxis.set_tick_params(labelsize=11)
    ax[idx + 1].yaxis.set_tick_params(labelsize=11)
plt.tight_layout()
plt.show()


# ### Oversample The Training Dataset Using ADASYN (Adaptive Synthetic Sampling)
#
# From the plots above, we can see that the minority class, visualised in yellow, is scattered in the original dataset, which make
#
```

```python
# In[6]:


X_train_over_sampled, Y_train_over_sampled = over_sample_data(
    method=Over_samplers.ADASYN,
    X_train=X_train,
    Y_train=Y_train,
)


# In[7]:


def print_value_counts(series: pd.Series) -> None:
    """
    Prints out the number of converters and non-converters in the label series.

    Args:
        pd.Series: The labels where 0 is a non-converter and 1 is a converter

    Returns:
        None
    """
    value_counts = series.value_counts()
    count_of_non_converters = value_counts.get(0, 0)
    count_of_converters = value_counts.get(1, 0)
    print(f"Number of Non-Converters (MCI -> MCI): {count_of_non_converters}")
    print(f"Number of Converters (MCI -> AD): {count_of_converters}")


print("Before Oversampling The Dataset:")
print_value_counts(Y_train)
print()
print("After Oversampling The Dataset (Using ADASYN):")
print_value_counts(Y_train_resampled)


# ### Perform Hyper-Parameter Tuning For Different Classification Models
#

# In[8]:


# NOTE: This takes approximately 20 minutes to run
tuned_hyperparameters = tune_hyperparameters(
    X_train_over_sampled, X_test, Y_train_over_sampled, Y_test, random_state=SEED
)


# ### Plot The Different Performance Metrics For Each Classification Model
#

# In[9]:


fig, ax = plt.subplots(len(tuned_hyperparameters) // 2, 2, figsize=(15, 15))
ax = ax.flatten()

for i, (model_name, result) in enumerate(tuned_hyperparameters.items()):
    cv_results = result["cv_results"]
    first_param = list(result["best_params"].keys())[0]
    param_values = cv_results["param_" + first_param]
    # RandomForest and DecisionTree use None for max_depth, we need
    # to format it to be able to visualise the results in the plot
    param_values = (
        np.where(param_values.mask, -1, param_values.data)
        if isinstance(param_values, np.ma.MaskedArray)
        else np.array(pd.Series(param_values).replace([None], -1))
    )
    # Plot these specific metrics for each model
    metrics = [
        "mean_test_f1_score",
        "mean_test_recall",
        "mean_test_accuracy",
        "mean_test_precision",
    ]
    metric_names = ["F1 Score", "Recall", "Accuracy", "Precision"]
    for metric, name in zip(metrics, metric_names):
        mean_scores = cv_results[metric]
        ax[i].plot(param_values, mean_scores, label=name, marker="o", linewidth=2)
        ax[i].set_title(f"{model_name} Model")
        ax[i].set_xlabel(first_param.replace("_", " ").title())
        ax[i].set_ylabel("Mean Score")
        ax[i].xaxis.set_tick_params(labelsize=11)
        ax[i].yaxis.set_tick_params(labelsize=11)
        ax[i].legend()

plt.tight_layout()
plt.show()


# ### Plot The Confusion Matrix For Each Of The Classification Models
#

# In[10]:


fig, ax = plt.subplots((len(tuned_hyperparameters) // 2), 2, figsize=(15, 15))
fig.suptitle("Confusion Matrices For All Models")
ax = ax.flatten()

for i, (model_name, metrics) in enumerate(tuned_hyperparameters.items()):
    Y_pred = metrics["predicted_labels"]
    display = ConfusionMatrixDisplay(
        confusion_matrix=confusion_matrix(Y_test, Y_pred),
        display_labels=["Stay In MCI", "Convert To AD"],
    )
    display.plot(ax=ax[i], cmap="inferno")
    ax[i].set_title(f"Confusion Matrix For {model_name}")
    ax[i].xaxis.set_tick_params(labelsize=11)
    ax[i].yaxis.set_tick_params(labelsize=11)
plt.tight_layout()
plt.show()


# ### Plot The ROC Curve For Each Of The Classification Models
#

# In[11]:


plt.rcParams.update({"font.size": 12})
plt.figure(figsize=(9, 6))

# Add a diagonal line for the ROC curve
plt.plot([0, 1], [0, 1], "--", label="Random Guess (ROC)", linewidth=3, color="gray")

for model_name, metrics in tuned_hyperparameters.items():
    Y_prob = metrics["predicted_prob"]
    fpr, tpr, _ = roc_curve(Y_test, Y_prob)
    roc_auc = roc_auc_score(Y_test, Y_prob)
    plt.plot(
        fpr,
        tpr,
        label=f"{model_name} (Area Under Curve = {roc_auc:.2f})",
        linewidth=3,
    )

plt.title("ROC Curve For All Models")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.grid(alpha=0.375)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.show()


# ### Create An Ensemble Model Of All The Individual Models
#
# Create an ensemble model of all the models, finding the best combination of models which yields the highest accuracy on the test
#

# In[12]:


# NOTE: This takes a long time to run
ensemble_model = create_ensemble_model(
    tuned_hyperparameters, X_train_over_sampled, X_test, Y_train_over_sampled, Y_test
)


# ### The Final Ensemble Model
#

# In[13]:


# The ensemble model which combines the various models to give a robust model
ensemble_model["model"]


# ### Plot The Confusion Matrix Of The Ensemble Model
#

# In[14]:


display = ConfusionMatrixDisplay(
    confusion_matrix=confusion_matrix(Y_test, ensemble_model["predicted_labels"]),
    display_labels=["Stay In MCI", "Convert To AD"],
)
display.plot(cmap="viridis")
plt.title(f"Confusion Matrix For The Ensemble Model")
plt.tight_layout()
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()


# ### Plot The ROC Curve For The Ensemble Model
#

# In[15]:


plt.figure(figsize=(9, 6))

# Add a diagonal line for the ROC curve
plt.plot([0, 1], [0, 1], "--", label="Random Guess (ROC)", linewidth=3, color="gray")

Y_prob = ensemble_model["predicted_prob"]
fpr, tpr, _ = roc_curve(Y_test, Y_prob)
auc = roc_auc_score(Y_test, Y_prob)
```

```
plt.plot(
    fpr,
    tpr,
    label=f"Ensemble (Area Under Curve = {auc:.2f})",
    linewidth=3,
)
plt.title("ROC Curve For The Ensemble Model")
plt.xlabel("False Positive Rate")
plt.ylabel("True Positive Rate")
plt.grid(alpha=0.375)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.legend()
plt.show()


# ### Statistics Of The Ensemble Model
#

# In[16]:


pd.DataFrame.from_dict(ensemble_model["metrics"], orient="index", columns=["Value"])


# ### The Accuracy For Each Individual Model In The Ensemble
#

# In[17]:


evaluate_predictor_accuracy(ensemble_model["model"], X_test, Y_test)


# ### Serialise & Export The Ensemble Model
#

# In[18]:


FILENAME = "ENSEMBLE_MODEL.sav"
dump(ensemble_model["model"], open(f"{OUTPUT_DIR}/{FILENAME}", "wb"))
```

B.7 Model Evaluation

```
#!/usr/bin/env python
# coding: utf-8

# # ML Model Evaluation
#

# In[1]:


import matplotlib.pyplot as plt, pandas as pd
from adni_utils import df_of_csv, OUTPUT_DIR
from modelling_evaluation_utils import *
from pickle import load
from sklearn.feature_selection import mutual_info_classif
from sklearn.inspection import permutation_importance
```

```
# ### Load The Saved Ensemble Model
#

# In[2]:


plt.rcParams.update({"font.size": 10.5})
LABEL_COL = "DX"
RANDOM_STATE = 42
ENSEMBLE_DIR = f"{OUTPUT_DIR}/ENSEMBLE_MODEL.sav"
with open(ENSEMBLE_DIR, "rb") as f:
    ensemble_model = load(f)
ensemble_model


# ### Load The Pre-Processed Datasets
#

# In[3]:


load_dataset = lambda filename: df_of_csv(f"{filename}_DATASET", input_dir=False)
base = load_dataset("BASE")
base_with_csf = load_dataset("BASE_WITH_CSF")
base_with_plasma = load_dataset("BASE_WITH_PLASMA")
full = load_dataset("FULL")


# ### Evaluating The Base Dataset
#

# In[4]:


base_model_evaluation = evaluate_ensemble_model(ensemble_model, base, LABEL_COL)
pretty_print_ensemble_model_results("BASE", base_model_evaluation)


# In[5]:


plot_ensemble_evaluation(base_model_evaluation, "BASE")


# ### Evaluating The Base With CSF Dataset
#

# In[6]:


base_with_csf_model_evaluation = evaluate_ensemble_model(
    ensemble_model, base_with_csf, LABEL_COL
)
pretty_print_ensemble_model_results("BASE_WITH_CSF", base_with_csf_model_evaluation)


# In[7]:


plot_ensemble_evaluation(base_with_csf_model_evaluation, "BASE_WITH_CSF")


# ### Evaluating The Base With Plasma Dataset
#

# In[8]:
```

```
base_with_plasma_model_evaluation = evaluate_ensemble_model(
    ensemble_model, base_with_plasma, LABEL_COL
)
pretty_print_ensemble_model_results(
    "BASE_WITH_PLASMA", base_with_plasma_model_evaluation
)


# In[9]:


plot_ensemble_evaluation(base_with_plasma_model_evaluation, "BASE_WITH_PLASMA")


# ### Evaluating The Full Dataset
#

# In[10]:


full_model_evaluation = evaluate_ensemble_model(ensemble_model, full, LABEL_COL)
pretty_print_ensemble_model_results("FULL", full_model_evaluation)


# In[11]:


plot_ensemble_evaluation(full_model_evaluation, "FULL")


# ### Plot The Mutual Relationship Between Each Feature And `DX`
#

# In[12]:


X = full.drop(columns=LABEL_COL)
Y = full[LABEL_COL]
mutual_info = mutual_info_classif(X, Y, random_state=RANDOM_STATE)
mutual_info = pd.Series(mutual_info)
mutual_info.index = X.columns
mutual_info = mutual_info.sort_values(ascending=False)
norm = plt.Normalize(mutual_info.min(), mutual_info.max())
cmap = plt.cm.inferno_r
mutual_info.plot.bar(
    ylabel="Mutual Information Score",
    xlabel="Features",
    figsize=(11, 6),
    color=(cmap(norm(mutual_info.values))),
    width=0.95,
    title="Feature Importance Based On Mutual Information",
)
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()


# ### Plot The Permutation Feature Importance
#

# In[13]:


result = permutation_importance(
    ensemble_model, X, Y, scoring="accuracy", random_state=RANDOM_STATE, n_jobs=-1
)
perm_sorted_idx = result.importances_mean.argsort()
```

```
mean_importance = result.importances_mean[perm_sorted_idx]
plt.figure(figsize=(11, 10))
plt.barh(
    X.columns[perm_sorted_idx],
    mean_importance,
    color=(cmap(norm(mean_importance))),
    height=0.93,
)
plt.xlabel("Mean Permutation Importance")
plt.ylabel("Features")
plt.title("Permutation Feature Importance")
plt.xticks(fontsize=12)
plt.yticks(fontsize=12)
plt.show()
```