UNIVERSITY OF
Southampton

University of Southampton Research Repository

Copyright © and Moral Rights for this thesis and, where applicable, any accompanying data are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis and the accompanying data cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content of the thesis and accompanying research data (where applicable) must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holder/s.

When referring to this thesis and any accompanying data, full bibliographic details must be given, e.g.

Thesis: Aditya Sivanand (2026) "Gaussian Oblique Decision Tree Technique for Classification and Regression", University of Southampton, Faculty of Social Sciences, School of Mathematical Sciences, PhD Thesis, pagination.

Data: Aditya Sivanand (2026) Gaussian Oblique Decision Tree Technique for Classification and Regression.

UNIVERSITY OF SOUTHAMPTON

Faculty of Social Sciences
School of Mathematical Sciences

Gaussian Oblique Decision Tree Technique for Classification and Regression

by

Aditya Sivanand

MSc University of Oxford, BSc University of Leeds

ORCiD: 0009-0004-8152-8380

A thesis for the degree of
Doctor of Philosophy

March 2026

University of Southampton

Abstract

Faculty of Social Sciences
School of Mathematical Sciences

Doctor of Philosophy

Gaussian Oblique Decision Tree Technique for Classification and Regression

by Aditya Sivanand

Decision tree techniques remain a cornerstone of interpretable machine learning techniques. Specifically, axis-parallel decision tree techniques are very popular in real-world applications. However, traditional axis-parallel decision tree techniques that perform orthogonal splits construct staircase decision boundaries, which would limit the predictive power when presented with data that has strong feature interactions. Hence, we propose a novel oblique decision tree technique that constructs a multivariate decision tree, called Gaussian mixture model Oblique Decision Trees (GODT) for classification and regression. We also propose a novel quadratic decision tree technique that constructs quadratic decision boundaries at each non-terminal node to split the data, called Gaussian mixture model Quadratic Decision Tree (GQDT) for classification. In these proposed techniques, the Gaussian mixture model is efficiently incorporated into the architecture of a binary tree-building process. We propose a novel and computationally cheaper way to construct hyperplanes for the oblique tree and hypersurface for the quadratic decision tree techniques. This is because, most of the current techniques have a high computational cost associated to them when trying to achieve a good prediction accuracy. Experiments using the GODT for classification and regression tasks, and GQDT on classification tasks on open-source datasets illustrate that both GODT and GQDT performs better in most cases when compared against existing established techniques both in test accuracy and computational times. Furthermore, as single tree-based techniques are known to have high-variance and sensitivity to training data, we extend our work to build an ensemble of GODT and GQDT techniques. Finally, we run extensive experiments to test the performance of the techniques that we propose on open-source datasets and we compare the performance against established techniques like the Classification and Regression Trees (CART), Support Vector Machines (SVM), Standard Random Forest (RF), Linear Discriminant Analysis (LDA), and Quadratic Discriminant Analysis (QDA).

Contents

List of Figures xi

List of Tables xiii

Declaration of Authorship xvii

Acknowledgements xix

1 Introduction 1

2 Review of decision trees and ensemble of trees 5

2.1 Introduction . . . 5
2.2 Introduction to the Notation . . . 5
2.3 Axis-parallel Decision Trees . . . 6
2.3.1 Axis-parallel Decision Tree Learning Process . . . 10
2.3.2 Axis-parallel Decision Tree Metrics . . . 11
2.3.3 Axis-parallel Decision Tree Prediction . . . 13
2.4 Oblique Decision Tree . . . 14
2.5 Ensemble Learning . . . 15
2.5.1 Boosting . . . 16
2.5.2 Bagging (Bootstrap Aggregation) . . . 17
2.5.3 Baseline machine learning methods . . . 21
2.5.3.1 Support Vector Machines (SVM) . . . 21
2.5.3.2 Linear Discriminant Analysis (LDA) . . . 21
2.5.3.3 Quadratic Discriminant Analysis (QDA) . . . 21
2.5.4 Broader Machine Learning Context . . . 22

3 Gaussian Mixture Model Oblique Decision Trees for Classification and Regression 25

3.1 Introduction . . . 25
3.2 Related work . . . 32
3.2.1 CART . . . 32
3.2.1.1 Selecting the best node . . . 33
3.2.1.2 Selecting the best threshold value . . . 33
3.2.2 Oblique Classifier 1 . . . 34
3.2.2.1 Perturbation algorithm . . . 34
3.2.2.2 Randomization . . . 35
3.2.2.3 Impurity Measure . . . 35

3.2.3 Soft vs. Hard Clustering: Gaussian Mixture Model and K-means 36
3.2.3.1 Hard Clustering using K-means 36
3.2.3.2 Soft Clustering using GMM 37
3.3 The proposed method 37
3.3.1 Gaussian Mixture Model 38
3.3.2 Calibration of the Gaussian Mixture Model 39
3.3.2.1 Parameter estimation 39
3.3.2.2 Initialization 41
3.3.2.3 Induced linear split 41
3.3.3 Hyperplane Generation in GODT 41
3.3.3.1 Hyperplane Generation for Classification 41
3.3.3.2 Derivation of the linear hyperplane (Eqs. 3.3–3.6) 46
Assumptions: 46
Bayes decision rule: 46
Normal vector (Eq. 3.5). 47
Threshold (Eq. 3.6). 47
3.3.3.3 Hyperplane Generation for Regression 47
3.3.4 Hyperparameters for GODT 48
3.3.4.1 Hyperparameters in GODT-Classification 48
3.3.4.2 Hyperparameters in GODT-Regression 48
3.3.5 Non-Splitting condition 49
3.3.5.1 Non-splitting condition for GODT-Classification 49
3.3.5.2 Non-splitting condition for GODT-Regression 50
3.3.6 Initialization 50
3.3.7 Detailed description of Algorithm 5 and Algorithm 6 51
Node management and termination: 51
Gaussian mixture model fitting: 51
Initialization strategy: 52
Conversion of GMM parameters to an oblique split. 52
Child node creation. 52
3.4 Experimental Results and Discussions 52
3.4.1 Classification Results and Discussions 52
3.4.1.1 Machine Learning Datasets 52
3.4.1.2 Experimental Setup 53
3.4.1.3 Hyperparameter Tuning Strategy 56
3.4.1.4 Hyperparameter Ranges 58
3.4.1.5 Classification performance 58
3.4.1.6 Statistical Comparison 60
3.4.2 Regression Results 61
3.4.2.1 Machine Learning Datasets 61
3.4.2.2 Experimental Setup 62
3.4.2.3 Regression Performance 63

4 Gaussian Mixture Model Oblique Random Forest for Classification and Regression 67
4.1 Introduction 67
4.2 Related Work 73

4.2.1 Standard Random Forests . . . 73
4.2.2 Extremely Randomized Trees (Extra Tree) . . . 74
4.2.2.1 Technique Overview . . . 74
4.2.3 Rotation Forests . . . 74
4.2.3.1 Technique Overview . . . 75
4.2.3.2 Gaussian Mixture Model . . . 75
4.3 The proposed method . . . 76
4.3.1 Bootstrap Aggregation (Bagging) . . . 77
4.3.2 Feature sub-sampling technique . . . 78
4.3.3 GODT Base Learner Recap . . . 78
4.3.4 Hyperparameters for Gaussian Oblique Random Forest (GORF) . 79
4.3.5 Non-splitting Condition in GORF . . . 80
4.3.6 Initialization . . . 80
4.4 Experimental Results and Discussions . . . 81
4.4.1 Classification Results . . . 81
4.4.1.1 Machine Learning Datasets . . . 82
4.4.1.2 Experimental Setup . . . 82
4.4.1.3 Classification performance . . . 84
4.4.1.4 Statistical Comparison . . . 86
4.4.2 Regression Results . . . 87
4.4.2.1 Machine Learning Datasets . . . 88
4.4.2.2 Experimental Setup . . . 88

5 Gaussian Mixture Model Quadratic Random Forest for Classification 93
5.1 Introduction . . . 93
5.2 Related Work . . . 97
5.2.1 CART . . . 97
5.2.2 Random Forests . . . 97
5.2.3 Summary . . . 98
5.3 The Proposed Method . . . 98
5.3.1 Gaussian Mixture Model . . . 99
5.3.2 Calibration of the Gaussian Mixture Model . . . 99
5.3.3 Hypersurface Generation . . . 100
5.3.3.1 Derivation of the quadratic hypersurface . . . 101
Assumptions: . . . 101
Quadratic term: . . . 103
Linear term: . . . 103
Bias term: . . . 103
5.3.4 Non-splitting condition for GQDT-C . . . 104
5.3.5 Initialization . . . 104
5.3.6 Ensemble of GQDT-C . . . 105
5.3.7 Hyperparameters for GQDT-C and GQRF-C . . . 107
5.4 Experimental Results . . . 109
5.4.1 Machine Learning Datasets . . . 109
5.4.1.1 Experimental Setup . . . 109
5.4.1.2 Classification Performance . . . 111
5.5 Computational Complexity of the Proposed Methods . . . 115

5.5.1 Classical Decision Tree Baselines 116
5.5.1.1 CART 116
5.5.1.2 Classical Random Forest (RF) 116
5.5.2 Cost Components in the Proposed Methods 116
5.5.2.1 Algorithm 6 based Initialization 117
5.5.2.2 Gaussian Mixture Model (GMM) 117
GODT with diagonal, shared covariance: 118
GQDT with full, separate covariances: 118
5.5.2.3 Hypersurface Construction 118
GODT with linear hyperplane. 118
GQDT with quadratic hypersurface. 118
5.5.3 Per-node and Per-tree Complexity 118
5.5.3.1 GODT 118
5.5.3.2 GQDT 119
5.5.4 Ensemble Methods: GORF and GQRF 119
5.5.4.1 GORF 119
Initialization (Algorithm 6 with m_{try} features): 119
GMM (diagonal, shared covariance): 119
Total per-tree cost: 119
Total forest cost (with T trees): 120
5.5.4.2 GQRF 120
5.5.5 Comparison Table 120
5.5.6 Interpretation of Computational Behaviour 120
Low-dimensional datasets (small p): 120
Moderate-dimensional datasets (tens to hundreds of features): 121
High-dimensional datasets ($p \gg 1000$): 121
Small sample size (small N): 121
Large sample size (large N): 122
Overall Summary 122

6 Conclusion and Future Work **123**
6.1 Conclusion 123
6.2 Future Work 125

Appendix A **127**
Appendix A.1 Additional Results for GODT Classification 127
Appendix A.2 Further Results for GODT-R 129
Appendix A.3 Macro-averaged Precision, Macro-averaged Recall, and Macro-averaged F1 Scores for GODT-C 130
Appendix A.4 GORF-C Initialization Comparisons 133
Appendix A.5 Macro-averaged Precision, Macro-averaged Recall, and Macro-averaged F1-Scores for GORF-C, GQDT-C, GQRF-C 134
Appendix A.6 Further Results for GQDT-C and GQRF-C 137
Appendix A.7 Statistical Comparison of the GQDT-C and the GQRF-C techniques 138
Appendix A.8 Pseudocode for Extremely Randomized Trees and Rotation Forests 140

Appendix A.9 Tree depth of GODT-C and CART for different datasets 142
Appendix A.10 Ensemble size of GORF-C and RF-C three different datasets of different sizes and dimensions 143
Appendix A.11 Log-Log plot of increasing sample size and feature size against training time 144

References **145**

List of Figures

2.1 Axis-parallel Decision Tree top-down construction and block construction for features $x_{1\cdot 2}$ and threshold values $\theta_1, \theta_2, \theta_3$ 8
2.2 Oblique hyperplane and axis-parallel hyperplanes splitting the sample input space (Murthy et al., 1994) 15
2.3 This figure demonstrates the bias-variance trade-off scenario. The Figure 2.3 (a) demonstrates what happens with the model error rate as the model complexity increases. For a low complexity model, the variance is low and the bias is high. On the contrary, for a highly complex model, the variance is high and the bias is low. Figures 2.3 (b), 2.3 (c), and 2.3 (d) show examples of underfit model, good fit model, and overfit model respectively. 20

3.1 Oblique hyperplane and axis-parallel hyperplanes splitting the sample input space (Murthy et al., 1994) 29
3.2 K-means and GMM cluster comparison 38
3.3 Illustration of linear hyperplane of the form $\mathbf{w}^\top \mathbf{x} - d = 0$ constructing a top-down oblique decision tree 43
3.4 Comparison of hyperplanes for binary classification: The orange line represents the hyperplane $\mathbf{w}^\top \mathbf{x} - d = 0$ derived using the Euclidean distance ($\mathbf{w} = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$). The blue line represents the hyperplane adjusted for covariance ($\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$), incorporating the Mahalanobis distance), tilting to account for feature correlations and varying variances as shown by the deep blue covariance ellipsoid. 45
3.5 We create a simple toy 2D dataset of 3 classes, and we demonstrate how the GODT-C technique would perform splits at each node 60

4.1 Ensemble of GODT to create GORF 77

Appendix A.1 Log–log plot of training time vs training sample size for RF, GORF-C, and GQRF-C. Here, we gradually increase the training sample size on the Madelon dataset. 144
Appendix A.2 Log–log plot of training time vs number of features considered per split (*mtry*) for RF, GORF-C, and GQRF-C. Here, we gradually increase the *mtry* on the Madelon dataset. 144

List of Tables

3.1 Information of the datasets used during the experimental stage 53
3.2 Test accuracies of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code. 54
3.3 Training times of the models in seconds. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code. . . 56
3.4 Hyperparameter ranges for GODT. 58
3.5 Hyperparameter search space for baseline models (MATLAB implementations). 58
3.6 Wilcoxon signed-rank test p-values (raw and Bonferroni-corrected, $m = 5$) excluding errors 60
3.7 Average Friedman Rank and Mean Accuracy 61
3.8 Information of the datasets used during the regression experimental stage 62
3.9 This table presents the (Mean Absolute Error) MAE for each technique on the test dataset 64
3.10 This table presents the training computational times of each dataset . . . 65

4.1 Test accuracies of the models. 84
4.2 Training times of the techniques in seconds. 85
4.3 Wilcoxon signed-rank test p-values (raw and Bonferroni-corrected, $m = 4$) comparing GORF-C to each classifier 86
4.4 Average Friedman ranks and mean test accuracies across all datasets. . . 86
4.5 This table presents the (Mean Absolute Error) MAE for each technique on the test dataset 89
4.6 Training Times for each technique in seconds 90

5.1 Test accuracies of the tree-based techniques, i.e., GQDT-C, GODT-C, and CART. 110
5.2 Training times of the tree-based techniques, i.e., GQDT-C, GODT-C, and CART in seconds. 112
5.3 Test accuracies of the ensemble-based techniques, GQRF-C, GORF-C, RF-C. 113

5.4 Training times of the ensemble-based techniques, GQRF-C, GORF-C, RF-C in seconds. 114
5.5 Computational complexity of classical and proposed models. 120

Appendix A.1 Table comparing the performance of GODT-C technique on some datasets when using full covariance compared to diagonal covariance when implementing the GMM. We record the test accuracies and computational times after running the same model for 10 times. The computational times is recorded in seconds 127
Appendix A.2 Test accuracy of the GODT-C technique with initialization using Algorithm 6 vs random initialization 127
Appendix A.3 Computational Times of the GODT-C technique with initialization using Algorithm 6 vs random initialization 128
Appendix A.4 This table presents the Mean Squared Error (MSE) for each technique on the regression dataset 129
Appendix A.5 Macro-averaged Precision of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code. 130
Appendix A.6 Macro-averaged Recall of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code. . . 131
Appendix A.7 Macro-averaged F1-Score of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code. 132
Appendix A.8 Test accuracy of the GORF-C technique with initialization using Algorithm 6 vs random initialization 133
Appendix A.9 Computational Times of the GORF-C technique with initialization using Algorithm 6 vs random initialization 133
Appendix A.10 Macro-averaged Precision of the techniques 134
Appendix A.11 Macro-averaged Recall of the techniques 135
Appendix A.12 Macro-averaged F1 Score of the techniques 136
Appendix A.13 Table comparing the performance of GQDT-C technique on some datasets when using full covariance (Full Cov) compared to diagonal covariance (Diag Cov) when implementing the GMM. We record the test accuracies and computational times (Comp Time) after running the same model for 10 times. The computational times are recorded in seconds137

Appendix A.14 Table comparing the performance of GQDT-C technique on some datasets with random initialization of the cluster centroids vs initialization using algorithm 6. We record the test accuracies and computational times (Comp Time) after running the same model for 10 times. The computational times are recorded in seconds 137
Appendix A.15 Wilcoxon Signed-Rank test results comparing GQDT-C and GQRF-C to other classifiers in their respective groups 138
Appendix A.16 Average Friedman ranks and mean classification accuracies for tree-based and ensemble-based classifiers 138
Appendix A.17 Average tree depth of GODT and CART across datasets. Datasets are grouped by dimensionality and sample size. Lower values indicate more compact trees. 142
Appendix A.18 Predictive performance (accuracy, %) vs ensemble size on the Madelon dataset for GORF and Random Forest. 143
Appendix A.19 Predictive performance (accuracy, %) vs ensemble size on the USPS dataset for GORF and Random Forest. 143
Appendix A.20 Predictive performance (accuracy, %) vs ensemble size on the Satimages dataset for GORF and Random Forest. 143

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published as:

Signed:.. Date:..................

Acknowledgements

First and foremost, I am deeply grateful to my supervisors, Joerg Fliege and Erengul Dodd, for their unwavering support throughout my PhD. They believed in me and gave me the opportunity to pursue this journey. This project would not have been possible without their constant encouragement, guidance, and support. It has been an incredibly rewarding and memorable experience working with you on such exciting research. I have learned so much, and I truly feel fortunate to have had such exceptional mentors.

I would also like to extend my sincere thanks to DEVnet for funding my PhD. My gratitude goes to all the colleagues at DEVnet for warmly welcoming me into their team. I especially want to thank Ralf Werner for his steadfast support during my PhD. I thoroughly enjoyed our discussions as we explored and developed our models. I learned so much from our conversations. Thank you for all your help and support.

A special thanks to the team at Smallspark Space Systems as well. It was a fantastic experience collaborating on some truly exciting CFD use cases. I am grateful for the opportunity to be part of that work.

I would also like to thank my fellow PhD students for making this journey so enjoyable. I truly appreciated all our discussions, coffee breaks, ultimate tic-tac-toe, and the lunch breaks that made the Ketley Room a vibrant and fun place. A huge thanks to Kulvir for always answering my endless questions and supporting me throughout this process.

I am deeply thankful to my loving parents, Sivanand Bharadhwaj and Saisudha Sivanand, and my dear sister, Sambhavi Sivanand, for their unwavering support throughout my life. Your love, encouragement, and emotional strength have helped me overcome every challenge. I am especially grateful to my father for inspiring me to pursue a PhD and motivating me to seize this opportunity, and to my mother for her endless patience, for always listening to my worries, and reassuring me with her calm presence. Thank you to my sister for making my days brighter, keeping me entertained, and cooking exciting new recipes to keep my spirits high.

Finally, I wholeheartedly dedicate my deepest gratitude to my guru, Shri Shirdi Sai Baba. Your blessings, guidance, and presence have given me hope and direction. Whatever I am today is because of you.

Chapter 1

Introduction

The decision tree technique has been widely popular for over four decades, due to the simplicity in implementing the model and interpreting the outcome. Some examples of applications include computer vision (Nowozin et al., 2011), information retrieval (Averbuch et al., 2004), medical diagnosis (Pereira et al., 2017), intrusion detection (Benferhat et al., 2013), signal processing (Vanli et al., 2019), and many more similar-use cases. The decision tree algorithm is a machine learning technique that is used for classification and regression prediction tasks. This area of decision trees has been widely studied due to its simplicity, interpretability, and wide applications in classification and regression.

The main idea behind a decision tree technique is to partition the input space into multiple sub-regions, where predictions are assigned to feature-based splits in a hierarchical structure (Breiman et al., 1984a; Quinlan, 2014). When presented with some training data, constructing a decision tree involves two major steps, (1) constructing a hierarchical, tree-structured partition of the input space, and (2) learning to predict the labels within each leaf node. When presented with some test data, we would navigate down the constructed decision tree from the root node to the leaf node, and would assign the test data the label at the leaf node. The basic idea of a decision tree is to separate data using a specific splitting criterion, recursively (Yang et al., 2019). This procedure requires optimization at each internal node of the tree, that partitions the training data in the node into subsets according to some example splitting criterion.

Popular decision tree techniques include Iterative Dichotomiser 3 (ID3) (Quinlan, 1986), Classification and Regression Trees (CART) (Breiman et al., 1984a), and C4.5 (Quinlan, 2014) with splitting criterion like information gain (Quinlan, 1986) for ID3 and C4.5 (Salzberg, 1994) or Gini impurity index (Breiman et al., 1984a) for CART. CART and C4.5 have been ranked in the top 10 algorithms in data mining experimentation (Wu et al., 2008). These popular techniques use axis-parallel splits,

which are splits that are perpendicular to the individual feature axes. If we construct ensembles of these axis-parallel decision trees, these would then lead to techniques like the Random Forest (Breiman, 2001), Bagging (Breiman, 1996a), and Gradient Boosting Machines (Friedman et al., 2001). When implementing these ensemble techniques across multiple classification and regression tasks, these have shown exceptional performance, which has caused them to be fundamental tools in numerous applications (Dietterich, 2000).

Even though axis-parallel decision trees have been very successful, they have some limitations. When presented with a situation that requires modelling complex decision boundaries, axis-parallel decision trees struggle as these complex decision boundaries naturally do not align with feature axes (Murthy et al., 1994). If axis-parallel decision trees are used to model data that requires complex decision boundaries, this would lead to numerous sequential splits, which would result in decision trees that are very deep and complex. Moreover, the overly complex model can result in overfitting and reduced or even lack of generalization of the model across different datasets. This issue is prominent when a dataset is high-dimensional, or contains features that are correlated or non-linear, which is common in most real world applications (Menze et al., 2011).

To address these challenges, oblique decision trees were introduced as encouraging alternatives to axis-parallel decision trees (Heath et al., 1993; Murthy et al., 1994). Contrary to the latter, oblique decision trees use a linear combination of multiple features, leading to a split that is formed due to hyperplanes at different angles. This would then allow an oblique decision tree to accurately model data that is best split using diagonal decision boundaries, which would allow for more flexibility. Previous work in the literature has observed that oblique decision trees can achieve the same performance as its axis-parallel counterparts by using fewer splits and reduced model complexity (Murthy et al., 1994; Wickramarachchi et al., 2016). However, oblique decision trees have not been widely implemented due to inherent computational difficulties. This is because, when constructing an oblique decision tree, finding an optimal hyperplane requires solving computationally intensive optimization problems. This has been a major setback for the use of oblique decision trees in literature (Manwani and Sastry, 2011; Wickramarachchi et al., 2016).

The idea of oblique splits has prompted researchers to propose the quadratic decision trees, which splits the data using quadratic or second-order polynomial decision boundaries (Ng and Leung, 2004). Constructing decision trees with quadratic decision boundaries allows the tree to model non-linear and curved decision surfaces, thus potentially enhancing predictive accuracy when compared to axis-parallel or linear decision boundaries. However, even though, theoretically, quadratic decision trees have their advantages, it is an area that has not been explored much in the literature.

This is because, constructing quadratic splits is considered more computationally intensive than constructing oblique splits (Bertsimas et al., 2017).

At the same time, the performance and robustness of machine learning models have improved massively due to the use of ensemble learning methods. Existing ensemble-based technique like Random Forests (Breiman, 2001), Bagging (Breiman, 1996a), Gradient Boosting Machines (Friedman et al., 2001), Rotational Forest (Rodriguez et al., 2006), and Extremely Randomized Trees (Geurts et al., 2006) increase the accuracy, reduce the variance, and improve robustness of the predictions across different datasets by combining predictions from individual decision trees (Dietterich, 2000). Most of the work in the area of ensemble methods has focussed on using axis-parallel decision trees. There is limited research in using either oblique or quadratic decision trees within the ensemble methodology. However, there is a high potential for oblique or quadratic-based ensemble techniques to perform well (Kuncheva and Rodríguez, 2007; Menze et al., 2011).

The gaps in the research provide us motivation for our thesis. Notably, there is a well-defined and pressing need for further work in the area of oblique and quadratic decision trees, where we focus on developing more efficient algorithms, capable of overcoming computational barriers without sacrificing accuracy. Moreover, developing ensemble techniques of these decision tree variants like oblique and quadratic are justified, as this could increase the accuracy of model significantly, mainly when we are working with datasets that require complex decision boundaries.

In this thesis, we present research that aims to address the current limitations in the area of oblique and quadratic decision trees and their respective ensembles. We present a novel and computationally efficient approach to use Gaussian mixture models to build decision trees that construct decision boundaries that are not axis-parallel. We extend this work further to create an ensemble of the decision tree techniques that we propose.

The reminder of the thesis is organized as follows:

- In Chapter 2, we review decision trees and set up the notation. We briefly review the theory behind ensemble learning, mainly looking at the ensemble of decision trees, and the widely used varieties. This chapter majorly focusses on explaining how tree-based techniques and ensemble techniques are constructed.
- In Chapter 3, we provide a detailed literature review of axis-parallel decision trees and oblique decision trees. We also propose a novel oblique decision tree technique that uses Gaussian mixture models called Gaussian Oblique Decision Tree (GODT) technique as an alternative to existing approaches to address the computational intensity without compromising on the accuracy. We implement the GODT technique on a combination of large-scale, high-dimensional, and

normal-sized datasets, and compare the performance and computational times against existing well established machine learning models. We also implement this technique for both classification and regression.

- In Chapter 4, we provide a detailed literature review of axis-parallel and oblique ensemble techniques. We also extend the GODT technique proposed in Chapter 3 to construct an ensemble of the GODT technique called Gaussian Oblique Random Forest (GORF) technique for classification and regression. We implement the GORF technique on a combination of large-scale, high-dimensional, and normal-sized datasets, and compare the performance and computational times against existing well-established machine learning models.
- In Chapter 5, we explore the idea of constructing a decision tree where the decision boundaries are non-orthogonal and non-linear. We propose a novel decision tree technique where we have a quadratic hypersurface called Gaussian Quadratic Decision Tree (GQDT) technique. This is an area that is very rarely explored, mainly due to the computationally intensive process to optimize a quadratic decision boundary. We propose a computationally cheaper way to construct this. Moreover, we demonstrate that this is done without compromising the accuracy of the technique on a wide variety of datasets. Furthermore, we extend the GQDT technique to construct an ensemble called Gaussian Quadratic Random Forest (GQRF). We implement GQDT and GQRF on a wide variety of datasets focussing on classification. We also compare the performance against other tree-based techniques. In this chapter, we also look at the computational complexity of the tree-based techniques using the big-O-notation.
- In Chapter 6, we conclude our work and discuss opportunities to explore further in the future.

Chapter 2

Review of decision trees and ensemble of trees

2.1 Introduction

In this chapter, we will be reviewing the concept behind a decision tree technique and ensemble learning. We will start by defining some notation that we will use throughout this chapter and the rest of the thesis. Then, we discuss axis-parallel decision trees, along with the learning process of these axis-parallel decision trees. Following this, we will introduce oblique decision trees before briefly looking at ensemble learning, mainly bootstrap aggregation (bagging) and boosting. Finally, we will wrap this chapter by looking at the concept of bias-variance trade-off and a discussion on other baseline machine learning techniques used in this thesis.

2.2 Introduction to the Notation

The decision tree technique is a supervised machine learning problem. In a supervised machine learning task, we are presented with a feature matrix $\mathbf{X} \in \mathbb{R}^{n \times p}$, with $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_n]^T$, that has n instances/observations (rows) and p attributes (columns). The feature matrix $\mathbf{X}$ is an $n \times p$ matrix, where each row in the matrix corresponds to an instance and the columns correspond to the p features. For a supervised learning problem, for every instance, we have a corresponding label/target. So, we are given the target values that are represented as $\mathbf{y} = [y_1, y_2, ..., y_n]^T$, which is an $n \times 1$ vector, which can also be represented as $\mathbf{y} \in \mathbb{R}^{n \times 1}$. For every row, we have a corresponding target value. We therefore represent the dataset as $\mathcal{D} = \{(\mathbf{X}, \mathbf{y})\}$, where $\mathbf{X} \in \mathbb{R}^{n \times p}$ and $\mathbf{y} \in \mathbb{R}^{n \times 1}$. We can also say

$\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$ to suggest that the first p columns represent the features with the last column corresponding to the target variable.

We focus on three scenarios of supervised learning:

- *Binary classification* is a special case of multi-class classification where the labels take a binary value of either a 0 or 1, -1 or 1, and so on. So, the $y_i \in \{1, 2\}$ for $i = 1, ..., n$, as there will be two classes, 1 and 2 namely.
- *Multi-class classification* is where the labels belong to multiple classes and take the form $y_i \in \{1, 2, ..., c\}$ for $i = 1, ..., n$, where c corresponds to the unique multiple classes in the dataset.
- *Regression* is where the target values belong to the real space, i.e., $y_i \in \mathbb{R}$ for $i = 1, ..., n$. So, the labels are real numerical values.

2.3 Axis-parallel Decision Trees

A decision tree is a hierarchically organized, axis-parallel structure performing a recursive binary partitioning of the input space $\mathbf{X}$. It is a tree-based structure with each node splitting the input data space into smaller regions (or simply boxes), by optimizing the values of a feature to split on when constructing the tree until a stopping criterion is met, such as when all the observations belong to a single class in classification. These small boxes or subsets are known as blocks. These are simply predictive models that tend to apply a series of conditions until a leaf node is reached. According to Friedman et al. (2001), a decision tree that we build is the same as partitioning the overall space $\mathbb{R}^p$, which is a p-dimensional feature space into a set of disjoint regions $\{P_1, P_2, ..., P_q\}$, where q corresponds to the number of leaf nodes. This can be represented as follows:

$$\mathbb{R}^p = \bigcup_{j=1}^{q} P_j \text{ with } P_j \cap P_k = \emptyset \text{ for } j \neq k. \tag{2.1}$$

Each region P_i is a subset of the overall space $\mathbb{R}^p$, and all the points within a subset are assigned the same class labels for classification and regression value for regression. The decision tree technique is divided into classification trees and regression trees depending on whether the labels in the dataset are categorical or continuous respectively.

The decision tree architecture is mainly divided into *univariate* (or *axis-parallel*) decision trees and *multivariate* (or *oblique*) decision trees. The axis-parallel decision tree on the input space $\mathbf{X}$ will be a binary, hierarchical, and orthogonal partition of the

input space, which devises a decision rule to predict the labels for the test data. The axis-parallel decision tree is a type of decision tree where at each non-terminal node, only a single feature is considered for the split rule during the node splitting process. In this thesis, we use the terms axis-parallel and univariate interchangeably. Similarly, we use multivariate and oblique interchangeably.

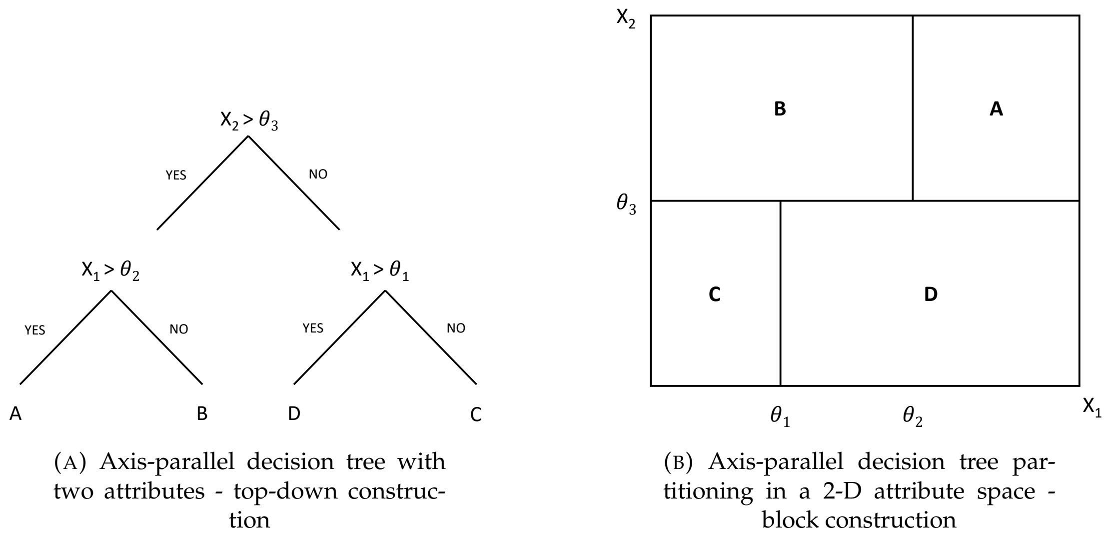
In this axis-parallel setting, we assign a split rule to each non-terminal node, which is a binary-valued function of the feature vector and we assign a class label to each terminal node, also known as the *leaf node*. The decision tree built will be strictly binary, finite and rooted tree **T**, i.e., finite set of nodes such that

1. every node k has exactly one parent node, $\mathcal{P}_k$ except for the root node $\mathcal{RT}_k$, which will not have any $\mathcal{P}_k$
2. every node k, except the leaf node, is the $\mathcal{P}_k$ of exactly two children nodes, called the left child $\mathcal{L}_k$ and the right child $\mathcal{R}_k$.
3. every node k without any children are called the leaf node $\mathcal{LF}_k$.

The axis-parallel decision tree can also be represented in the form of blocks $B_k \subset \mathbb{R}^p$. Every node $k \in \mathbf{T}$ is connected with a block B_k. At the root node of the decision tree **T**, we have $B_\Omega = \mathbb{R}^P$, where the root note Ω represents the whole data. Each internal non-terminal node $k \in \mathbf{T} \backslash \mathcal{LF}_k$ with two child nodes representing a split of its parent's node block into two sections, with $\zeta_k \in \{1, ..., P\}$ corresponding to the dimension of the axis-parallel split, and the θ_k corresponding to the constant threshold value or the location of the split on the axis. To be precise, if we have the respective left child, $\mathcal{L}_k$ and the right child, $\mathcal{R}_k$, we can define the respective blocks of left child node $B_{\mathcal{L}_k}$ and right child node $B_{\mathcal{R}_k}$ as follows:

$$B_{\mathcal{L}_k} := \{x \in B_k | x_{\zeta_k} \leq \theta_k\}, \text{ and } B_{\mathcal{R}_k} := \{x \in B_k | x_{\zeta_k} > \theta_k\}. \tag{2.2}$$

So, the information on **T**, $\boldsymbol{\zeta}$, and $\boldsymbol{\theta}$ make up a decision tree. When we are presented with a new input vector, to perform classification using a top-down decision tree approach, we begin at the root node, and work our way down the non-terminal nodes based on the split rule at each non-terminal node, until we arrive at the leaf node. The feature vector follows the hierarchy of the tree until it reaches the leaf node in a top-down manner, where we would use the class label of the leaf node to assign the class to the feature vector. We conduct a test to see if the input $x_{\zeta_k} \leq \theta_k, x_{\zeta_k} > \theta_k$, where the x_{ζ_k} is a corresponding feature and θ is a constant threshold value. This test is equivalent to an axis-parallel hyperplane in an attribute space, so forth given the name axis-parallel splitting.



(A) Axis-parallel decision tree with two attributes - top-down construction

(B) Axis-parallel decision tree partitioning in a 2-D attribute space - block construction

FIGURE 2.1: Axis-parallel Decision Tree top-down construction and block construction for features $x_{1 \cdot 2}$ and threshold values $\theta_1, \theta_2, \theta_3$

For example, the Figure 2.1 shows an example of an axis-parallel decision tree technique, when the corresponding partition creates a 2-D attribute space (Murthy et al., 1994). The Figure 2.1(A) is the hierarchical representation of the tree, whereas the Figure 2.1(B) is the block representation of the tree. In this example $\zeta_k = 1, 2$, for the threshold $\theta_k = \theta_1, \theta_2,$ and θ_3. This image illustrates how a hierarchical construction of a decision tree can be represented as blocks. Looking at Figure 2.1(A), at the root node, it is presented with the whole dataset, and starts with a decision rule on one feature, in this case if $X_2 > \theta_3$. If this decision rule is met, this would lead to the decision rule of if $X_1 > \theta_2$. If this is the case, we arrive at the leaf node of A, and if the rule is not met, we arrive at the leaf node of B. A similar process is followed if the decision rule of $X_2 > \theta_3$ fails to hold. The block construction in Figure 2.1(B) demonstrates the same concept.

In the context of axis-parallel decision trees, the objective is to partition the decision space into two binary classes by selecting the most appropriate feature and value to split on at each non-terminal node. This process involves evaluating all possible features and values for splitting at each non-terminal node, and selecting the combination that results in the lowest impurity score, which indicates that the node is pure. When a node is pure, it suggests that it contains data observations belonging to the same class. When we say pure, this usually means that all the data points in a region belong to the same class for classification or have a very similar value for regression. The leaf node is then created by converting the non-terminal node into the leaf node. Some popular axis-parallel decision tree techniques are ID3 (Quinlan, 1986), CART (Breiman et al., 1984b), and C4.5 (Quinlan, 2014), that use different impurity measures like information gain, GINI index, and gain ratio respectively, to select the optimal feature and split value at each non-terminal node. The impurity measure is used to evaluate the quality of a split, with a lower score indicating a better split. The

goal is to minimize the impurity score, which leads to a pure node, and ultimately, a well-defined decision boundary. In decision tree literature, impurity is a measure of how mixed the classes (or values) are in a region.

Algorithm 1 DecisionTreeAlgorithm(Dataset, Selection Condition, Stopping Condition)

```
 1: Input: Training data D ∈ ℝ^{n×(p+1)}, Selection Condition and Stopping Condition.
 2: Output: A fully constructed decision tree trained on the training data
 3: function DETERMINEBESTSPLIT(Node)
 4:     Input: The non-terminal or the root-node (at the first iteration)
 5:     Output: Best feature x_{ζ_k}, best threshold value θ_k, best left node B_{L_k}, best right node B_{R_k}.
 6:     Initialise the BestSelectCriterionValue = −∞ for Classification, and ∞ for regression
 7:     Initialise x_{ζ_k}, θ_k, B_{L_k}, B_{R_k}.
 8:     for each of x_{ζ_k} in the node do
 9:         for each unique threshold value in the feature do
10:             Split the data into B_{L_k} and B_{R_k} based on the threshold value
11:             if the size of B_{L_k} or B_{R_k} < Minimum Sample Size then
12:                 Continue to the next iteration
13:             end if
14:             Calculate the SplitCriterionValue
15:             if the SplitCriterionValue improve θ_k then
16:                 Update θ_k with SplitCriterionValue
17:                 Update x_{ζ_k}, θ_k, B_{L_k}, B_{R_k} with the current split
18:             end if
19:         end for
20:     end for
21:     return x_{ζ_k}, θ_k, B_{L_k}, B_{R_k}
22: end function
23: function GETPOTENTIALSPLIT(Node, Depth)
24:     Input: The non-terminal node (or the root node at the first iteration) and the existing depth of tree
25:     Output: A non-terminal TreeNode with children or represented as a leaf node itself
26:     if The Node is pure OR Depth ≥ MaxDepth OR # of sample in the node is < MinSampleSplit then
27:         leaf node created with label of the class as majority class for classification or mean for regression
28:         return LeafNode
29:     end if
30:     x_{ζ_k}, θ_k, B_{L_k}, B_{R_k} = DetermineBestSplit(Node)
31:     if the x_{ζ_k} is None then
32:         leaf node created with label of the class as majority class for classification or mean for regression
33:         return LeafNode
34:     end if
35:     Create a new TreeNode with x_{ζ_k} and θ_k
36:     Set TreeNode.LeftChild = GetPotentialSplit(B_{L_k}, Depth + 1)
37:     Set TreeNode.RightChild = GetPotentialSplit(B_{R_k}, Depth + 1)
38:     return TreeNode
39: end function
```

2.3.1 Axis-parallel Decision Tree Learning Process

The learning process of a decision tree involves learning the features to split on and the value to split on. Referring back to the Figure 2.1, this is represented as the x_1, x_2 and $\theta_1, \theta_2,$ and θ_3. Some popular axis-parallel decision tree algorithms include, ID3 (Quinlan, 1986), CART (Breiman et al., 1984b), and C4.5 (Quinlan, 2014). The difference between each technique is due to the type of impurity measure utilized, i.e., information gain for ID3, Gini index for CART, and gain ratio for C4.5.

A decision tree's training to prediction process can be split into 3 different functions, the tree initialization, determining the best split, and the node splitting process. Once we construct a decision tree, we would then need a function to perform the prediction.

We formulate a general axis-parallel decision tree architecture in Algorithm 1:

Algorithm 2 DecisionTreeInitialization(Dataset, Selection Condition, Stopping Condition)

1: **Input:** Training data $\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$, Selection Condition and Stopping Condition.
2: **Output:** A fully constructed decision tree trained on the training data
3: Initialize an empty decision tree: $T = \emptyset, \zeta = \emptyset, \theta = \emptyset$
4: Assign all the data to the *RootNode* of the tree
5: Tree = DecisionTreeAlgorithm(RootNode, Selection Condition, Stopping Condition)

Algorithms 1 and Algorithms 2 describe a general architecture for constructing a decision tree. The decision tree training consists of three main parts, decision tree initialization, the process to determine the best split, and performing the splits to separate the data.

From an algorithmic view point, the technique begins by *initializing* an empty decision tree and the respective variables. Note that the axis-parallel decision trees consider one feature at a time. After initializing an empty decision tree and the variables, the technique first assigns all the training data to the root node. The technique then tries to determine the best splitting feature and splitting value, before splitting the data.

Once the initialization process is complete, we try to *determine the best split*. For this, we assign all the data to the root node, and we begin by splitting the data at the root node into two non-terminal nodes (binary splits). The goal of the decision tree technique at the root node is to split the data in such a way that each node has data corresponding to a single class, such that when presented with an unseen data, it makes correct predictions. At any node k (which is the root node to begin with), the algorithm checks whether the data observations in the selected node are either pure or if the sample size is less than a threshold value or if the maximum depth has been reached. If none of these conditions are met, the tree tries to examine the different ways to split the data. At each iteration, starting at the root node, the algorithm analyzes and

evaluates all the features and all the possible split points to determine the best way to split a node. Then, for each feature and split value combination, the decision tree algorithm uses impurity measures to determine the quality of split, i.e., how well a split is separating the data, leading to a pure node. For a classification problem, depending on the type of decision tree technique, we pick the combination that either lowers the Gini impurity or increase the information gain. In a regression problem, we perform the split on the feature and split value that minimizes the mean squared error.

The algorithm then uses the GetPotentialSplit function iteratively on the left and the right nodes to construct a decision tree.

2.3.2 Axis-parallel Decision Tree Metrics

During the process of constructing an axis-parallel decision tree, we use a splitting criterion (or metric) at each node. The metric that is used varies from classification and regression. The metric is used to determine the quality of the split and to decide whether to split a node further. For classification, we use metrics like Gini impurity, entropy (information gain), and misclassification error. For regression, we use mean squared error. Below, we discuss these metrics.

Gini Impurity

During the tree construction process, the Gini impurity looks at the probability that a randomly chosen sample is misclassified i.e., incorrectly labelled. We can represent this as follows (Friedman et al., 2001):

$$\text{Gini}(B_k) = 1 - \sum_{c=1}^{C} p_c^2, \tag{2.3}$$

where B_k is the block corresponding to node k, C is the total number of classes, $p_c = \frac{n_c}{n}$ is the proportion of samples belonging to class c in the block, n_c is the total number of samples corresponding to class c in B_k and n is the total number of samples in B_k.

When we arrive at a node, our aim is to minimize the weighted Gini impurity of the child node:

$$\Delta\text{Gini} = \text{Gini}(B_k) - \left(\frac{n_{\mathcal{L}_k}}{n_k} \text{Gini}(B_{\mathcal{L}_k}) + \frac{n_{\mathcal{R}_k}}{n_k} \text{Gini}(B_{\mathcal{R}_k}) \right), \tag{2.4}$$

where B_k is the parent block, $B_{\mathcal{L}_k}$ and $B_{\mathcal{R}_k}$ are the left and right child blocks respectively, and $n_k, n_{\mathcal{L}_k}, n_{\mathcal{R}_k}$ are the number of samples in the parent, left, and right nodes, respectively.

Entropy (Information Gain)

Entropy measures the level of impurity or disorder in a dataset, in this case in a node. We define the entropy as follows (Friedman et al., 2001):

$$\text{Entropy}(B_k) = -\sum_{c=1}^{C} p_c \log_2(p_c), \tag{2.5}$$

where C is the total number of classes and $p_c = \frac{n_c}{n}$ is the proportion of samples belonging to class c in the block.

So now, the information gain is calculated as follows:

$$\text{Information Gain} = \text{Entropy}(B_k) - \left(\frac{n_{\mathcal{L}_k}}{n_k} \text{Entropy}(B_{\mathcal{L}_k}) + \frac{n_{\mathcal{R}_k}}{n_k} \text{Entropy}(B_{\mathcal{R}_k}) \right), \tag{2.6}$$

where B_k is the parent block $B_{\mathcal{L}_k}$ and $B_{\mathcal{R}_k}$ are the left and right child blocks respectively, and $n_k, n_{\mathcal{L}_k}, n_{\mathcal{R}_k}$ are the number of samples in the parent, left, and right blocks, respectively.

Misclassification Error

The misclassification error measures the fraction of incorrect predictions at a node. We define this as follows (Friedman et al., 2001):

$$\text{Misclassification Error}(B_k) = 1 - \max_c p_c, \tag{2.7}$$

where $p_c = \frac{n_c}{n}$ is the proportion of samples belonging to class c in the block.

We aim to minimize the weighted misclassification error across child nodes.

Mean Squared Error

In a regression task, we aim to split the data in a way that minimizes the variability of the target values. One popular measure used is the Mean Squared Error (MSE). The MSE measures the average squared difference between the predicted value and the actual target values. This is calculated as follows (Friedman et al., 2001):

$$\text{MSE}(B_k) = \frac{1}{n_k} \sum_{i=1}^{n_k} (y_i - \bar{y}_k)^2, \tag{2.8}$$

where y_i is the target value of the i-th sample in B_k, $\bar{y}_k = \frac{1}{n_k} \sum_{i=1}^{n_k} y_i$ is the mean of the target values in B_k, and n_k is the number of samples in B_k.

When we arrive at a node, our aim is to minimize the weighted sum of MSE of both the child nodes after the split:

$$\Delta\text{MSE} = \text{MSE}(B_k) - \left(\frac{n_{\mathcal{L}_k}}{n_k} \text{MSE}(B_{\mathcal{L}_k}) + \frac{n_{\mathcal{R}_k}}{n_k} \text{MSE}(B_{\mathcal{R}_k}) \right). \quad (2.9)$$

Once we construct an axis-parallel decision tree, when presented with test data, we would want to use the decision tree to predict the target values of the test data. We will look at axis-parallel decision tree prediction in the next section.

2.3.3 Axis-parallel Decision Tree Prediction

The reason we train a model is so that when presented with an unlabelled dataset, we can use the decision tree to predict the class labels for classification and numerical values for regression. To predict the labels for an unlabelled dataset, the input vector traverses through the trained decision tree recursively until it arrives at a leaf node $\mathcal{LF}_k$, where we assign this unlabelled input data with the label of the leaf node. For the classification setting, the label θ_k of the leaf node will be the label of the majority class in the leaf node, whereas for regression, the θ_k will be the mean on all the values in the leaf node. Algorithm 3 illustrates the decision tree prediction process.

Algorithm 3 DecisionTreePrediction(**T**, unlabelled data sample)

1: **function** DECISIONTREEPREDICT((Node, Sample))
2:     **if** Node $k \in \mathcal{LF}_k$ **then**
3:         **return** Class label of the Node
4:     **else**
5:         **if** $x_{\zeta_k} \leq \theta_k$ **then**
6:             **return** DecisionTreePrediction(Left Child Node, Sample)
7:         **else**
8:             **return** DecisionTreePrediction(Right Child Node, Sample)
9:         **end if**
10:     **end if**
11: **end function**

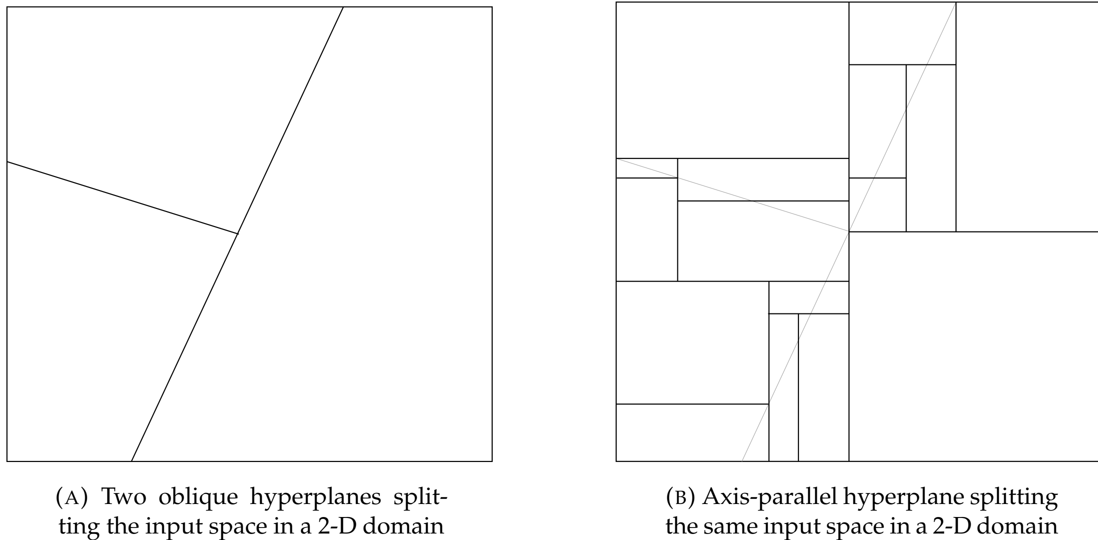
Once we create an initial split, the technique then iterates through the data below and data above the split to build trees until the nodes in the split region are pure or meet a certain threshold criterion. In a classification setting, the label will be the majority class label of the leaf node $\theta_{\mathcal{LF}_K}$, and in the regression setting, it will be the mean of $\theta_{\mathcal{LF}_K}$. Initially, we focused on building trees with axis-parallel splits at each node, and this can be considered as axis-parallel hyperplane in the feature space. Axis-parallel decision trees consider one feature at a time by performing an extensive search over

all the potential splits available and selects the split with the lowest overall entropy or the highest gain. An alternative to this is an oblique decision tree that performs a linear split by considering the linear combination of the feature space. We briefly introduce oblique decision trees in the next subsection. We will look at oblique decision trees in detail in Chapter 3.

2.4 Oblique Decision Tree

In section 2.3, we reviewed the theoretical underpinnings of axis-parallel decision trees and how these are constructed. In this section, we briefly introduce oblique (or multivariate) decision trees. In an oblique decision tree setting, at each non-terminal node, a linear hyperplane separates the classes on the decision tree in an aim to improve the purity of the nodes incrementally at the subsequent child nodes. The aim is to achieve the splitting and purity that the axis-parallel counter part achieves, but potentially with fewer iterations. Furthermore, in an oblique decision tree, we have multiple features participating in the node splitting, unlike its axis-parallel counter part. This is as a result of the oblique decision tree's better handling of non-linear relationships between features. Even though the univariate decision tree has its advantages, when we are presented with data that is correlated, we would prefer having a hyperplane splitting the input space due to its better separability rather than a step-like function. Moreover, the axis-parallel trees can provide us with complex tree structures and increased computational cost, when the inherent decision boundaries are not axis-parallel (Yang et al., 2019). Due to the flexible splits of the oblique decision trees, constructing an oblique decision tree can lead to fewer leaf nodes with the same computational accuracy with potentially a reduced tree size (Murthy et al., 1994). As illustrated using a very simple example in Figure 2.2, the axis-parallel decision tree can be equated to a multivariate decision tree, but in the axis-parallel case, the decision boundary is represented as steps, whereas in the multivariate case, the decision boundary is represented as a linear line (or a hyperplane). However, work in the area of oblique decision tree suggests that construction of oblique decision trees are generally computationally intensive due to the process of finding the best oblique split. We will look at Oblique decision trees in detail in Chapter 3.

In the next section, we will briefly introduce the background of ensemble learning, mainly boosting and bootstrap aggregation (bagging). This will briefly explain how an ensemble of decision trees can be constructed.



(A) Two oblique hyperplanes splitting the input space in a 2-D domain

(B) Axis-parallel hyperplane splitting the same input space in a 2-D domain

FIGURE 2.2: Oblique hyperplane and axis-parallel hyperplanes splitting the sample input space (Murthy et al., 1994)

2.5 Ensemble Learning

Ensemble learning is a practice in Machine Learning where two or more learners are aggregated to produce better performance. The work of Dasarathy and Sheela (1979) was the earliest example of implementing ensemble learning. They suggested to use multiple classifiers to partition the feature subspace. Hansen and Salamon (1990) demonstrated the improvement in classification performance by creating an ensemble of neural networks. However, Schapire et al. (1999) presented the idea of constructing strong classifiers using an ensemble of classifiers which is a better alternative to random guesses. This was given the name Boosting, with the popular *AdaBoost* algorithm being derived from this theory.

Dietterich (2000) discuss the reason for an ensemble technique performing better than a single classifier. There are two main reasons: statistical and computational. Firstly, we look at a statistical reason. When we have a given training data, we may have multiple hypotheses that may be equally good on this specific training data. So, following the idea of an ensemble by combining multiple hypotheses would eventually reduce the risk of choosing a single wrong hypothesis. Furthermore, we look at a computational reason. When an algorithm is prone to local optima, combining the predictions through an ensemble from different random searches may essentially provide a better approximation, that is much closer to the ground truth value.

When looking into building ensemble systems, there are three main strategies that we need to follow (Zhang and Ma, 2012): Data selection, training classifiers, and combining the classifiers. During the process of data sampling, it is important that we

select different subsets of the initial training data. Ultimately, the objective is to achieve different errors on the classifiers to essentially be able to gain from combining the classifiers. Ideally, the classifier outputs should be independent, or even better, negatively correlated (Qi et al., 2009). The most common technique to achieve diversity in the data is to take random subsets using *bootstrapping*. Bootstrapping refers to a re-sampling technique that is used to estimate the variability of a statistic or model parameter (Abney, 2002). It is a powerful and widely used method for making inferences about a population or sample when the underlying distribution is unknown or difficult to determine.

The basic idea behind bootstrapping in the context of ensemble learning is to create multiple re-samples, known as bootstrap samples, from the original data set by randomly selecting observations with replacement. By repeatedly re-sampling from the original data, we simulate the process of drawing multiple independent samples from the population.

Using bootstrapped sample data leads to bagging, which stands for bootstrap aggregation. Furthermore, we can also utilize a random subset of the available feature space during the training process. During the actual training process of the classifier, we use the decision tree algorithm to train the individual trees. Finally, we look at the different ways in which the classifiers can be combined. Like Zhang and Ma (2012) mentioned, the combination of the classifiers depends on the type of output they produce. For example, for categorical outputs, we use a majority voting strategy, whereas for continuous outputs, we use arithmetic methods, such as an average, sum or product. Creating ensembles is not just limited to tree-based algorithms. Any machine learning models' output can be combined together using ensemble learning. However, our work is focussed on creating an ensemble of tree-based techniques, specifically ensemble of non-axis-parallel decision trees.

In the next subsections, we will discuss the idea of Boosting and Bagging.

2.5.1 Boosting

Boosting is a type of ensemble learning that combines so-called set of "weak" learners into so-called "strong" learners in an aim to minimize the training error. The idea behind boosting is that we select a random sub-sample of the input data space, fit a model on the data, and train the model sequentially. The aim is that each model strives to offset for the weakness of the predecessor model. The weak rules from the individual classifiers are consolidated to create a single strong prediction rule with each iteration (Schapire et al., 1999).

Algorithm 4 Boosting (e.g., Gradient Boosting)

1: Initialize model $F_0(x) = 0$ ▷ $F_0(x) = 0$ represents the initial model
2: **for** $b = 1$ to B **do** ▷ B is the number of weak learners (trees)
3: Compute residuals r_i for each sample
4: Train weak learner $h_b(x)$ on residuals
5: Compute learning rate α_b
6: Update model: $F_b(x) = F_{b-1}(x) + \alpha_b h_b(x)$
7: **end for**
8: **return** final model $F_B(x)$

Boosting techniques are majorly divided into three algorithms, Adaptive boosting or AdaBoost, Gradient boosting, and Extreme gradient boosting (or XGBoost). The major difference between the three techniques is the way the weights are assigned to each training instance and the way the model is updated at each iteration. For example, the way the AdaBoost technique works is by sequentially identifying data points that have been misclassified and adjusting their weights with an aim to minimize the training error. Sequentially, the AdaBoost technique optimizes until it achieves a strong predictor. On the contrary, the Gradient boosting and XGBoost techniques train on the residual error of the previous predictor.

However, as suggested by Ferreira and Figueiredo (2012) even though boosted decision trees are slightly more accurate than random forests, they still hold some drawbacks. Boosted decision trees, when presented with a noisy dataset, are more sensitive. Moreover, the boosted trees computation cannot be parallelized across trees as the performance of one tree is dependent on its predecessor, which is unlike random forests. Another major drawback is that the extension of the boosting techniques to multi-class classification is not straight forward.

2.5.2 Bagging (Bootstrap Aggregation)

Bootstrapping is a tool that is used to measure statistical accuracy. This is just a procedure to assess the uncertainty of the estimators. In the context of ensemble learning, when we train multiple trees, each tree is trained on diverse bootstrapped samples of the original dataset. This new sub-sample of the data is acquired by sampling the initial data with replacement n times from the training dataset.

Bootstrap Aggregation, or Bagging algorithm by Breiman (1996a) is one of the earliest introductions to ensemble learning. Let us assume that we have the dataset as $\mathcal{D} = \{(\mathbf{X}, \mathbf{y})\}$, where $\mathbf{X} \in \mathbb{R}^{n \times p}$ and $\mathbf{y} \in \mathbb{R}^{n \times 1}$ and we fit a predictor $\hat{f}(\mathbf{x})$ on the data set.

Now, if we have B diverse bootstrap samples of the original dataset, and if we fit a predictor to each of these samples, we then arrive at the bootstrap estimator

$$\hat{f}^{(b)}(\mathbf{x}), \qquad b = 1, ..., B. \tag{2.10}$$

Bagging essentially is the procedure of averaging across all the B decision tree outputs that have been trained on the B diverse bootstrap samples, $\mathcal{D}_1, \mathcal{D}_2, ..., \mathcal{D}_B$. A predictor $f_b(\mathbf{x})$, which in this case a decision tree, is trained independently on each bootstrap sample $\mathcal{D}_b$. Once training is complete, the predictor fitted on the bth bootstrap sample is represented as $\hat{f}^{(b)}(\mathbf{x})$. The technique of bagging reduces the variance of the predictions and balances the performance of the trained model.

The following steps describes the process of bagging:

1. For $b = 1, ..., B$
   - Acquire bootstrap indices $(i_1, ..., i_n)$, from the bootstrap sample $\mathcal{D}_b$, which is generated by sampling n data observations from the original dataset with replacement.
   - We then train the model, and get the predictor $\hat{f}^{(b)}(x)$ based on the bootstrap sample $\mathcal{D}_b = \{(x_{i_1}, y_{i_1}), ..., (x_{i_n}, y_{i_n})\}$
2. For regression problems, the bagged predictor is obtained by averaging the predictions:
$$\hat{f}_{\text{Bagged}}(x) = \frac{1}{B}\sum_{b=1}^{B}\hat{f}^{(b)}(x). \tag{2.11}$$
3. For classification problems, the final prediction is obtained by majority voting among the B models:
$$\hat{f}_{\text{Bagged}}(x) = \text{mode}\{\hat{f}^{(1)}(x), \hat{f}^{(2)}(x), \ldots, \hat{f}^{(B)}(x)\}. \tag{2.12}$$

As a result, the model's performance with bagging can reduce the variance of the predictions when compared to without bagging. This can be represented as:

$$\mathbb{E}_D[(\hat{f}(x) - \mathbb{E}_D[\hat{f}(x)])^2] \geq \mathbb{E}_D[(\hat{f}_{Bagged}(x) - \mathbb{E}_D[\hat{f}_{Bagged}(x)])^2], \tag{2.13}$$

where $\mathbb{E}_{\mathcal{D}}$ describes the expectation over the distribution of the dataset. So, generally, the variance of the bagged model is less than that of the base model trained on the full dataset. However, the decrease in variance comes at a price of a small increase in the bias.

Furthermore, on every bootstrapped sample, we have some unused data which we can use to estimate the error of the bagged model. This is called the out-of-bag (OOB) test error estimate. So, for each $i = 1, ..., n$, the OOB sample is

$$\tilde{B}_i = \{b : x_i \text{ is not in the training set}\} \subseteq \{1, ..., B\}. \tag{2.14}$$

We would now have the following OOB estimate

$$\hat{f}^{OOB}(x) = \frac{1}{|\tilde{B}_i|} \sum_{m \in \tilde{B}_i} \hat{f}^{(b)}(x). \tag{2.15}$$

When constructing a random forest, the idea is to induce randomization into the individual trees, that contribute towards removing the correlation in predictions. In order to induce this randomness, the following strategy is followed:

- In bagging, the aim is to take random subsets of the training data when building each decision tree within the forest. The use of bagging essentially reduces the variance and prevents overfitting.
- We take the random sub-sample of the feature space, where for classification, this value is $\sqrt{p}$, where p is the number of features and for regression this is equal to $\frac{p}{3}$ (Probst et al., 2019). These are proposed starting values that could further be tuned.

The advantages of a random forest technique when compared to a decision tree technique are generally discussed using an important idea in statistics called bias-variance trade-off. So, when we have a model that displays high bias, it may be the situation that the model is oversimplifying the underlying relationship in the data. At the same time, such a model may consistently miss capturing important patterns within the data, leading to a low variance. So, a model with high bias has low variance.

On the contrary, a model with high variance is extremely sensitive to the data and captures the noise and random fluctuation in the data. Such a model fits the training data very well, failing to generalize on new unseen data. So, models with high variance are known to have low bias. Hence, it is key to build a model that is a trade-off between the bias and variance.

In the context of decision trees and random forests, separate decision trees are known to have low bias, but high variance, as variations to the dataset would produce different trees. There are, indeed, stopping criterias and pruning strategies that would help reduce the variance in the trees. However, the randomization of the decision

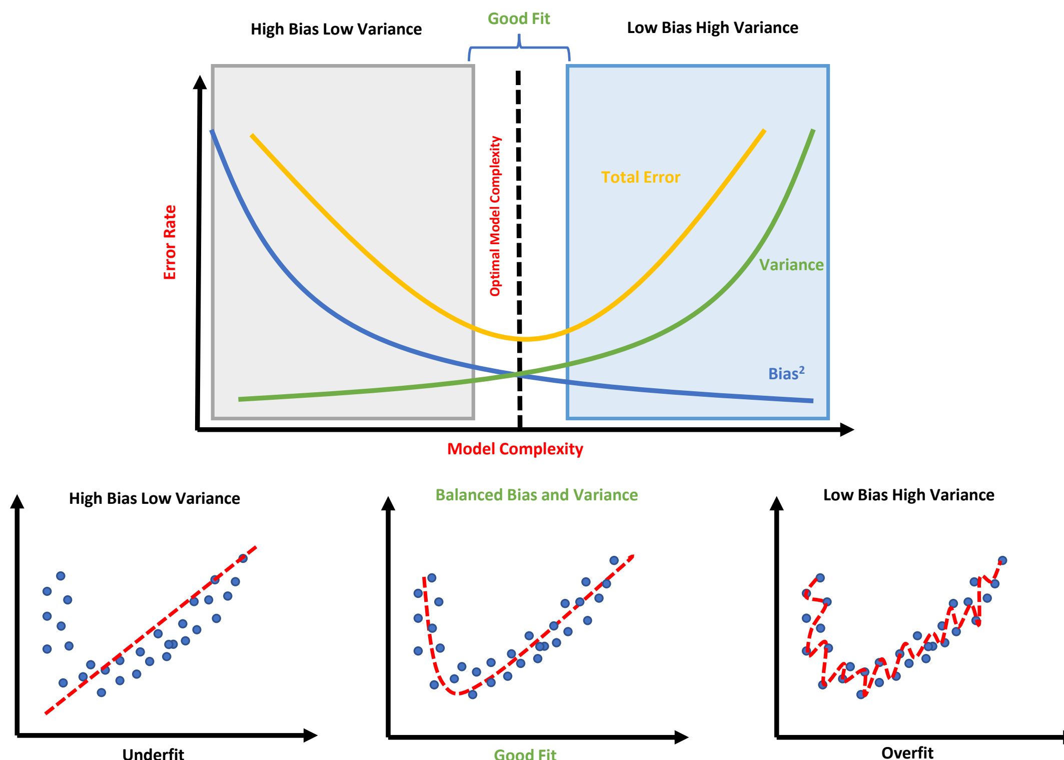


FIGURE 2.3: This figure demonstrates the bias-variance trade-off scenario. The Figure 2.3 (a) demonstrates what happens with the model error rate as the model complexity increases. For a low complexity model, the variance is low and the bias is high. On the contrary, for a highly complex model, the variance is high and the bias is low. Figures 2.3 (b), 2.3 (c), and 2.3 (d) show examples of underfit model, good fit model, and overfit model respectively.

trees through bagging and taking the random sub-sample of the feature space de-correlates their predictions and leads to a more robust model. This is because, training a decision tree on different subsets of the training data would make the trees less correlated with each other. This would increase the bias by keeping the variance much lower than the variance of the individual trees, which would compensate for the increase in bias, thus leading to better predictions as each decision tree in a forest would provide different predictions due to the variation in the trees.

When referring to Random Forests in this section, we precisely referred to the Breiman-Random Forest (Breiman-RF) (Breiman, 2001), which uses a combination of bagging and random sub-sampling of the feature space. There is another popular variant of Random Forest called Extremely Randomized Trees (ERT) (Geurts et al., 2006). ERT chooses a random sub-sample p of the feature space and then chooses a single split location for each of the p features randomly, which is unlike Breiman-RF. And unlike Breiman-RF, ERT utilizes the entire dataset and does not use bagging. We will discuss these techniques further in Chapter 4 and 5. Although this chapter has

focused on decision trees and ensemble variants, it is important to situate these methods within the broader landscape of machine learning research.

2.5.3 Baseline machine learning methods

To evaluate the performance of our proposed techniques in chapters $3 - 5$, we compare it against several widely used classical machine learning algorithms. These methods were selected because they represent strong linear and non-linear baselines and are commonly used in both classification and regression tasks. Decision tree-based baselines such as CART and Random Forests are described in detail in this chapter, chapters 3 and 4. In this section, we briefly summarize the remaining baseline models.

2.5.3.1 Support Vector Machines (SVM)

Support Vector Machines technique is a supervised learning model that constructs a decision boundary by maximizing the margin between classes in the feature space (Cortes and Vapnik, 1995). In the linear case, SVMs learn a hyperplane of the form $\mathbf{w}^\top \mathbf{x} + b = 0$ that separates the data with maximum margin. For non-linearly separable data, SVMs employ kernel functions to implicitly map the data into a higher-dimensional feature space where a linear separation is possible. In our experiments, SVMs serve as a strong discriminative baseline for comparison with the oblique decision boundaries produced by Gaussian Oblique Decision Tree (GODT) Technique.

2.5.3.2 Linear Discriminant Analysis (LDA)

Linear Discriminant Analysis is a classification method that models each class as a Gaussian distribution with a class-specific mean but a shared covariance matrix across classes (Izenman, 2013). Under this assumption, the resulting decision boundary between classes is linear. LDA is closely related to the linear Bayes classifier and provides a natural baseline for GODT, as both methods produce linear decision boundaries derived from Gaussian assumptions.

2.5.3.3 Quadratic Discriminant Analysis (QDA)

Quadratic Discriminant Analysis extends LDA by allowing each class to have its own covariance matrix (Friedman et al., 2001). This relaxation leads to quadratic decision boundaries, enabling QDA to model more complex class distributions at the cost of

increased model complexity and sensitivity to estimation error. QDA is included as a baseline to assess whether the additional flexibility of class-specific covariance structures provides an advantage over the shared-covariance assumption used in GODT.

2.5.4 Broader Machine Learning Context

Machine learning has a broader spectrum of methods that aims to infer patterns from data and make predictions. Traditional methods include supervised learning, unsupervised learning, and reinforcement learning, each involving different mechanisms for learning from labelled and unlabelled data (Bishop, 2006a). Beyond classical statistical learning techniques, modern approaches also integrate elements of probabilistic modelling, optimization and representation learning, thereby creating a rich and rapidly evolving ecosystem of methodologies. Despite the diversity, a fundamental challenge shared across many learning strategies concerns how models balance predictive performance, computational cost, and the ability to generalize well across diverse problem settings.

In the past decade, the rise of deep learning has significantly shaped the field of machine learning. Architectures such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and transformer models have achieved state-of-the-art performance in areas including computer vision, speech recognition, and natural language processing (Goodfellow et al., 2016; LeCun et al., 2015). These models learn highly complex non-linear representations from large datasets and, aided by advances in computational hardware, have become dominant in many AI applications. However, these deep architectures often behave as black-box models, making it difficult to interpret the learned decision logic or guarantee reliable behaviour in a high-risk or a highly regulated environments.

In contrast, interpretable models including decision trees and rule-based systems remain crucial in domains where transparency, accountability, fairness, and regulatory compliance are non-negotiable. Industries such as healthcare, finance, cyber security, and law frequently require models whose decision boundaries can be understood, audited, and justified to practitioners and stakeholders (Rudin, 2019). Unlike deep learning systems, decision trees yield explicit conditional statements that directly map features to predictions, enabling trust and clear communication between model developers and end-users. Therefore, even as deep learning continues to advance performance frontiers, research into interpretable oblique decision trees and their ensemble extensions provides solutions for critical real-world scenarios.

Neural network models were not included in the experimental comparison in this thesis. The focus of this work is on interpretable, geometry-driven models with

explicit decision boundaries, which allows for a more direct comparison with the proposed oblique decision tree framework. The inclusion of neural network-based models and their systematic comparison with GODT is identified as a direction for future works in Chapter 6.

Chapter 3

Gaussian Mixture Model Oblique Decision Trees for Classification and Regression

3.1 Introduction

The decision tree is a hierarchical technique that has been widely popular for over four decades, due to the simplicity in its implementation and interpretation. Some examples of applications include computer vision (Nowozin et al., 2011), information retrieval (Averbuch et al., 2004), intrusion detection (Benferhat et al., 2013), medical diagnosis (Pereira et al., 2017), signal processing (Vanli et al., 2019), computational biology, bio-informatics (Chu and Li, 2014; Remita et al., 2017), ecological data analysis (De'ath and Fabricius, 2000), image processing (Buluswar and Draper, 1994), remote sensing (Lagacherie and Holmes, 1997), and text classification (Joachims, 1998; Pang et al., 2015; Sebastiani, 2002; Silva et al., 2017). In this chapter, we will look at oblique decision trees, and how to use Gaussian mixture models to implement a computationally cheaper and accurate oblique decision tree for classification and regression.

The decision tree technique implements recursive partitioning of the input space into much smaller subsets which are pure (i.e., data corresponding to a single class). This is achieved by optimizing an impurity measure. This would then secure a constructed tree that would result in good prediction performance when presented with the test data. The type of impurity measure used differs depending on the decision tree technique used. We will discuss this further later in this section. The decision tree technique is mainly categorized into univariate (axis-parallel or orthogonal) and multivariate (oblique) decision trees based on the different ways in which we could split the feature space. In the rest of the thesis, we refer to univariate decision tree

techniques as axis-parallel and multivariate as oblique. We prefer the term oblique, as the word multivariate would include non-linear combination of feature variables.

The *axis-parallel decision tree* is a type of decision tree where at each non-terminal node, only a single feature is considered for the split rule that would lead to the axis-parallel split. In this axis-parallel setting, we assign a split rule to each non-terminal node, which is a binary-valued function of the feature vector. When the data in a node correspond to a single class or most of the nodes corresponds to a single class, the class label of the majority class is assigned to the terminal node, also known as the leaf node. In the axis parallel decision tree setting, at each non-terminal node, the best feature to split on and the best value to split on is decided based on an impurity measure.

Some popular and widely used examples of the axis-parallel decision tree techniques include Iterative Dichotomiser 3 (ID3) (Quinlan, 1986), Classification And Regression Trees (CART) (Breiman et al., 1984b), and C4.5 (Quinlan, 2014). The impurity measure used for a classification task for ID3, CART, and C4.5 are Information gain, Gini index, and Gain ratio respectively, which are used to greedily optimize the features to split on and the values to split on. To build regression trees, we either use mean squared error or variance reduction. At each non-terminal node, we need to first look at all the potential features to split on and the potential values to split on using the feature space. Then we utilize the stated impurity measures to calculate the best splitting features and values iteratively. The best impurity score that we can achieve is zero, which suggests that the non-terminal nodes are pure i.e., have data observations of the same class. This would result in the respective node being converted to leaf node. In the majority of the axis-parallel decision tree techniques, the aim is to optimize the impurity measure, where the best orthogonal split is decided from a pool of splits based on the impurity measure.

When we are presented with a new input vector, to perform classification using a decision tree, we begin at the root node, and work our way down the non-terminal nodes based on the split rule at each non-terminal node, until we arrive at a leaf node. In the axis-parallel case, we conduct a test to see if the input $x_i < \theta, x_i \geq \theta$, where the x_i is a corresponding feature for $i = 1, ..., p$, and θ is a constant threshold. The feature vector follows the hierarchy of the tree until it reaches the leaf node, where we would use the class label of the leaf node to assign the class to the feature vector. Figure 3.1 shows an example of this technique, when the corresponding partition creates a 2-D attribute space. Figure 2.1 (A) shows a top-down representation, and Figure 2.1 (B) shows a block representation.

Many varieties of axis-parallel techniques look at the decision trees from a single attribute point of view. As discussed earlier, Breiman et al. (1984c); Quinlan (1986, 2014) introduced the techniques CART, ID3, and C4.5 that construct a decision tree by

evaluating one feature at a time. At every non-terminal (non-leaf) node, every feature is considered individually, to see if the quality of the split is improved. These classical axis-parallel decision tree techniques have motivated some further work in this area that looks at different ways to split the nodes, and compare between different impurity criteria. Abuzaid et al. (2016) proposes a new way of splitting a decision tree when constructing a deep and large decision tree and tree-based ensembles to manage high computational needs. The paper proposes to partition a decision tree on vertical partitioning (i.e., partitioning by feature rather than the data), with a set of optimized data structures to reduce the CPU and computational cost during training. So, at every node, one has a subset of features rather than the data instances. Similarly, the random forest technique (Breiman, 2001), which unlike the standard decision tree techniques, takes a random subset of the feature space along with bootstrapping to induce randomness.

Similarly, the Extremely Randomized Tree (ERT) (Geurts et al., 2006) technique, which like the random forest technique, selects a random subset of the feature space, but instead for every selected feature, selects the split point at random to induce higher randomness and build diverse trees. On the other hand, we have work proposed by Fan (2016) which looks at new ways to estimate the generalization error decision tree classifiers that are statistically more accurate, robust, and more efficient than the popular cross validation technique. There are also techniques like Quick, Unbiased, Efficient, Statistical Trees (QUEST) (Loh and Shih, 1997) that introduce new split selection methods to address the issue of bias in standard decision tree algorithms. In standard decision tree techniques, there is bias towards features with more potential splits. Therefore, the author proposes a strategy that select the best splitting variable and point using statistical tests. On the other hand, Zhou and Feng (2017) generates an ensemble of decision trees with the characteristics of a deep neural networks, mainly considering the layer-by-layer processing, in-model feature transformation, and sufficient model complexity.

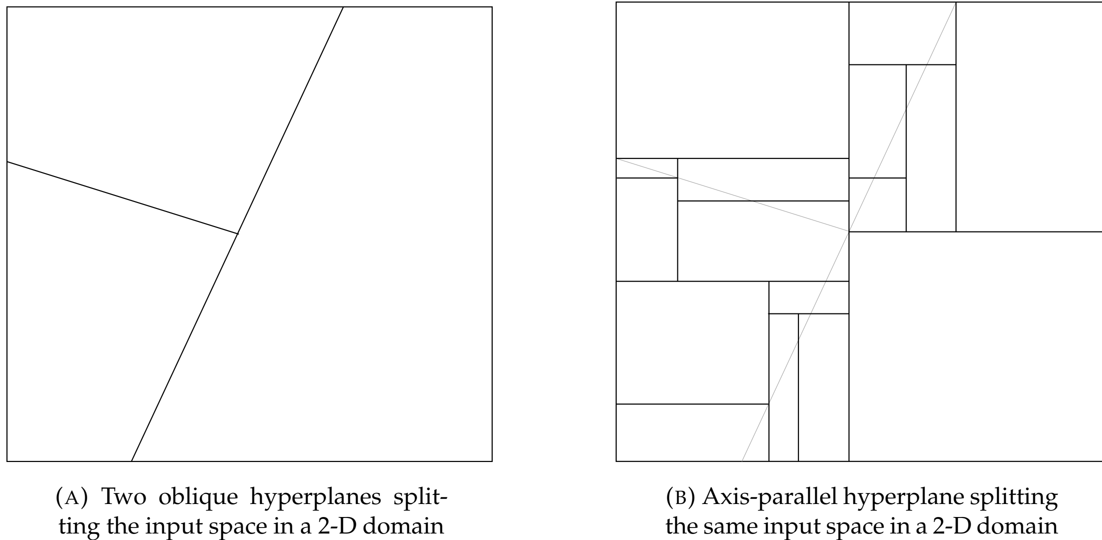
There are standard axis-parallel decision tree techniques that are dependent on frequency-based heuristic measure (Wang et al., 2020) in the literature. However, there is work in this area which focuses solely on devising new split conditions. Wang et al. (2014) proposes a measure that is not simply frequency-based, but is a combination of frequency and segment based heuristic measure to replace the existing only-frequency-based heuristic measure to split the non-terminal nodes of a decision tree. This work can differentiate the scenarios with same frequency information by taking the class permutation into consideration. The authors in Chandra and Varghese (2009); Chandra et al. (2010) propose splitting criteria that are better than those already existing in an aim to construct compact decision trees. This new split criteria would lead to less number of distinct classes in the sub trees after a split of the node. As Manwani and Sastry (2011) suggests, the hyperplane that we arrive at may not always

be the final decision boundary. Therefore, the process of splitting needs to be repeated iteratively until we arrive at a set of nodes with the highest purity. Hence, the need for a good splitting criteria that efficiently splits the node leading to both small and pure decision trees.

However, looking at all the features collectively would yield us a better understanding of the hidden relationships between them. As Yıldız and Alpaydın (2012) suggests, axis-parallel decision tree using a single feature to split a space with an orthogonal boundary would result in larger trees and poor generalization. Moreover, Brodley and Utgoff (1992) insists that axis-parallel decision tree splits using a single feature would restrict the split through the instance space that is orthogonal to the axis. Breiman et al. (1984c); Brodley and Utgoff (1992); Utgoff and Brodley (1990) suggest that this bias would be unsuitable for situations where the input variables are related (or correlated).

In contrast to the axis-parallel setting, in an oblique decision tree, at each non-terminal node a linear hyperplane (rather than an orthogonal hyperplane) separates the classes on the decision tree in an aim to improve the purity of the nodes incrementally at the subsequent nodes. In the oblique decision tree scenario, we have multiple features participating during the node splitting, unlike its axis-parallel counterpart. Rather than selecting one best feature, we select the best linear combination of features. The linear combination consists of multiplication of weights with the attributes. Even though the axis-parallel decision tree has its advantages, when we are presented with data that is correlated, we would prefer having a hyperplane splitting the data due to its better separability rather than a step-function.

Murthy et al. (1994) shows in their contributions that when the underlying problem requires oblique splits, using axis-parallel splits would lead to less accurate trees. Moreover, the axis-parallel trees can provide us with complex tree structures and increased computational cost, when the inherent decision boundaries are not axis-parallel (Yang et al., 2019). As illustrated in Figure 3.2, the axis-parallel decision tree provides the same class separation to an oblique decision tree, but in the axis-parallel case, the decision boundary is represented as steps, where each orthogonal split to the axis increases the depth of the trees. Whereas, in the oblique case, the decision boundary is represented as a straight line (or a linear hyperplane), where a similar separability can be achieved quicker (i.e., in less number of splits), potentially leading to quicker computational times to construct a tree. However, even though oblique decision trees potentially have lower number of splits as compared to the axis-parallel counterpart, literature in this area suggests that calculating the optimized hyperplane takes longer, leading to an overall higher computational time (Murthy et al., 1994; Norouzi et al., 2015). Furthermore, Yang et al. (2019) also suggests that oblique decision trees generally produce smaller trees with better performance, but highlight the increase in computational cost. This is another issue with



(A) Two oblique hyperplanes splitting the input space in a 2-D domain

(B) Axis-parallel hyperplane splitting the same input space in a 2-D domain

FIGURE 3.1: Oblique hyperplane and axis-parallel hyperplanes splitting the sample input space (Murthy et al., 1994)

optimization-based oblique decision tree techniques, where there is complexity involved in solving the optimization problem to construct the best hyperplane. As a result, an optimization-based oblique decision tree can only be constructed in reasonable time for small to medium-sized datasets where computational time increases as the data size increases (Bollwein and Westphal, 2022).

One of the first oblique decision tree techniques was proposed by Breiman et al. (1984c), which was a variant of the CART technique that constructs CART with linear combination, referred to as the CART-LC technique. The concept of the CART-LC technique is to maximize the goodness of split. However, there are some limitations to this technique. CART-LC is a deterministic technique, with no procedure to escape a local minima. CART-LC also produces only a single tree per dataset; there is no option to produce a different variant of the tree. CART-LC sometimes makes adjustments that increase the impurity of the split. Furthermore, there is no upper bound on the time spent at constructing a hyperplane at a node. Thus, these limitations prompted more research into this area, which proposed another technique called Oblique Classifier 1 (OC1) (Murthy et al., 1994) which is another early and widely known paper in this area. OC1 is an extension to CART-LC, by proposing to use random restarts and perturbations to escape the local minima. We will discuss further about OC1 in the Literature Review section.

Some of the other techniques proposed in this area include non-binary decision trees. Rather than constructing a decision tree with binary splits, a multi-way splitting decision tree called Linear Machine Decision Tree (LMDT) was proposed in Utgoff and Brodley (1991). Rather than using an univariate test for a decision node based on a heuristic measure like information gain, the author uses a linear model that works as a multivariate test that splits multi-way. Furthermore, the same author takes this work

further to discuss the things to consider when constructing multivariate decision trees along with discussing why a multivariate tree is better than a univariate tree (Brodley and Utgoff, 1995). The author suggests that axis-parallel decision trees are potentially larger in depth compared to oblique decision trees which may prevent concepts being represented concisely.

It is evident that techniques that use Support Vector Machines (SVMs) to construct the hyperplane of an oblique decision tree are amongst very popular techniques (Alamdar et al., 2019; Chen et al., 2011; Qi et al., 2013; Richhariya and Tanveer, 2018; Shao et al., 2012; Sharma et al., 2019). However, SVM-based techniques present the issue of multi-class problem as an SVM is not well suited for multi-class problems. Dietterich and Bakiri (1994); Hastie and Tibshirani (1998); Hsu and Lin (2002); Krebel (1999); Manwani and Sastry (2011); Platt et al. (1999); Subirats et al. (2010); Xu et al. (2013) use techniques like one-versus-one, one-versus-all, one-versus-one-versus-rest, majority-class-versus-rest, and other similar techniques to first convert the multi-class problem into a binary-class problem before constructing the decision tree. However, we have to note that adding an extra technique into the hierarchy of the decision tree, in addition to the existing SVM method, would lead to an increase in the computational cost of running the model (Ganaie et al., 2020).

There are a few approaches in the literature that use an SVM variant called multi-surface proximal SVM (MPSVM) (Mangasarian and Wild, 2005; Manwani and Sastry, 2011) to construct oblique decision trees for a two class problem. The aim here is to construct the bisector of two clustering hyperplanes with one hyperplane closest to one class and as far as possible from the data of the other class. The bisector constructed is based on an impurity measure. This is repeated iteratively to build an oblique decision tree. However, one drawback with the MPSVM technique (Mangasarian and Wild, 2005) is that it is only suitable for binary class classification. So, in order to overcome this issue, Manwani and Sastry (2011) combines the majority class into a single class, and the remaining class observations into another class before constructing the hyperplane. A further top-down approach called SVM-ODT (Barros et al., 2011; Menkovski et al., 2008) is proposed, which is an extension of C4.5 technique allowing for both univariate splits and oblique splits, generated using SVM. However, this technique still has the same issue that it cannot deal with a multi-class problem.

Twin-bound SVM (TBSVM) and variants of twin-bound SVM seem to be a popular technique widely used in the literature to construct ensembles of oblique decision trees. The idea behind a twin bound SVM is to construct two non-parallel hyperplanes, unlike the classical SVM, which constructs a single hyperplane. The aim is to have each class close to the respective hyperplane and further away from the other hyperplane. Alamdar et al. (2019); Chen et al. (2011); Qi et al. (2013); Richhariya and Tanveer (2018); Shao et al. (2012); Sharma et al. (2019) propose different variants

of TBSVMs. TBSVM solves two quadratic programming problems whereas MPSVM solves a liner programming problem.

There are also techniques that use supervised learning (Loh and Shih, 1997; Loh and Vanichsetakul, 1988; Yildiz and Alpaydin, 2005) during the node splitting process to build a multivariate decision tree. Furthermore, when exploring techniques to separate classes, discriminant analysis is a strong contender as the inherent behaviors of this technique is to separate the classes. So, techniques like Feature selection-based Classification Trees (FACT) (Loh and Vanichsetakul, 1988), LDT (Yildiz and Alpaydin, 2005), and QUEST (Loh and Shih, 1997) have been introduced that can be categorized into this area. However, multivariate decision tree techniques that use supervised techniques (López-Chau et al., 2013), like Fisher's discriminant analysis, would have issues when working in the multi-class setting. Fisher's discriminant analysis is designed for a binary problem and would again require techniques like one-vs-all to adapt it to the multi-class setting. This would make the technique slow and add to the overall computational burden.

As we discussed earlier, the time it takes to train an oblique decision tree is a major issue in this area. Moreover, as most techniques depend on the construction of SVMs, using additional procedures like one-vs-all to adapt the model to a multi-class setting adds significant additional runtime. In addition, we want to construct a binary tree model to retain model interpretability.

To address these issues, we propose the Gaussian Mixture Model Oblique Decision Tree (GODT) technique for classification and regression, that incorporates the unsupervised Gaussian Mixture Model (GMM) technique to construct an oblique decision tree. We use the clusters that are identified using GMM to generate linear decision boundaries that split the input data at each non-terminal node. We are motivated here by Extremely Randomized Trees, Linear Discriminant Analysis, and Gaussian Mixture Models. See Friedman et al. (2001) for more details on these techniques. Our GODT method is built for real-valued attributes but also works well with categorical attributes.

As our computational results show, this algorithm exhibits good classification accuracy when compared with existing counterparts, and has computational times close to its axis-parallel counterparts in most scenarios. We also focus on the computational times, as training an oblique decision tree generally comes with high running costs. We also note that some optimization problems that need to be solved within these methods are NP-hard (Murthy et al., 1993; Norouzi et al., 2015). NP-hard refers to a class of computational problems for which no known algorithm can solve all instances efficiently (in polynomial time). This means that finding an optimal solution is computationally expensive, especially as the problem size grows. Many

optimization problems involved in training oblique decision trees fall into this category, making them computationally challenging

Our proposed algorithm is suited to perform both binary or multi-class classification. When it comes to the technique that we propose, there is a potential to extended our work to non-binary splitting. However, in the thesis, we focus on binary splitting as we have a computationally cheaper way to construct a single hyperplane. Our contributions in this chapter include the following:

1. We propose a novel technique that uses GMMs to build an oblique decision tree.
2. The use of GMM constructs a hyperplane fairly quickly which is generally considered as a task with high computational time, and in some cases even NP-hard.
3. We show that this algorithm has high accuracy, and performs either similar to or better than existing established techniques in almost all the cases that we have tested.
4. We demonstrate the performance of the algorithm on a mix of high-dimensional and large-scale datasets, to display the efficiency and versatility of the algorithm.

3.2 Related work

In this section, we discuss the details of CART (Breiman et al., 1984c) along with a popular and widely known technique on oblique decision trees. We discuss tree-based techniques that we use as comparison in the experiments section. We look at work proposed by Murthy et al. (1994) that introduces a new system for the construction of oblique decision trees (OC1) that combines deterministic hill-climbing with two forms of randomization to find a good oblique split. Wang et al. (2020) proposes another multivariate decision tree technique that uses K-means clustering to construct linear hyperplanes. However, a larger emphasis is placed on class imbalance in this work. As we do not focus on class imbalance in this thesis, we do not discuss this work proposed by Wang et al. (2020) any further.

3.2.1 CART

In previous sections, we have mentioned different axis-parallel decision trees like ID3, CART, and C4.5 that are amongst popular techniques. In the experiments section, we work with categorical, numerical and attributes which are a mix of both. We know that ID3 can only handle categorical values. So, even though CART and C4.5 can both

handle categorical and numerical attributes, we proceed with CART, as it can handle outliers, and is the basis for the Standard Random Forest which we will look in Chapter 4 and Chapter 5. So, in this subsection, we will discuss the process of constructing a CART.

CART is used to construct binary decision trees, that can be implemented on both classification and regression problems. The idea behind a classification and regression tree is to perform recursive splitting of the input space depending on a splitting criteria which in this case is Gini impurity for classification and mean squared error for regression.

The CART technique has two main steps in the process of building the tree:

- Selecting the best feature to split on
- The best threshold value to split on

3.2.1.1 Selecting the best node

In the process of constructing a decision tree using the CART algorithm, when we arrive at a node, the CART technique checks for all the possible attributes to split on and selects the attribute that has the highest reduction in impurity for classification or mean squared error for regression.

The CART technique uses Gini impurity as the splitting criterion:

$$\text{Gini}(k) = 1 - \sum_{i=1}^{c} p_i^2, \tag{3.1}$$

where c is the number of unique classes and p_i is the proportion of observations in class i at node k

3.2.1.2 Selecting the best threshold value

In the CART algorithm, both the feature and its corresponding split threshold are selected jointly by evaluating all possible feature-threshold combinations and choosing the one that produces the two child nodes with the highest purity. This corresponds to minimizing the Gini impurity, ideally achieving a value of 0, which indicates a pure node.

For categorical features, the goal is to find a split that results in child nodes containing samples from a single class. In contrast, for regression trees, the split is chosen to minimize the mean squared error of the resulting nodes.

3.2.2 Oblique Classifier 1

CART with Linear Combination, commonly known as CART-LC was the first ever oblique decision tree algorithm to be proposed (Breiman et al., 1984a). Oblique Classifier 1 (OC1) is a new system for the induction of oblique decision trees was devised by Murthy et al. (1994) which is an extension of CART-LC, to address its limitations. This technique combines the use of deterministic hill-climbing with two different forms of randomization to find an optimal hyperplane at each non-terminal node of a decision tree. OC1 mostly uses deterministic hill-climbing to have a good computational efficiency, and then uses randomization to avoid local minima.

Below, we briefly discuss the details of the OC1 induction, i.e., the technique for finding the coefficients of the hyperplane at each non-terminal node and the method for computing the impurity of a hyperplane.

3.2.2.1 Perturbation algorithm

The OC1 technique utilizes a randomized perturbation algorithm to find the hyperplane that splits a given input space into non-overlapping subsets. In order to make sure that this technique's performance is at least as similar as the axis-parallel counterpart, OC1 uses an oblique split only when it improves the performance over the axis-parallel split. So, this technique first finds the best axis-parallel split at each non terminal node, before implementing an oblique split only when there is evidence for the oblique split providing better separability of the axis-parallel split.

There are a large number of ways to partition data using a hyperplane, so finding every possible hyperplane would be computationally expensive. Hence, the possible hyperplanes' search strategy for this technique is perturbing the current hyperplane to a new location. Now, let the equation of the current hyperplane H at a non-terminal node be $\sum_{i=1}^{p} a_i x_i + a_{p+1}$. If a point $D_j = (\mathbf{x}_j, y_j)$ is substituted into the equation, we will have $\sum_{i=1}^{p} a_i x_j + a_{p+1} = V_j$ for coefficient a_i and feature vector x_i for $i = 1, ..., p$. Here, V_j informs us on whether a data point D_j is above or below a hyperplane H. The idea behind this step of OC1 is to adjust the coefficients of the hyperplane H one-at-a-time to find a locally optimal value. This is how OC1 perturbs a hyperplane.

If through perturbation, OC1 finds a better hyperplane H_1 where H_1 has a better impurity i.e., lower than H, the hyperplane H is then discarded or H_1 is discarded for vise-versa. Now, if H and H_1 have identical impurities, H is still discarded and the technique proceeds with H_1. This is the perturbation step for the OC1 technique.

3.2.2.2 Randomization

When the split reaches a local minimum of the impurity measure, the perturbation algorithm pauses. When a perturbation of any single coefficient of a current hyperplane does not result in the decrease of the impurity measure, this suggests that a local minimum has been found for the OC1's search space. There is also the possibility that a potential local minimum can be a global minimum. The OC1 algorithm implements two ways of escaping the local minimum:

1. perturbing the hyperplane with a random vector
2. re-starting the perturbation algorithm with a different random initial hyperplane

The way perturbing the hyperplane with a random vector works is by choosing a random vector and adding it to the coefficients of the current hyperplane when the technique reaches a local minimum. Then the optimal amount by which the perturbation occurs in the random direction is calculated.

The other method where the local minima is avoided depends on the idea of performing multiple local searches. The scenarios where random perturbation does not escape the local minima, Murthy et al. (1994) proposes to just restart with a new initial hyperplane. When neither the deterministic local search nor the random jumps does not find a better split, the restarts end.

3.2.2.3 Impurity Measure

The main aim for the OC1 algorithm is to split the $n-$dimensional attribute space into regions that contain data of the same class. A new non-terminal node is added during the splitting process to minimize the impurity of the training data. The OC1 technique is inherently designed to work with a large number of impurity measures, such as:

- Information Gain Quinlan (1986)
- The Gini Index Breiman et al. (1984b)
- The Twoing Rule Breiman et al. (1984b)
- Max Minority Murthy et al. (1993)
- Sum Minority Murthy et al. (1993)
- Sum of the Variances Murthy et al. (1993)

The impurity measure used would depend on the type of dataset used. This is dependent on different factors like the type of target variable i.e., categorical or continuous, class distribution and also the properties desired in a decision tree.

When we work on categorical target variables, information gain or the Gini index are popular criteria. The Twoing rule is commonly used for classification trees to encourage more balanced class proportions on each side of the binary splits. The max minority and sum minority are designed to enhance handling of imbalanced datasets. And finally, the sum of variance is suitable for regression problems where minimizing the variance within nodes leads to better predictions.

3.2.3 Soft vs. Hard Clustering: Gaussian Mixture Model and K-means

When we look at clustering techniques, we come across two common approaches which are hard clustering and soft clustering techniques. A popular example of a hard clustering technique is K-means, which is a deterministic approach, whereas a popular example of a soft clustering technique is GMM, which is probabilistic. The way in which these techniques vary depends on how data points are assigned to the clusters and how flexible the clusters are.

3.2.3.1 Hard Clustering using K-means

K-means is a widely used clustering technique due to its computational efficiency, simplicity, and speed. In K-means, each data point is assigned to the one cluster whose centroid is closest to it, so that the total (within-cluster) sum of squared distances between points and their own centroid is as small as possible. Until convergence, the data points are reassigned to each cluster by iteratively updating the centroids. Figure 3.2 shows an example of the comparison of the K-means clusters (Figure 3.2 (A)) and GMM clusters (Figure 3.2 (B)).

- **Hard assignment process:** At every iteration, the K-means technique assigns each data point to the closest cluster using the Euclidean distance to the centroid of the cluster. After assigning the data point, the data belongs to the cluster without the use of any uncertainty measures
- **Advantages:** K-means is an efficient, simple, and fast technique to implement. In general, K-means, due to the hard cluster assignment, is ideal for datasets with well-separated and spherical clusters. Moreover, when the clusters have similar density and size, K-means is a well-suited contender.

- **Limitations:** As each data point is allocated to only a single cluster, K-means may find it difficult with overlapping clusters, which generally is the case with complex datasets, and data points may not be easily separated.

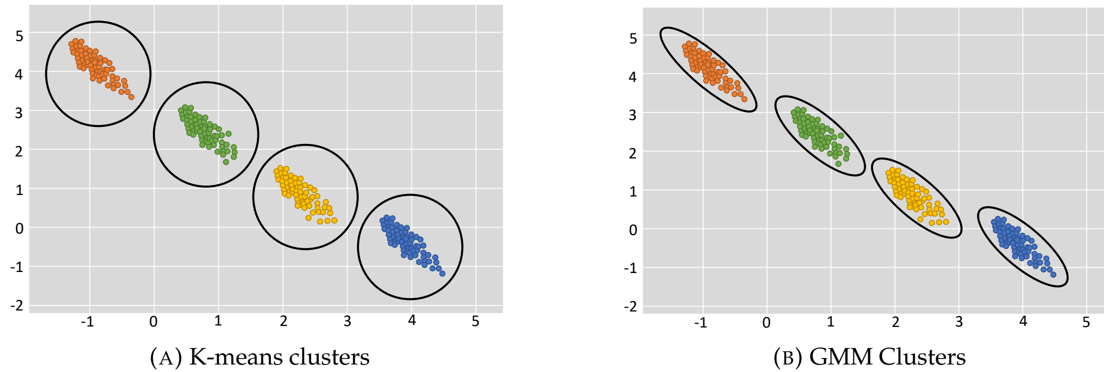
3.2.3.2 Soft Clustering using GMM

Unlike deterministic approaches like K-means which performs hard clustering, GMM is a probabilistic approach that performs soft clustering. In soft clustering, a data point can be part of multiple clusters with different probabilities. In the context of GMM, the data is modelled as a combination of Gaussian distributions, where each Gaussian represents a cluster. Rather than a single deterministic assignment in K-means, in a GMM, the data points are assigned a probability of corresponding to each cluster.

- **Soft assignment process:** The probability that each data point corresponds to each Gaussian component depending on the data's likelihood under each component, which is calculated by the GMM using Expectation-Maximization (EM). The clusters are then refined iteratively by using the probabilities to update the mean, covariance, and mixing coefficient parameters. The mixture (or prior) components represent the probability that a randomly selected data observation comes from the jth component, before even observing any data, thus has the term 'prior'.
- **Advantages:** One of the biggest advantages in soft clustering techniques is the flexibility when working with complex data. When we use the GMM, this will also account for the variance in the data and this is done using the covariance matrices. The component covariance matrices determines the variability and the orientation of the data. Geometrically, the eigenvalues and eigenvectors of the covariance structure can be used to identify the shape of the cluster ellipsoid. The eigenvalues provides us the lengths of the axes of the ellipsoid and the eigenvectors provide the orientation.
- **Limitations:** In comparison to K-means, GMMs are computationally intensive due to the EM algorithm, mainly in high-dimensional examples. GMMs need to be carefully initialized to avoid poor convergence to local optima. Furthermore, having too many components may lead to GMMs overfitting the data.

3.3 The proposed method

In this section we introduce a novel Gaussian Mixture Model Oblique Decision Tree (GODT) technique for classification and regression. The major difference between



(A) K-means clusters

(B) GMM Clusters

FIGURE 3.2: K-means and GMM cluster comparison

existing oblique decision tree techniques and our technique is that our method utilizes Gaussian mixture models at each non-terminal node to build soft clusters of two components to replicate a binary tree structure. We purely focus on creating sub-spaces of the initial data using Gaussian mixture models. In the literature, most of the oblique decision tree techniques try to build a hyperplane and then optimize the parameters defining the hyperplane to best separate the classes. The limitations in these techniques prompted us to build the oblique decision tree using GMM. Our aim is to build an oblique decision tree technique that has a high accuracy, along with low computational times. The details of this model are stated in Algorithm 5 along with the model architecture and an illustration of the GODT.

3.3.1 Gaussian Mixture Model

A GMM is a parametric probability density function represented as a weighted sum of Gaussian component densities (Reynolds, 2009). It is used to represent normally distributed sub-populations amongst a larger population. The idea behind mixture models is that they do not require any information on the association of the data with the sub-population, letting the model learn the sub-populations in an automatic way. So, this would categorize this as a form of unsupervised learning techniques. The major applications of GMMs include areas where the mixture components and means of the GMM are used to identify location. Let Z follow a mixture of Normal distributions with mean $\boldsymbol{\mu}_j$, covariance $\boldsymbol{\Sigma}_j$, and component weights ϕ_j for the multi-dimensional model for $j = 1, ..., M$ where M corresponds to the number of components.

$$
\begin{aligned}
p(\mathbf{z}) &= \sum_{j=1}^{M} \phi_j N(\mathbf{z}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) \\
N(\mathbf{z}|\boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j) &= \frac{1}{\sqrt{(2\pi)^P |\boldsymbol{\Sigma}_j|}} \exp\left(-\frac{1}{2}(\mathbf{z}-\boldsymbol{\mu}_j)^\top \boldsymbol{\Sigma}_j^{-1}(\mathbf{z}-\boldsymbol{\mu}_j)\right) \\
\sum_{j=1}^{M} \phi_j &= 1, \quad \phi_j \geq 0 \text{ for all } j.
\end{aligned} \tag{3.2}
$$

The reason we describe the above is because, for our technique, we use a GMM. So, we state (3.2) to define the notation. We use the iterative Expectation-Maximization (EM) algorithm to estimate the model parameters of the algorithm. Please refer to Bishop (2006a) for more details on GMM and EM.

Looking at Algorithm 5 and also at Figure 3.3, we can observe that the GODT technique is a binary decision tree technique that implements GMM at each non-terminal node, to construct two soft clusters. Referring back to equation (3.2), based on the Algorithm 5, for our method with $M = 2$ we have two mean vectors $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, we have a shared covariance matrix so $\boldsymbol{\Sigma}$ across both components, and we have ϕ_1 and ϕ_2 which describe the prior probabilities of the components.

3.3.2 Calibration of the Gaussian Mixture Model

At each non-terminal node of the tree, the samples assigned to that node are represented by a feature matrix

$$
\mathbf{X} = \{\mathbf{x}_i\}_{i=1}^{m}, \qquad \mathbf{x}_i \in \mathbb{R}^p,
$$

where m denotes the number of samples reaching the node. To obtain a probabilistic partition of the data, we fit a two-component Gaussian mixture model (GMM) of the form

$$
p(\mathbf{x}) = \sum_{j=1}^{2} \phi_j \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}),
$$

where ϕ_j are the mixture weights (priors), $\boldsymbol{\mu}_j$ are the component means, and $\boldsymbol{\Sigma}$ is a shared diagonal covariance matrix. The term "prior" refers to the mixture weights ϕ_j (mixing proportions).

3.3.2.1 Parameter estimation

In GODT we use $j = 2$ mixture components at each non-terminal node to induce a binary split. The GMM parameters $\{\phi_j, \boldsymbol{\mu}_j, \boldsymbol{\Sigma}\}$ are estimated by maximum-likelihood

estimation using the Expectation-Maximization (EM) algorithm, as implemented in MATLAB's standard GMM routine (Bishop, 2006b; Dempster et al., 1977). The EM algorithm maximizes the marginal log-likelihood of the observed data,

$$\mathcal{L} = \sum_{i=1}^{m} \log \left(\sum_{j=1}^{M} \phi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}) \right).$$

In the E-step, posterior responsibilities

$$\gamma_{ij} = p(z_i = j \mid \mathbf{x}_i),$$

are computed for each mixture component. Using Bayes' rule, the responsibilities are given by

$$\gamma_{ij} = \frac{\phi_j \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma})}{\sum_{\ell=1}^{M} \phi_\ell \mathcal{N}(\mathbf{x}_i \mid \boldsymbol{\mu}_\ell, \boldsymbol{\Sigma})}.$$

In the M-step, the mixture weights and means are updated as (Bishop, 2006b; McLachlan and Peel, 2000)

$$\phi_j = \frac{1}{m} \sum_{i=1}^{m} \gamma_{ij}, \qquad \boldsymbol{\mu}_j = \frac{\sum_{i=1}^{m} \gamma_{ij} \mathbf{x}_i}{\sum_{i=1}^{m} \gamma_{ij}}.$$

The covariance matrix is constrained to be diagonal and shared across components,

$$\boldsymbol{\Sigma} = \operatorname{diag}(\sigma^2),$$

which corresponds to a pooled variance estimate under a conditional independence assumption. To ensure numerical stability and to prevent degenerate solutions, a regularization parameter $\beta > 0$ is added to each diagonal entry of the covariance matrix, yielding (Bishop, 2006b; McLachlan and Peel, 2000)

$$\boldsymbol{\Sigma} \leftarrow \boldsymbol{\Sigma} + \beta \mathbf{I}.$$

The shared diagonal covariance is estimated using a responsibility-weighted pooled variance:

$$\boldsymbol{\Sigma} = \operatorname{diag}(\sigma^2), \qquad \sigma_j^2 = \frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{M} \gamma_{ik} (x_{ik} - \mu_{jk})^2, \quad k = 1, \ldots, p.$$

Since $\sum_{j=1}^{M} \gamma_{ij} = 1$ for each data point $\mathbf{x}_i$, we have $\sum_{i=1}^{m} \sum_{j=1}^{M} \gamma_{ij} = m$, yielding the pooled variance estimate above. All GMM parameters are estimated using MATLAB's `fitgmdist` function, with `CovarianceType` set to `diagonal`, `SharedCovariance` set to `true`, and `RegularizationValue` set to β to ensure numerical stability (MathWorks, a,b,c).

3.3.2.2 Initialization

To reduce sensitivity to initialization, we initialize the two component means in the EM algorithm using K-means clustering. The K-means centroids are obtained using the PCA-based procedure in Algorithm 6. Details and motivation are provided in Section 3.3.6 and 3.3.7 which looks at initialization in detail and Algorithm 6 in detail.

3.3.2.3 Induced linear split

Under the assumption of two Gaussian components with a shared covariance matrix, the optimal decision boundary is linear. Accordingly, the node split is defined by a hyperplane

$$\mathbf{w}^\top \mathbf{x} = d,$$

where

$$\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2).$$

Because $\boldsymbol{\Sigma}$ is diagonal, this expression reduces to elementwise scaling of the mean difference by the inverse variances. This is explained in more detail in section 3.3.3.

3.3.3 Hyperplane Generation in GODT

When constructing an oblique decision tree, the hyperplanes are linear, allowing more flexibility in the decision tree structure. The axis-parallel decision trees use a single feature during construction, where on the contrary, the oblique decision trees use hyperplanes that are a linear combination of the features. This allows for the decision tree to model complex relationships within the data. Below, we look at the hyperplane generation for GODT for classification and regression.

3.3.3.1 Hyperplane Generation for Classification

When performing classification, the aim of Gaussian Oblique Decision Tree technique for Classification (GODT-C) is to find the hyperplane that best separates the classes in the data. The hyperplane is constructed with an aim to maximize the purity of data on either side of the decision boundary.

Algorithm 5 GODT(Dataset, Purity Condition, Regularization Parameter)

Input: Training data $\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$, task type (classification / regression), purity or error threshold δ, regularization parameter β, minimum leaf size $m_{\min}$

Output: Fully constructed decision tree $\mathbf{T}$, with cluster centroids $\boldsymbol{\mu}_1^{(j)}$, $\boldsymbol{\mu}_2^{(j)}$ and shared covariance matrices $\boldsymbol{\Sigma}^{(j)}$ at each level $j = 1, \ldots, k$

1: Initialise an empty tree $\mathbf{T}$ and an empty set of nodes Ψ
2: Assign all data in $\mathcal{D}$ to the *RootNode* and insert the *RootNode* into Ψ
3: **if** the *RootNode* satisfies the non-splitting condition **then**
4:     Convert the *RootNode* into a *LeafNode* and assign the majority class label (classification) or the mean target value (regression)
5:     **return T**
6: **else**
7:     **for** $i = 1$ to k **do** ▷ for a sufficiently large k
8:         Select a *Node* N from Ψ
9:         Let the feature matrix in N be $\mathbf{X} \in \mathbb{R}^{m\times p}$ and the corresponding labels be $\mathbf{y} \in \mathbb{R}^{m\times 1}$ for $m \subseteq n$
10:         **if** $m \leq m_{\min}$ **then**
11:             Convert N into a *LeafNode* and assign the majority class (classification) or mean target value (regression)
12:             **continue**
13:         **end if**
14:         Initialise the centroids for the Gaussian Mixture Model (GMM) using K-means on the data in N, where the initial K-means centroids are obtained via Algorithm 6
15:         Fit a 2-component Gaussian mixture model to $\mathbf{X}$ by maximum-likelihood estimation using the EM algorithm, with shared diagonal covariance matrix $\boldsymbol{\Sigma} = \mathrm{diag}(\boldsymbol{\sigma}^2)$ and regularization parameter β added to each diagonal entry
16:         Extract the estimated mixture weights π_1, π_2, component means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2$, and diagonal variances $\boldsymbol{\sigma}^2$
17:         **for** $j = 1, 2$ **do** ▷ for the two mixture components
18:             **if** the non-splitting condition is satisfied in the child region **then**
19:                 Convert the non-terminal *Node* into a *LeafNode* and assign the majority class label (classification) or the mean target value (regression)
20:             **else**
21:                 Add the non-terminal *Node* to Ψ
22:             **end if**
23:         **end for**
24:     **end for**
25: **end if**

Algorithm 6 K-means initialization Wang et al. (2018)

Input: The matrix sample $\mathbf{X}_k \in \mathbb{R}^{k \times p}$ where $k \subset n$ and $\mathbf{X}_k$ are rows from $\mathbf{X}$

Output: The output will be the two values $\mu_1^{(0)}$ and $\mu_2^{(0)}$ which are two centroid values that can be used as initial cluster centroids.

1: We generate a hyperplane in the form of equation $\mathbf{w}^\top \mathbf{x} = d$

2: $\mathbf{w}$ is calculated using the eigenvector with the maximum eigenvalue as a result of PCA

3: $d = \text{median}\{\mathbf{w}^\top \mathbf{x} | \mathbf{x} \in \mathbf{X}\}$ is the midpoint of projections of the sample data

4: We split the data observation in the sample into $\mathbf{X}_a$ and $\mathbf{X}_b$ where $\mathbf{x} \in \mathbf{X}_a$ if $\mathbf{w}^\top \mathbf{x} > d$ or $\mathbf{x} \in \mathbf{X}_b$ otherwise

5: Now, $\boldsymbol{\mu}_i^{(0)} = \frac{1}{|\mathbf{X}_i|} \sum_{\mathbf{x} \in \mathbf{X}_i} \mathbf{x}, i = 1, 2.$

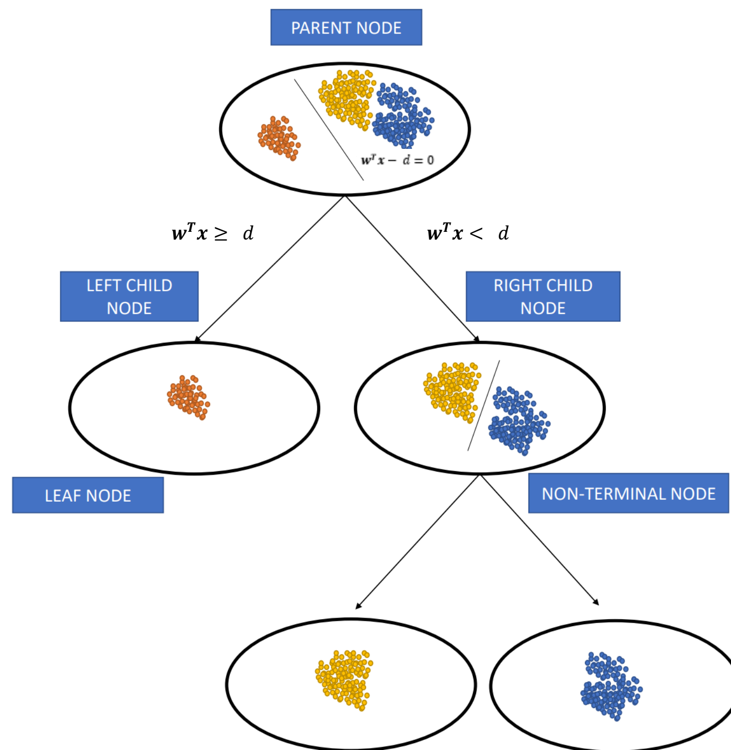


FIGURE 3.3: Illustration of linear hyperplane of the form $\mathbf{w}^\top \mathbf{x} - d = 0$ constructing a top-down oblique decision tree

Although the splitting strategy in GODT-C is unsupervised and does not directly optimize any class-based objective, it aims to generate partitions that are informative for classification. In particular, by identifying hyperplanes that separate the underlying structure of the data distribution, the resulting regions can, in practice, align with class boundaries and therefore improve prediction performance. The motivation can be summarized as follows:

- **Potential for Class Separation:** While class labels are not used when constructing the hyperplane, the unsupervised partitioning may separate clusters in the feature space that correspond strongly to class membership.
- **Improved Prediction Accuracy:** If the partitions align with label structure, the leaf regions of the tree will contain more homogeneous data, leading to improved classification performance.

Let us now look at the mathematical formulation of the hyperplane. In order to split the nodes at each non-terminal node, we would require a hyperplane. In classification, the hyperplane is generally represented in the following form:

$$\mathbf{w}^\top \mathbf{x} = d, \tag{3.3}$$

where:

- $\mathbf{w}$ is the normal vector to the hyperplane, which would define the orientation.
- d is the cut-point, that determines the position of the hyperplane and where the hyperplane separates the data.

Once we have the hyperplane, the aim is to assign data points to either side of the hyperplane based on their projections along $\mathbf{w}$. We use the following binary split criterion:

$$\mathbf{w}^\top \mathbf{x} \geq d \quad \text{or} \quad \mathbf{w}^\top \mathbf{x} < d. \tag{3.4}$$

Now, in order to construct this hyperplane, it is necessary to determine the normal direction $\mathbf{w}$ and the cut-point d. In order to calculate the value of $\mathbf{w}$, we will use the information that we acquire from executing the GMM at each non-terminal node. At each run, once the GMM is executed, this would output us the means of the two clusters, $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, along with the diagonal covariance matrix that is shared between the two components, $\boldsymbol{\Sigma}$. A shared covariance is when multiple GMM components can have the same covariance matrix across different classes or clusters. The shared covariance is computed as a weighted sum of the data covariances across all clusters. The shared covariance matrix would allow the clusters to have the same orientation. Using this information, we can calculate the normal direction of the hyperplane as (Hart et al., 2000, Chapter 2, Section 2.6.2):

$$\mathbf{w} = \boldsymbol{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2). \tag{3.5}$$

We use the transformation $\mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ rather than using $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ because the $\mathbf{\Sigma}^{-1}$ accounts for the spread and correlation of the features. The $(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$ is modified by the inverse covariance matrix $\mathbf{\Sigma}^{-1}$. The mean difference vector is reoriented and re-weighed based on the variability and correlation in the attribute set in order to highlight the direction with the maximum discriminability. Geometrically, this transformation ensures that the hyperplane is oriented to best separate the classes. This is based on the Mahalanobis distance rather than the Euclidean distance. This is illustrated in Figure 3.4. A similar geometric intuition is used when constructing the decision boundaries for linear discriminant analysis (Hart et al., 2000, Chapter 2, Section 2.6.2).

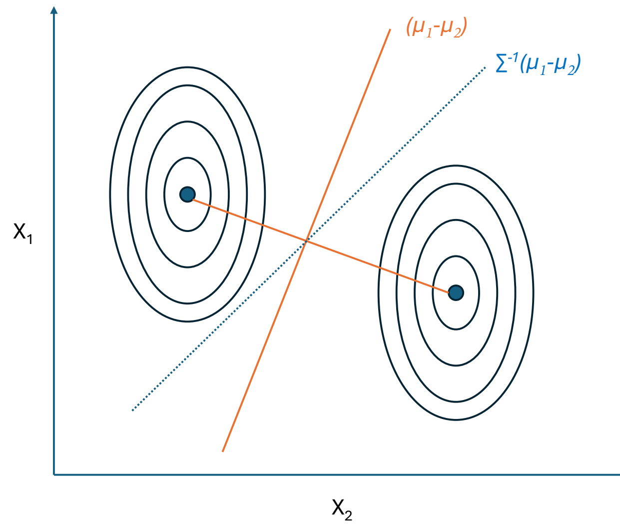


FIGURE 3.4: Comparison of hyperplanes for binary classification: The orange line represents the hyperplane $\mathbf{w}^\top \mathbf{x} - d = 0$ derived using the Euclidean distance ($\mathbf{w} = (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$). The blue line represents the hyperplane adjusted for covariance ($\mathbf{w} = \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$), incorporating the Mahalanobis distance), tilting to account for feature correlations and varying variances as shown by the deep blue covariance ellipsoid.

The hyperplane intersects the line connecting the two centroids $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ at a point $\mathbf{x}_0 \in \mathbb{R}^p$. We compute the boundary intersection point $\mathbf{x}_0$ as follows (Hart et al., 2000, Chapter 2, Section 2.6.2):

$$\mathbf{x}_0 = \frac{1}{2}(\boldsymbol{\mu}_1 + \boldsymbol{\mu}_2) - \frac{\ln(\phi_1/\phi_2)}{(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^\top \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2). \tag{3.6}$$

The scalar threshold in Eq. (3.3) is then $d = \mathbf{w}^\top \mathbf{x}_0$. If the prior probabilities of each component are equal, then the equation (3.6) simplifies to the mid-point between the two clusters. However, if we have unequal prior probabilities, the optimal decision boundary is displaced away from the more likely mean. So, in our case, the intersect is not a 50:50 split when $\phi_1 \neq \phi_2$.

When implementing GMM, the reason we use a diagonal covariance matrix compared to a full covariance matrix is because using a diagonal covariance matrix gives us a computational advantage due to the simpler matrix operations, efficient storage, and faster computations amongst others. We performed a small experiment where we implemented the GODT-C technique with full covariance and diagonal covariance. The performance is recorded in Table A.1 in Appendix A.1. We observe that using diagonal covariance provides us with either similar or better accuracy along with quicker computational times. For completeness, we now derive Eqs. (3.3–3.6).

3.3.3.2 Derivation of the linear hyperplane (Eqs. 3.3–3.6)

Assumptions: We now make explicit the construction of the linear split used at each non-terminal node. Let $\mathbf{x} \in \mathbb{R}^p$ denote a feature vector reaching the node and let $z \in \{1, 2\}$ denote the latent GMM component assignment. We assume that the node data are modelled by a two-component Gaussian mixture model with *shared* covariance,

$$p(\mathbf{x}) = \sum_{j=1}^{2} \phi_j \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}), \qquad \phi_j > 0, \sum_{j=1}^{2} \phi_j = 1, \tag{3.7}$$

where $\boldsymbol{\mu}_j \in \mathbb{R}^p$ are the component means and $\boldsymbol{\Sigma} \in \mathbb{R}^{p \times p}$ is a common covariance matrix. Throughout, we assume $\boldsymbol{\Sigma}$ is symmetric positive definite (hence invertible). In GODT we further restrict $\boldsymbol{\Sigma}$ to be diagonal and regularized as $\boldsymbol{\Sigma} \leftarrow \boldsymbol{\Sigma} + \beta \mathbf{I}$ with $\beta > 0$ to ensure numerical stability.

Bayes decision rule: Given the fitted GMM parameters $(\phi_1, \phi_2, \boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma})$, the Bayes-optimal assignment compares the posterior probabilities of the two components:

$$\text{assign } \mathbf{x} \text{ to component 1 if } p(z = 1 \mid \mathbf{x}) \geq p(z = 2 \mid \mathbf{x}). \tag{3.8}$$

Equivalently, the decision boundary is defined by the zero set of the log-posterior ratio,

$$\log \frac{p(z = 1 \mid \mathbf{x})}{p(z = 2 \mid \mathbf{x})} = 0. \tag{3.9}$$

Using Bayes' rule (Stone, 2013) and substituting the Gaussian densities gives

$$\log \frac{\phi_1}{\phi_2} - \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x} - \boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}^{-1}(\mathbf{x} - \boldsymbol{\mu}_2) = 0. \tag{3.10}$$

Because both components share the same covariance matrix, the quadratic terms in $\mathbf{x}$ cancel when expanding the two quadratic forms. Therefore the boundary is linear in $\mathbf{x}$

and can be written as a hyperplane of the form

$$\mathbf{w}^\top \mathbf{x} = d, \tag{3.11}$$

which corresponds to Eq. (3.3), with the binary split criterion

$$\mathbf{w}^\top \mathbf{x} \geq d \quad \text{or} \quad \mathbf{w}^\top \mathbf{x} < d, \tag{3.12}$$

corresponding to Eq. (3.4).

Normal vector (Eq. 3.5). Rearranging (3.10) yields the normal direction

$$\mathbf{w} = \mathbf{\Sigma}^{-1}(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2), \tag{3.13}$$

which is Eq. (3.5).

Threshold (Eq. 3.6). The scalar threshold d is obtained by collecting the remaining (non-$\mathbf{x}$) terms in (3.10), giving

$$d = \frac{1}{2}\left(\boldsymbol{\mu}_1^\top \mathbf{\Sigma}^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2^\top \mathbf{\Sigma}^{-1}\boldsymbol{\mu}_2\right) - \log\left(\frac{\phi_1}{\phi_2}\right), \tag{3.14}$$

which is Eq. (3.6). When $\phi_1 = \phi_2$ (equal priors), the logarithmic term vanishes and the split becomes symmetric in the projected space. When $\phi_1 \neq \phi_2$, the threshold shifts toward the less probable component, reflecting the influence of the priors on the decision boundary (Bishop, 2006b).

3.3.3.3 Hyperplane Generation for Regression

The construction of hyperplanes during a regression task is similar to classification with small variations. In a regression task, the aim of GODT is to construct a hyperplane to separate the data in regions, however, unlike classification, the aim here is to reduce the prediction error within each partition.

The are two main objectives to constructing hyperplane in GODT regression (GODT-R):

- **Minimize Prediction Error:** The hyperplane will aim to separate the data such that this reduces the error (e.g., Mean Squared Error (MSE)) in each child node.
- **Creating Prediction Regions:** The aim of the regions created by GODT-R is to make sure that the regions created by the splits should have a low MSE in the target, leading to more accurate and consistent predictions.

In GODT-R, the split hyperplane at each non-terminal node is constructed identically to the classification case using the calibrated two-component GMM (Sections 3.3.1-3.3.3). The difference lies in the stopping criterion and leaf prediction, nodes stop splitting based on an MSE threshold and leaves predict the mean target value.

3.3.4 Hyperparameters for GODT

In our GODT technique for classification and regression, we have three main hyperparameters, namely, the δ parameter to control the purity of the node/sub-region in classification and the prediction error within the region in regression. The β parameter that sets the regularization for the GMM, and finally the M which looks at the number of components in GMM.

3.3.4.1 Hyperparameters in GODT-Classification

Firstly, as we aim for a binary split, we work with the approximation of having only two components for the GMM and we are fully aware that there may be more, but these will be explored in future work. So, $M = 2$.

Secondly, δ is the purity hyperparameters that takes the value between $(0, 1]$ and requires tuning. As the value for $\delta \rightarrow 1$, this would mean that each non-terminal node is fully pure and data in the node belongs to only a single class. This is a parameter that we tune on the validation dataset. The aim of the purity threshold δ is to prevent unnecessary splitting when the node already has a high purity.

Finally, β is the regularization hyperparameter; very crucial for a GMM to ensure that the GMM is stable and well-defined. The β value is added to the diagonal of the covariance matrices to prevent them from becoming singular or ill-conditioned. This is a parameter that we tune and we ensure to try small positive values.

3.3.4.2 Hyperparameters in GODT-Regression

Similar to GODT-C, as we aim for a binary split in GODT-R, we work with the approximation of having only two components for the GMM. So, $M = 2$.

Secondly, the δ hyperparameter in regression aims to control the prediction error, where it uses MSE to look at how consistent the target values within each node or sub-region are. This is unlike classification where we aim to have pure nodes based on the homogeneity of classes. We set a threshold for the MSE when splitting the nodes,

as the tree can stop splitting further when the MSE in the node reaches the threshold. We tune this hyperparameter on the validation data.

Finally, β is the regularization hyperparameter which ensures that the GMM is stable and well-defined. This is a parameter that we tune on the validation dataset and we make sure to try small positive values.

3.3.5 Non-Splitting condition

When building a decision tree, it is necessary to understand the limit until which a node can be split further. The condition used with which we decide whether to split a non-terminal node any further or whether to convert it into a leaf node is known as the non-splitting condition. Below, we look at two important non-splitting conditions, namely, the purity measure for classification and the error measure (based on MSE) for regression.

The reason we choose a purity-based stopping rule is for its simplicity, interpretability, and consistency with the decision-tree framework. It provides a direct and intuitive control over when a node should be considered sufficiently homogeneous. Unlike criteria such as information gain thresholds, statistical significance tests, or minimum node-size constraints, the purity threshold δ explicitly quantifies the dominance of a single class within a node. This makes the stopping condition independent of the particular impurity measure used elsewhere (e.g., Gini or entropy), and allows for a straightforward hyperparameter interpretation.

3.3.5.1 Non-splitting condition for GODT-Classification

In classification, we calculate the purity of a node, and then use the threshold δ to decide whether to split a node further or to convert it into a leaf node. Let us say have a instances at a non-terminal node, where $a \subset n$

$$\text{purity}(\mathbf{X}_a) = \max(\frac{n_i}{a}), \text{i = 1, ..., c.} \tag{3.15}$$

Here, a denotes the total number of instances in $\mathbf{X}_a$ and $\mathbf{X}_a$ is a subset of rows from $\mathbf{X}$. We let $\frac{n_i}{a}$ determines the proportion of nodes of class i for $i = 1, ..., c$ for c unique classes in the sample matrix $\mathbf{X}_a$ at each non-terminal node. At each non-terminal node, if $\max(\frac{n_i}{a}) \geq \delta$, the non-terminal node is converted to a leaf node. Otherwise, we continue with the splitting.

3.3.5.2 Non-splitting condition for GODT-Regression

In regression, we decide whether to split a node any further or whether to convert the node to a leaf node depending on the Mean Squared Error (MSE) in each node rather than the purity. We assess the MSE against the threshold value δ to make this decision. If we say have a instances at a non-terminal node, where $a \subset n$

$$\text{MSE}(\mathbf{X}_a) = \frac{1}{a}\sum_{i=1}^{a}(y_i - \bar{y}_{X_a})^2. \tag{3.16}$$

Here, a denotes the total number of instances in $\mathbf{X}_a$, $\mathbf{X}_a$ is a subset of rows from $\mathbf{X}$, y_i are the observed target values, and $\bar{y}_{X_a}$ is the mean of the target values in X_a. At each non-terminal node, if $\text{MSE}(\mathbf{X}_a) < \delta$, the non-terminal node is converted to a leaf node. Otherwise, we continue with the splitting.

3.3.6 Initialization

Another major consideration when constructing clusters is cluster initialization. This is because, the performance of a clustering technique which converges to a local maximum of the likelihood function depends highly on initial cluster centroids. If a clustering technique has a poor initialization, this may lead to empty clusters, slower convergence, and a high chance of getting stuck in bad local minima (Celebi et al., 2013). Techniques like K-means converge to a local minimum, but GMM converges to a local maximum due to the EM algorithm. Moreover, Panić et al. (2020) demonstrated that careful initialization when using the EM algorithm is necessary, even when applied in highly favorable settings, and performing random initialization would converge to bad critical points (Jin et al., 2016).

The standard CART technique provides consistent results at every run, provided that we are using the same sample of data. However, in the GMM, because it is based on the EM algorithm, at every run, the result that we achieve varies. So, to solve this issue, we initialize GMM cluster centroids using a less time-complex technique like K-means, where the K-means cluster centroids are initialized using the technique mentioned in Wang et al. (2018). Because we use the EM algorithm for determining the hyperplane parameter, we have some random nature in our algorithm leading to some variability, and to reduce the variability we add double initialization.

The three separate parameters for the GMM, i.e., the mean, covariance, and the prior mixture components require initialization as part of the implementation. Firstly, the means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are initialized using K-means with a further step of K-means centroids initialized using Algorithm 6. We should also note that as part of Algorithm 6, the data is centered at every non-terminal node. Secondly, for the diagonal

covariance matrix $\mathbf{\Sigma}$, the diagonal elements are initialized as the variance of each respective feature. Finally, the prior mixture components ϕ are set to uniform for initialization, i.e., if we have $M = 2$ for example, each component receives an initial equal weight of $\frac{1}{2}$.

We observe in a small experiment that the computational times for training a GODT do not vary a lot with initializing using Algorithm 6 and random initialization. However, for high-dimensional instances, it shows that initialization using Algorithm 6 improves performance, where otherwise no significant difference can be observed. For this reason, we prefer to keep the initialization using Algorithm 6 as part of the standard version of our algorithm. Please refer to the Tables A.2 and A.3 in Appendix A.1 for the results.

Here, our contribution is to build an oblique decision tree technique using hyperplane splitting without computationally intensive hyperplane optimization.

3.3.7 Detailed description of Algorithm 5 and Algorithm 6

Algorithm 5 describes the practical construction of the Gaussian Oblique Decision Tree (GODT) in a top-down manner. For clarity and reproducibility, we now describe the execution flow and implementation details, while referring to earlier sections for the mathematical formulation of the Gaussian mixture model calibration and the hyperplane.

Node management and termination: The algorithm begins by assigning all training samples to the root node, which is inserted into a processing set Ψ. At any iteration, Ψ contains all non-terminal nodes that are candidates for further splitting. The algorithm repeatedly selects a node $N \in \Psi$, extracts the corresponding subset of data $(\mathbf{X}, \mathbf{y})$, and checks whether N satisfies a non-splitting condition. In the classification setting, this condition is based on a purity threshold; in the regression setting, it is based on a mean squared error (MSE) threshold. Nodes containing fewer than a certain number of samples, they are also converted directly into leaf nodes. When a node is converted into a leaf, it is assigned a prediction equal to the majority class (classification) or the mean target value (regression).

Gaussian mixture model fitting: If node N does not satisfy the non-splitting condition, it is split by fitting a two-component Gaussian mixture model (GMM) to its feature matrix $\mathbf{X}$. As detailed in Section 3.3.2, the GMM is estimated by maximum-likelihood using the Expectation-Maximization (EM) algorithm with a shared diagonal covariance matrix and regularization parameter β added to the

diagonal to ensure numerical stability. The fitted GMM yields the component means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, the shared covariance matrix $\boldsymbol{\Sigma}$, and the mixture weights π_1 and π_2.

Initialization strategy: Because the EM algorithm is sensitive to initialization, the component means of the GMM are not initialized randomly. The means are obtained from a K-means clustering step whose centroids are computed using the PCA-based procedure described in Algorithm 6. Algorithm 6 constructs two initial centroids by projecting the data onto the first principal component, splitting the projections at their median, and computing the mean of each resulting subset. These centroids are then used to initialise a two-cluster K-means algorithm, whose output assignments are passed to the EM procedure as initial component memberships.

Conversion of GMM parameters to an oblique split. Once the GMM has been fitted at node N, its parameters are converted into an oblique split of the form $\mathbf{w}^\top \mathbf{x} = d$. The normal vector $\mathbf{w}$ is computed from the difference between the component means scaled by the inverse shared covariance matrix, and the cut-point using equation 3.6. This results in a linear decision boundary.

Child node creation. The samples in node N are partitioned into two child nodes . Each child node inherits the corresponding subset of the data. For each child, the non-splitting condition is immediately evaluated. If satisfied, the child is converted into a leaf; otherwise, it is added to the processing set Ψ for further expansion.

3.4 Experimental Results and Discussions

In this section, we will discuss more about the data preparation, experimental setup and the final results for both GODT classification and regression.

3.4.1 Classification Results and Discussions

In this subsection, we will look at the different machine learning datasets that we will be working with, along with the experimental setup, and the classification performance for GODT-Classification (GODT-C).

3.4.1.1 Machine Learning Datasets

We conduct our experiments on 22 benchmark datasets that have come mainly from the UCI repository (Dua and Graff, 2017), and we also use some datasets from other

Dataset	# Features	# Training data	# Testing data	# Classes
	High-Dimensional dataset			
Gisette	5000	5250	1750	2
Madelon	500	1950	650	2
Usps	256	6974	2324	2
	Large-scale dataset			
Acoustic	50	73930	24643	3
Combined	100	73930	24643	3
Poker	10	769758	256252	10
	Normal-sized dataset			
Abalone	7	3133	1044	3
Adult	14	36632	12210	2
Breast Cancer	9	513	170	2
Dermatology	34	275	91	6
Drug Consumption	31	1414	471	7
Ecoli	7	252	84	3
Glass	9	161	53	7
Heart Failure	11	224	75	3
Ionosphere	34	263	88	2
Iris	4	113	37	3
Letter	16	15000	5000	16
Movementlibras	90	270	90	3
Mushroom	22	6093	2031	2
Satimages	36	4826	1609	2
Segmentation	19	157	53	7
Wine	12	134	44	3

TABLE 3.1: Information of the datasets used during the experimental stage

websites (Chang and Lin). Most of these datasets have been evaluated in detail in previous studies on oblique decision trees and other machine learning techniques. Among the 22 datasets, 14 are multi-class datasets while the remaining 8 are binary datasets. The details of the dataset are provided in Table 3.1. We do not perform any sort of dimensionality reduction, as we want to observe the performance of our model on small, large-scale, and high-dimensional datasets. However, a very small number of datasets have been scaled between $[-1, 1]$ at the source (Chang and Lin), so we use them in the same form without performing any further pre-processing. We split the dataset into a 75:25 train-test split, where a random subset of 20% of the training data is used for validation. We use test accuracy as our performance measure. The test accuracy (out of sample accuracy) looks at the total number of correct predictions on the test data out of the total number of observations in the test data as a percentage.

3.4.1.2 Experimental Setup

There are six different models we compare here, five of which (GODT-C, SVM, LDA, CART, and QDA) have been implemented in MATLAB 2019b on a computer which is Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz 3.40 GHz with 8.00 GB RAM. The OC1 model has been implemented in Python on the same computer specification. We

Dataset	GODT-C	SVM	LDA	CART	QDA	OC1
	High-Dimensional dataset					
Gisette	91.80	**96.80**	88.80	93.50	50.00	**
Madelon	**76.00**	57.33	58.67	74.83	51.11	65.44
Usps	**91.86**	91.48	82.20	87.15	42.37	89.94
	Large-scale dataset					
Acoustic	**70.55**	67.00	57.81	68.20	66.89	*
Combined	76.31	**80.61**	66.81	78.70	70.34	*
Poker	56.68	50.19	50.12	52.64	50.12	**61.35**
	Normal-sized dataset					
Abalone	**55.15**	52.67	53.87	51.16	54.99	48.72
Adult	**76.38**	74.22	**76.38**	74.39	**76.38**	66.58
Breast Cancer	**97.81**	96.35	**97.81**	95.62	96.35	92.65
Dermatology	**95.65**	**95.65**	92.39	91.30	32.61	87.91
Drug Consumption	**78.13**	17.41	63.27	73.89	46.28	68.94
Ecoli	**85.71**	79.76	79.76	78.57	66.67	78.31
Glass	**67.44**	58.14	51.16	55.81	58.14	61.90
Heart Failure	**85.00**	55.00	71.67	75.00	77.78	76.27
Ionosphere	84.76	86.67	82.86	85.71	63.81	**93.27**
Iris	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
Letter	82.12	**84.14**	59.56	78.40	63.88	***
Movementlibras	**76.67**	71.11	56.67	66.67	65.56	11.24
Mushroom	**100.00**	97.21	87.40	99.92	99.43	99.55
Satimages	**89.12**	87.10	80.11	86.01	86.25	83.67
Segmentation	88.68	86.79	79.25	83.02	81.13	**88.57**
Wine	**100.00**	91.67	**100.00**	94.44	**100.00**	82.86

TABLE 3.2: Test accuracies of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code.

compare our GODT-C technique with the following techniques that aim at separating the classes using a decision boundary, either orthogonal, linear, or quadratic. These techniques are as follows:

- Support Vector Machine (SVM)
- Linear Discriminant Analysis (LDA)
- Classification and Regression Tree (CART)
- Quadratic Discriminant Analysis (QDA)
- A System for Induction of Oblique Decision Trees (OC1)

We use OC1 rather than other Oblique decision tree techniques like LMDT (Utgoff and Brodley, 1991) because, this technique is established and has the code base in C published by the author, along with an available Python wrapper for the C implementation, which would mean that running OC1 technique would give us the true model performance in comparison to when we implement a model from scratch,

which is where we can potentially sacrifice the true performance of the comparison model.

All experiments were conducted using a consistent evaluation protocol across datasets and methods. Each dataset was split into training and testing sets using the same random seed for all models to ensure fair comparisons. Model training and hyperparameter tuning were performed using the training data and validation dataset respectively, while the test set was used exclusively for final performance evaluation. Classification performance was measured using accuracy, and regression performance was measured using mean squared error (MSE). We discuss more on the hyperparameter tuning strategy in section 3.4.1.3.

For the CART technique, we utilize the `fitctree()` to learn a single CART decision tree and the function `predict()` to test the model performance. For the SVM we use the `fitcecoc()` to train the model and the function `predict()` to test the model performance. For the LDA and QDA, we use `fircdiscr()` to train the model with the `discrimType` being either linear or quadratic, along with the `predict()` to test the model performance. For SVM, CART, LDA, and QDA, we use the Statistics and Machine Learning toolbox in MATLAB. For the OC1 we use the sklearn oblique tree library, which constructs a Python interface for the oblique tree code written in C. The reason that we use the above techniques is due to their capability to construct an axis-parallel hyperplane or a linear hyperplane. So, CART constructs an axis-parallel decision boundary, where as the remaining techniques constructs a linear decision boundary, apart from QDA which constructs a quadratic decision boundary. Moreover, the above techniques can be implemented on both binary and multi-class data. The SVM uses Error-Correcting Output Codes (ECOC) to extend the binary classification problem to multi-class classification task.

Throughout this thesis, we assume that all input features are numerical. When categorical variables are present, a standard approach is to transform them into numerical representations prior to training. One common technique is one-hot encoding, where a categorical variable with k categories is represented by k binary indicator features. This transformation allows GODT-based methods to be applied without modification, as they operate on numerical feature spaces.

However, one-hot encoding increases the dimensionality of the feature space and may introduce sparsity, particularly for categorical variables with many levels. This can negatively impact computational efficiency and may affect the stability of oblique split learning. In contrast, classical decision trees such as CART can handle categorical variables directly by considering subset-based splits, which avoids dimensionality expansion but may become computationally expensive when the number of categories is large.

As a result, while GODT and similar methods in the other chapters can accommodate categorical features via standard pre-processing techniques, classical decision trees retain an advantage in handling categorical variables natively. Extending GODT to directly support categorical splits, or developing hybrid strategies that combine native categorical handling with oblique decision boundaries, constitutes an interesting direction for future research.

Dataset	GODT-C	SVM	LDA	CART	QDA	OC1
	High-Dimensional dataset					
Gisette	185.7200	157.6600	90.8300	4.7725	44.0500	**
Madelon	2.4563	43.6181	0.8737	0.2688	0.3004	89.4430
Usps	5.7451	3.1617	1.3886	1.7287	0.8722	184.2293
	Large-scale dataset					
Acoustic	19.4608	299.2567	1.5999	4.3866	0.5041	*
Combined	90.0542	372.3168	0.8972	8.1299	0.8939	*
Poker	8.8307	81.3286	0.7516	0.6860	0.2231	4778.1208
	Normal-sized dataset					
Abalone	0.5214	1.5474	2.3698	1.0086	0.2698	12.5145
Adult	3.0967	784.6169	0.8075	0.7297	0.1892	1351.2251
Breast Cancer	0.4846	0.5246	0.7688	0.6409	0.1251	18.0284
Dermatology	0.0508	1.2145	1.2733	0.0583	0.2404	6.8514
Drug Consumption	0.2104	94.5625	1.2491	0.0494	0.3382	32.8815
Ecoli	0.4881	1.5283	1.0574	0.6271	0.2533	11.3392
Glass	0.3594	0.6638	0.7102	0.6261	0.1993	0.3030
Heart Failure	0.5346	3.7935	1.0844	0.6115	0.1226	0.2809
Ionosphere	0.3569	0.4570	0.7165	0.6476	0.1121	1.0369
Iris	1.3062	0.5610	0.9392	0.7477	0.1648	0.5010
Letter	1.0739	4.3940	0.3791	0.0966	0.5844	***
Movementlibras	0.6789	1.1943	0.8359	0.6660	0.3137	54.7651
Mushroom	0.3424	73.3111	0.7167	0.2271	0.1804	0.4564
Satimages	0.6664	0.6918	0.1353	0.1048	0.1733	208.4484
Segmentation	0.7485	0.8060	0.7267	0.6578	0.2011	5.8773
Wine	0.1110	0.5936	0.6947	0.6660	0.1391	1.6618

TABLE 3.3: Training times of the models in seconds. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code.

3.4.1.3 Hyperparameter Tuning Strategy

Hyperparameter tuning was carried out systematically for all methods using grid search over predefined parameter ranges. For each model, multiple combinations of hyperparameters were evaluated on the validation set, and the configuration achieving the best validation performance was selected. The same tuning strategy, data splits, and evaluation metrics were used across all methods to ensure a fair and consistent comparison. For the Gaussian Oblique Decision Tree (GODT), tree-level hyperparameters were tuned using grid search. These included the node purity

threshold δ, the GMM regularization parameter β, and the minimum number of samples per leaf. The number of Gaussian mixture components was fixed to $M = 2$ in order to implement a binary tree structure.

The δ parameter takes the value $\delta \in (0, 1]$. The regularization parameter $\beta > 0$, must be a non-negative real number. This is, in practice, a very small positive value. We tune this value on the validation dataset. When tuning the GODT-C model, the purity parameter is incremented by 0.05 to find the value that gives the best accuracy on the validation data at each non-terminal node. We measure the performance of different techniques using the test dataset. We select the parameter combination that provides us with the highest accuracy on the validation dataset. This parameter combination is again used when training the final model. In most cases, the δ parameter is close to one. Alongside these hyperparameters, we have some further hyperparameters like maximum depth and minimum leaf size in GODT-C technique, to keep it consistent with standard decision trees.

To ensure a fair and reproducible comparison, all baseline models were tuned using a grid search over predefined hyperparameter ranges. The same training and validation splits were used across all models for each dataset, ensuring that observed performance differences reflect modelling choices rather than tuning options. Unless explicitly stated, hyperparameters not listed were fixed to their default values in the corresponding MATLAB and Python implementations.

The selected search ranges were chosen to balance options and computational feasibility, and were informed by standard practice in the literature as well as MATLAB's recommended defaults. For support vector machine (SVM) baselines, multiclass classification was implemented using MATLAB's error-correcting output codes (ECOC) framework with linear SVM base learners. The box constraint parameter C of the underlying SVM classifiers was tuned via grid search, while all other parameters were fixed to their default values. For support vector regression, an RBF kernel was used with the box constraint tuned.

For linear and quadratic discriminant analysis (LDA and QDA), models were trained using MATLAB's default implementation. No hyperparameter tuning was performed for these methods, and covariance estimation followed the default formulation. These models were included as classical reference baselines.

The parameter choices in OC1 are designed to balance search quality with computational efficiency when identifying oblique splits. Deterministic hill-climbing is used to rapidly locate locally optimal hyperplanes, while multiple restarts (R) allow exploration of different regions of the search space and help mitigate sensitivity to initialization. Random jumps (J) are introduced at local minima to escape stagnation without incurring the high computational cost of global stochastic optimization. Sequential coefficient perturbation is employed due to its simplicity and comparable

empirical performance relative to alternative perturbation orders. The Twoing Rule is used as the impurity measure, as it has been shown to perform robustly across a wide range of classification problems in oblique decision tree induction.

For the techniques like SVM and OC1 where we observe instability in the learning process, i.e., the test accuracies are not the same at every repetition, we repeat the process 10 times and record the average of the performance metrics executed 10 times. We also record the training times of the different algorithms. For these, we run the algorithm 10 times and record the average time taken to train on the dataset. A summary of all tuned hyperparameters and their corresponding search ranges is provided in Tables 3.4 and 3.5.

3.4.1.4 Hyperparameter Ranges

Hyperparameter	Range
Purity threshold δ	[0.5, 1.0] (step 0.05)
GMM regularization β	$\{10^{-6}, 10^{-5}, 10^{-4}, 10^{-3}\}$
Minimum leaf size	$\{1, 5, 10, 20\}$

TABLE 3.4: Hyperparameter ranges for GODT.

Model	Tuned hyperparameters and search ranges
CART (Decision Tree)	• Minimum leaf size: $\{1, 5, 10, 20\}$ • Split criterion: {Gini, MSE}
OC1 (Oblique Decision Tree)	• Number of restarts (R): 20 (effective range: 20–50) • Random jumps per local minimum (J): 5 (up to 20 tested) • Coefficient perturbation order: Sequential • Impurity measure: Twoing Rule
SVM (multiclass via ECOC)	• Kernel type: {linear, RBF} • Box constraint C (`BoxConstraint`): $\{10^{-3}, 10^{-2}, \ldots, 10^{3}\}$ • Kernel scale (`KernelScale`, RBF only): auto (fixed)
LDA	• MATLAB default settings
QDA	• MATLAB default settings

TABLE 3.5: Hyperparameter search space for baseline models (MATLAB implementations).

3.4.1.5 Classification performance

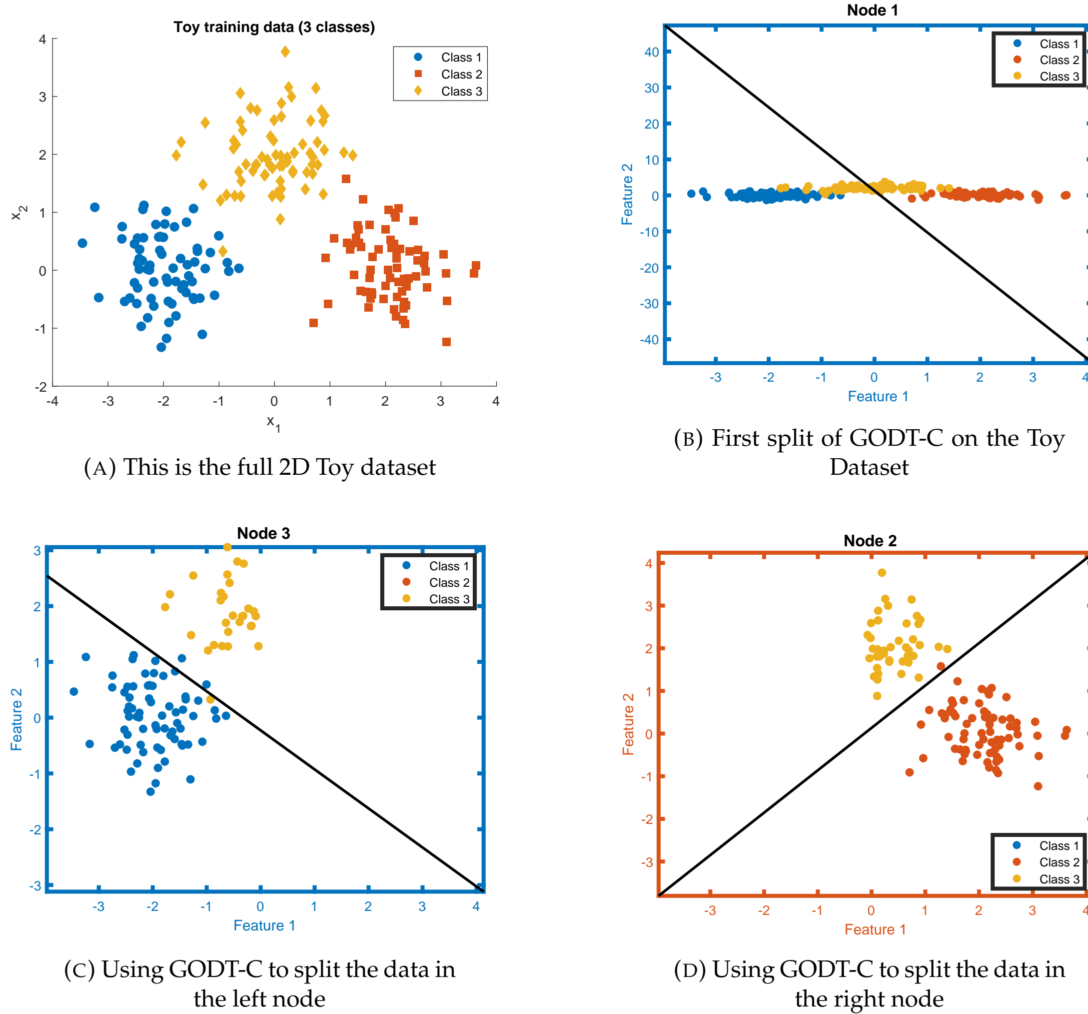
Table 3.2 illustrates the test accuracies (i.e., classification performance) of GODT-C, SVM, LDA, CART, QDA, and OC1 on 22 open-source different datasets. We observe that our GODT-C technique either performs better or performs similar to the top

performing model on 16 out of 22 datasets. So, for example, on the Gisette dataset, the SVM performs better with an accuracy of 96.80. However, we observe that the our technique returns a third best performance of 91.80 behind CART, which is still high. Another such example is the Letter dataset, where the SVM again exhibits a better performance of 84.14. However, on the same dataset, we observe our technique performing at a close second place with a decent accuracy of around 82.12. Like the No Free Lunch theorem (Adam et al., 2019) suggests, there is no single technique that is best for all the datasets. However, we observe that our technique has a consistently decent performance across all the datasets. Tables A.5, A.6, and A.7 in Appendix A.3 share the macro-averaged precision, macro-averaged recall, and macro-averaged F1-scores. These convey a similar message that GODT-C performs decently good across different datasets.

Referring to the computational times illustrated in Table 3.3, we observe that our technique's computational time is very similar to CART, except for a couple of cases in the large-scale and high-dimensional setting. As research in the oblique tree area suggests, constructing oblique trees is computationally intensive. We observe that on most datasets, our technique has computational times similar to or quicker than SVMs. We do observe that on high-dimensional datasets, our technique has a computational time greater than CART, yet less than the SVM.

We implement the GODT-C technique on binary, multi-class, high-dimensional, low-dimensional, normal-sized datasets. We observe that on all of the different types of datasets, the GODT-C performs consistently better or very close to the best model in computational accuracy. Looking at the computational times, GODT-C consistently either has low computational times or computational times similar to the axis-parallel counterpart on normal-sized datasets. On the high-dimensional and large-scale datasets, the computational times are similar to or quicker than the SVM and quicker than OC1. This shows that we have a versatile technique with the capability to have a decent accuracy and computational time across different types of datasets. Furthermore, Figure 3.5 demonstrates how the GODT-C does splitting on a toy 2-dimensional dataset that we create that consists of 3 classes. The GODT-C requires three splits to achieve leaf nodes that majorly consist of data corresponding to a single class.

We also compare the tree depth of GODT-C with CART in Appendix A.9, Table A.17. We observe that, on most datasets, the depth of GODT is lower than that of CART (i.e., GODT builds shorter trees) while achieving comparable performance. This indicates that oblique split–based decision trees require fewer splits to reach the same performance compared with their axis-parallel counterparts.



(A) This is the full 2D Toy dataset

(B) First split of GODT-C on the Toy Dataset

(C) Using GODT-C to split the data in the left node

(D) Using GODT-C to split the data in the right node

FIGURE 3.5: We create a simple toy 2D dataset of 3 classes, and we demonstrate how the GODT-C technique would perform splits at each node

Method	Rank	Raw WSR p	Bonferroni p
SVM	2	6.425×10^{-3}	3.213×10^{-2}
CART	3	4.184×10^{-4}	2.092×10^{-3}
OC1	4	3.599×10^{-3}	1.800×10^{-2}
QDA	5	1.318×10^{-4}	6.590×10^{-4}
LDA	6	1.964×10^{-4}	9.820×10^{-4}

TABLE 3.6: Wilcoxon signed-rank test p-values (raw and Bonferroni-corrected, $m = 5$) excluding errors

3.4.1.6 Statistical Comparison

We use a Wilcoxon signed-rank test to determine the significance amongst each pair of the classifiers that we use. The Wilcoxon signed-rank test is used to compare two related samples, matched samples, or to conduct a paired difference test of repeated measurements on a single sample to assess whether their population mean ranks differ (Woolson, 2007). Techniques like OC1 fail to provide results on some benchmark datasets. In such cases, we exclude these datasets rather than imputing them with a

Method	Average Friedman Rank	Mean Accuracy
GODT-C	1.56	82.99
SVM	3.64	76.24
CART	3.64	79.32
OC1	3.81	75.82
QDA	4.17	68.19
LDA	4.19	74.39

TABLE 3.7: Average Friedman Rank and Mean Accuracy

value which may eventually end up inflating the performance of that specific classifier. We have followed a similar procedure as Katuwal et al. (2020). The results are reported in Table 3.6. For the Wilcoxon signed-rank test, we have the following:

- **Null Hypothesis (H_0)**: There is no significant difference in the performance (e.g., accuracy) between GODT-C and the other classifiers.
- **Alternative Hypothesis (H_1)**: GODT-C performs significantly different (two-sided) than the other classifiers

Since all classifiers yield raw Wilcoxon p-values below 0.05, and after Bonferroni correction, all adjusted p-values remained below 0.05. Therefore, we reject the null hypothesis (H_0) in favor of H_1, concluding that GODT-C's performance differs significantly from each of the other classifiers.

The average Friedman rank provides a relative comparison of classifier performance across all datasets, where a lower rank indicates better performance. This is reported in Table 3.7. Mean accuracy refers to the average classification accuracy achieved by each model across the evaluated datasets.

GODT-C obtained the lowest average Friedman rank and the highest mean accuracy, suggesting that it consistently performed better than the other classifiers. These results indicate that GODT-C may serve as a strong baseline for future comparisons.

3.4.2 Regression Results

In this subsection, we will discuss the performance of GODT technique for regression (GORT-R) in comparison to other established techniques. In the previous sections, we consider the adjustments that we make to the existing GODT-R architecture to make it suitable for the regression task, along with the experimental setup.

3.4.2.1 Machine Learning Datasets

We conduct our experiments on 20 datasets that are a combination of open-source datasets from the UCI repository Dua and Graff (2017) and datasets from our

	# Features	# Training data	# Testing data
	Large-scale dataset		
mcev10	5	45000	15000
mcev100	5	45000	15000
mcev1000	5	45000	15000
bejoingpm	11	31318	10439
	Normal-sized dataset		
electrical grid	14	7500	1200
combined cycle	5	7176	2392
air quality	10	7018	2339
bias correction	25	5691	1897
gas turbine	12	5558	1853
parkinsons	22	4700	1175
airfoil self noise	5	1127	376
concrete	9	773	257
can combustion	26	596	198
energy efficiency	10	576	192
Istanbul stock exchange	10	402	134
forest fires	12	388	129
3d aeroplane	33	385	128
auto mpg	5	294	98
computer hardware	10	157	52
ale	4	80	27

TABLE 3.8: Information of the datasets used during the regression experimental stage

industrial partners. The non-public industrial partners datasets were provided by our industrial partners in the finance and space sector. These datasets relate to market-consistent embedded values and computational fluid dynamics datasets, and are used here with permission. The industrial partners are happy for us to publish the datasets as part of the thesis. The details of all the datasets are provided further in the Table 3.8. In order to test the performance of our model, we work with different sizes of datasets, both large-scale and normal-sized. In order to train and test the performance of our technique, we split the dataset into a 75:25 train-test split, where a random subset of 20% of the training data is used for validation. To measure how well our technique is performing in comparison to other machine learning techniques that we used, we use performance metrics like the Mean Absolute Error (MAE) and Mean Squared Error (MSE). Furthermore, we do not perform any dimensionality reduction on the data that we use. However, we do perform some pre-processing to remove irrelevant features like ID's, timestamps, and other similar features that are not helpful for the prediction of the targets. This would ensure that the model is being trained on data that is most relevant.

3.4.2.2 Experimental Setup

We implement the GODT-R technique in MATLAB 2019b on a computer which is Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz 3.40 GHz with 8.00 GB RAM. We compare our GODT-R method against Classification and Regression Trees (CART), and Support

Vector Machine. Both of these techniques are available as part of MATLAB's Statistics and Machine Learning Toolbox, and have been implemented in the same computer configuration. The reason we chose the above comparison methods is because we wanted well-established techniques for regression, and techniques that are either tree-based or construct a hyperplane.

The CART technique for Regression in MATLAB constructs binary trees by minimizing the prediction error. It uses MSE to decide the feature to split on and the value to split on. The aim of SVM for regression technique in MATLAB is to find a function that approximates the relationship between the attributes and the target by minimizing the prediction error.

For the CART technique, we utilize the `fitrtree()` to learn a single CART regression tree and the function `predict()` to test the model performance. For the SVM, we use the `fitrsvm()` to train the model with Radial Basis Function kernel (RBF-kernel) and the function `predict()` to test the model performance. The reason that we use the above techniques is due to their capability to construct an axis-parallel hyperplane or capture complex relationships. So, CART constructs an axis-parallel decision boundary, whereas the SVM with RBF-kernel is powerful at capturing complex, non-linear relationships.

The training process for the GODT-R, CART, and SVM techniques requires validation datasets. Specifically, for GODT-R, we use the validation dataset to tune the node error parameter δ and the regularization parameter β. The regularization parameter β, must be a non-negative real number. This is, in practice, a very small positive value. We tune this value on the validation dataset. We select the parameter combination that provides us with the lowest MAE on the validation dataset. This parameter combination is used again when training the final model. The hyperparameter tuning strategy used here is similar to section 3.4.1.3. The only difference is the value of parameter δ value ranges for regression, which we set to a small value, as the aim is to have a low MAE at each non-terminal node.

3.4.2.3 Regression Performance

The Table 3.9 displays the MAE of GODT-R, CART, and SVM on 20 different datasets that cover different domain areas. We measure the regression performance based on MAE. Comparing GODT-R against CART, GODT-R performs better than CART 9 out of 20 times based on the MAE. On the contrary, CART performs better than GODT-R 7 out of 20 times. On the 7 out of 20 times CART performs better than GODT-R, the MAE of the GODT-R is very close to CART 4 out of the 7 datasets. On the contrary, the scenarios when GODT-R performs better, we observe that CART has a similar performance on 2 datasets. We come across one instance on the Istanbul stock

	GODT-R	CART	SVM
	Large-scale dataset		
mcev10	2.8775	2.9675	**2.7636**
mcev100	1.0809	0.9895	**0.9184**
mcev1000	0.6768	0.5315	**0.3465**
bejoingpm	**52.3552**	52.9147	61.7168
	Normal-sized dataset		
electrical grid	0.0266	0.0249	**0.0240**
combined cycle	3.3705	**3.1326**	9.8323
air quality	0.1655	**0.1143**	6.8039
bias correction	**0.9863**	1.1327	2.5138
gas turbine	**0.8551**	0.9775	11.7125
parkinsons	0.0379	**0.0320**	0.0517
airfoil self noise	4.5527	**3.9664**	5.4898
concrete	9.8687	**7.6510**	12.1005
can combustion	**0.0708**	0.0811	0.0936
energy efficiency	1.4787	**1.1785**	1.8973
istanbul stock exchange	**0.0052**	**0.0052**	0.0061
forest fires	16.8986	18.4209	**9.7672**
3d aeroplane	**0.9471**	1.1973	1.0948
auto mpg	**2.7072**	3.0593	4.5314
computer hardware	**12.7967**	33.9584	51.9094
ale	**0.1795**	0.2034	0.3149

TABLE 3.9: This table presents the (Mean Absolute Error) MAE for each technique on the test dataset

exchange dataset, where the performance of GODT-R and CART is exactly the same. So, in these 20 datasets, we do not see a scenario where our technique performs poorly. If we compare the GODT-R technique against SVM, we observe that out of the 20 different datasets, GODT-R performs better than an SVM in 15 out of 20 datasets. On the remaining 5 datasets, SVM performs better than GODT-R. Our technique has a consistent decent performance across all of the datasets. A similar result was reflected by the MSE in Appendix A.2 Table A.4.

One thing that is consistent across literature in the area of oblique decision trees is that the training of oblique decision trees are computationally intensive. Moreover, most literature in this oblique decision tree area is focused on classification. As a result, we have limited knowledge of the performance of oblique decision trees on regression tasks. However, based on our knowledge in the performance of oblique decision trees in classification, we know that training an oblique decision tree is a computationally intensive task. Looking at the computational times across the different datasets in Table 3.10, we observe that the computational times of GODT-R is slightly slower compared to that of CART, nevertheless quicker than an SVM in most cases. When we look at the average computational times of GODT-R, CART, and SVM, these are given as 0.9458 seconds, 0.4939 seconds and 12.7101 seconds respectively. So, GODT-R is second best behind CART in computational times.

We implement our GODT-R technique on a diverse variety of datasets. We have

	GODT-R	CART	SVM
	Large-scale dataset		
mcev10	1.322474	0.205576	51.45576
mcev100	0.821766	0.079358	51.96682
mcev1000	1.630908	1.114834	39.50536
bejoingpm	0.08671	4.3329	85.90265
	Normal-sized dataset		
electrical grid	1.977073	0.579835	2.605253
combined cycle	2.916071	0.055392	3.877759
air quality	2.309924	1.048259	2.019167
bias correction	1.2444	0.06893	1.4361
gas turbine	1.140194	0.18974	1.694454
parkinsons	1.788394	0.033035	1.762865
airfoil self noise	0.233709	0.018554	0.264911
concrete	0.196753	0.205458	0.058164
can combustion	0.427625	0.167912	0.55365
energy efficiency	0.433456	0.030224	1.097657
istanbul stock exchange	0.726321	0.550912	0.079763
forest fires	0.098277	0.042507	4.616932
3d aeroplane	0.333668	0.866374	0.284676
auto mpg	0.149007	0.112017	4.582627
computer hardware	1.014906	0.135679	0.319657
ale	0.065921	0.04196	0.1193

TABLE 3.10: This table presents the training computational times of each dataset

large-scale datasets and normal-sized datasets. At the same time, the datasets come from a diverse domain. From the point of view of accuracy and computational times, we observe that there is generalization to the GODT-R technique. When we look across the 20 datasets, GODT-R achieved a mean MAE of 5.5969 (standard deviation of 11.65), whereas CART's mean MAE was 6.6269 (standard deviation of 13.27) and SVM's was 9.1944 (standard deviation of 16.44). Considering these together, these figures show that not only does GODT-R make smaller errors on average, but its performance is also more consistent across a variety of datasets, making it a robust and reliable choice for regression tasks.

In the next chapter, we will explore creating an ensemble of the GODT-C and GODT-R techniques.

Chapter 4

Gaussian Mixture Model Oblique Random Forest for Classification and Regression

4.1 Introduction

In Chapter 3, we looked at the novel Gaussian Oblique Decision Tree (GODT) technique for classification and regression, which constructs an oblique decision tree using Gaussian Mixture Models (GMM). The GODT technique constructs oblique (linear) hyperplanes at each non-terminal node to perform a binary split of a node. We observed that GODT performed decently well across multiple datasets. In this chapter, we will explore the option of creating an ensemble with the base learner being our GODT technique for both classification and regression. We will compare the performance of the Gaussian Oblique Random Forest (GORF) model against popular techniques like the Breiman-Random Forest (Breiman-RF) (Breiman, 2001), also known as the standard random forest, where the base learner is a Classification and Regression Tree (CART) (Breiman et al., 1984c).

Random Forest is a widely popular Machine Learning technique that is used in a large variety of classification and regression tasks, such as medical diagnosis (e.g., diabetes, heart disease or cancer (VijiyaKumar et al., 2019)), text classification to effectively categorize documents and emails (e.g., spam detection (Nigam et al., 2000)), visual tracking (Gall and Lempitsky, 2013), stock market forecasting (Patel et al., 2015), drug discovery such as identifying potential drug candidates by analyzing chemical properties (Svetnik et al., 2003), modelling climate data to predict temperature change (Prasad et al., 2006), and many more such examples.

Looking at the architecture, a random forest is made up of multiple decision trees. It is an ensemble technique which has a higher robustness, capability to handle multi-class labels, ability to construct parallel decision trees, and the capacity to provide feature importance. The random forest technique is still a widely used model, even though it has nearly been more than two decades since it was first proposed, simply due to its widely proven state-of-the-art performance, and its large applications. The standard random forest technique is an ensemble framework for the decision trees. Recent work in this area involving 179 models for 121 datasets (Fernández-Delgado et al., 2014), shows that the standard random forest is the top performer, implying that a decision tree is a perfect example to create an ensemble.

Breiman (1996a) introduced the idea of perturb and combine strategy, which is a foundation of the later proposed random forest technique (Breiman, 2001). The main idea behind ensemble learning is to build a single classifier by combining the multiple base learners with an aim to reduce the variance of the single classifier. From this point onwards, to distinguish Breiman's Random Forest (Breiman, 2001) from oblique decision tree-based random forest techniques, we refer to Breiman's Random Forest (Breiman, 2001) as the Standard Random Forest. The Standard Random Forest technique is an adaptation of the bagging technique. The bagging technique constructs multiple decision trees from random sub-sample of the training data, and eventually the predictions for individual decision trees are averaged.

The standard random forest technique takes this a step further by taking a random sub-sample of the feature space with an aim to further reduce the variance of the model. The random sub-sampling of the feature space forces each tree to only consider a small random subset of features at each split, leading to the trees becoming less correlated with each other. So, the individual overfitting tendencies are averaged out, thus reducing the overall model variance. Generally, ensemble techniques are known to reduce the variance when compared to using a single model (Breiman, 1996b; Geurts et al., 2006; Zhang and Zhang, 2008). In supervised learning, the overall prediction error can be understood through the bias–variance decomposition. Based on the bias-variance decomposition theory, the bias error is introduced based on how far the model's average predictions are from the target. The variance looks at how the predictions change based on the change in the training data (Breiman, 1996b; Kohavi et al., 1996). As a fully-grown decision tree inherently has low bias and high variance, creating an ensemble with this as the baseline model will be an ideal choice.

Bagging proposed by Breiman (1996a) is not the first of its kind, however was one of the first widely recognized ensemble methods. There has been earlier literature on weighted averages (Jacobs et al., 1991; Nilsson, 1969), Bayesian model averaging (Hoeting et al., 1999), ensemble heuristics like stacked generalization (Wolpert, 1992), and many such work. However, research in the area of ensemble learning suggests

that rather than building a single classifier, building an ensemble model produces better performance (Dietterich, 2000; Ren et al., 2016).

The decision tree technique within a random forest is a type of hierarchical technique that discriminates at different levels to identify and assign the class or values to a specific pattern based on classification or regression tasks. The idea behind a decision tree technique is to perform recursive partitioning of the input data into smaller subsets, by optimizing an impurity measure with an aim to improve the classification (or regression) performance. When it comes to optimizing an impurity measure, we have many options like Gini index, information gain, and entropy, with the choice of the impurity measure depending on the decision tree technique being implemented. The corresponding random forest built using the decision trees varies depending on the underlying impurity measure used for each tree. So, we can have a random forest technique that is built using the CART (Classification and Regression Tree technique) or the C4.5 technique, where the former uses Gini impurity and the latter uses gain ratio. The random forest techniques implemented from a CART or C4.5 constructs decision trees that have axis-parallel (or orthogonal) hyperplanes to the feature axis. This would correspond to a box-like or stair decision boundaries.

However, looking at all the features collectively would yield us a better understanding of the hidden relationships between them. Yıldız and Alpaydın (2012) suggests that axis-parallel decision trees using a single feature to split a space with an orthogonal boundary would result in larger trees and poor generalization. Moreover, Brodley and Utgoff (1992) insists that axis-parallel decision tree splits using a single feature would restrict the split through the instance space that is orthogonal to the axis. Breiman et al. (1984c); Brodley and Utgoff (1992); Utgoff and Brodley (1990) suggests that this bias would be unsuitable for situations where the input variables are related (or correlated).

In contrast to the axis-parallel setting, in a oblique decision tree, at each non-terminal node, a linear hyperplane (rather than an orthogonal hyperplane) separates the classes on the decision tree in an aim to improve the purity of the nodes incrementally at the subsequent nodes. In the oblique decision tree scenario, we have multiple features participating during the node splitting, unlike its axis-parallel counterpart. Rather than selecting one best feature, we select the best linear combination of features. The linear combination consists of multiplication of weights with the attributes. Even though the axis-parallel decision tree has its advantages, when we are presented with data that is correlated, we would prefer having a hyperplane splitting the data due to its better separability rather than a step-function. In Breiman (2001), the author suggested that the use of oblique splits in place of random splits produced results that were never achieved so far. However, constructing an oblique random forest i.e., an ensemble of oblique decision trees, is an area that did not gain widespread attention

when compared to random forests constructed using an ensemble of axis-parallel decision trees.

Random forest techniques are divided into two types depending on the underlying structure of the decision tree. We have either an oblique random forest (if the random forest is constructed using an ensemble of oblique decision trees) and then we have the standard random forests (if the random forest is constructed using an ensemble of axis-parallel decision trees). There are many variants of the standard random forest techniques in the literature. A couple of established variants of the standard random forest by Breiman (2001) which uses CART as the baseline tree model, extremely randomized trees by Geurts et al. (2006), which still uses CART with some additional variations, and rotational forests by Rodriguez et al. (2006), where the principal components are used to rotate the feature space. These are ensemble learning models that utilize a decision tree as a base learner with an aim to improve the robustness and accuracy of the forest. However, there are some variations when it comes to the way the three techniques differ.

The standard random forest (Breiman, 2001) constructs an ensemble of axis-parallel decision trees, where each decision tree is a CART. The standard random forest uses bootstrap aggregation along with sub-sampling of the feature space when constructing each decision tree in the forest. Each decision tree is constructed within a bootstrap sample of the training data with replacement. At every split, a random subset of the feature is selected. Then a best split is selected among those subsets using Gini impurity for classification and mean squared error for regression. In this technique, the overfitting is reduced and the model is generalized using the bootstrapping and random feature selection.

Extremely Randomized Trees, also known as Extra Trees is an ensemble of axis-parallel decision trees that is an adaptation of the standard random forest. Unlike the standard random forest, Extra Trees use the full data set and does not perform any bootstrapping of the training dataset. At each split, Extra Trees technique selects a random subset of the feature space like the standard random forest, but rather than optimizing these splits using something like the Gini impurity measure, Extra Trees selects a split threshold randomly, inducing more randomness into the tree building process. This reduces the overfitting at the cost of a small increase in bias. However, as there is not a step involved in selecting the optimal split, this constructs a forest quickly in comparison to the standard random forest.

Rotational Forests is an ensemble technique that is a variant of the standard random forest technique. The difference from the standard random forest technique is that the rotational forest technique generates individual decision trees based on a subset of the feature space that has been transformed using Principal Component Analysis (PCA). The PCA rotates the feature subset by finding the principal components that exhibit

the maximum variance in the data. The aim of rotational forest is to encourage more diversity from one decision tree to another within the ensemble, which promotes robustness in the final model. Another difference of the rotational forests from the standard random forests is that it is trained on the whole training data rather than taking a subset of the training data.

The random forest technique has been a popular technique for over two decades, but major work in this area has been focused on using an axis-parallel decision tree to construct an ensemble (Banfield et al., 2006). Decision trees can be constructed either using axis-parallel hyperplanes or oblique hyperplanes. The current popular standard random forest techniques use axis-parallel decision trees, however, axis-parallel decision trees are always not the best option to learn complex decision boundaries (Menze et al., 2011; Murthy et al., 1994; Zhang et al., 2017). These limitations of standard random forests motivated research in the area of oblique random forest. There has been limited work in the area of oblique random forests. However, it is considered that having a linear decision boundary is better than having many stair-like axis-parallel decision boundaries. The oblique random forest uses an oblique hyperplane to split the data at each non-terminal node of the individual baseline decision trees. When constructing an oblique hyperplane, it uses the linear combination of the features in the input data.

The research in the area of oblique random forests are limited. However, most of the work in the area of oblique random forests aims to construct an ensemble of oblique decision trees where each oblique decision tree is constructed using some form of a Support Vector Machine (SVM) based technique. Zhang and Suganthan (2015) proposes constructing an ensemble of oblique decision tree using Multi-Surface Proximal SVM (MPSVM). In this technique, rather than constructing a single hyperplane, the MPSVM technique constructs two parallel hyperplanes, with each hyperplane close to the data points of each class. Then, the decision boundary is set at the mid-point of the two parallel hyperplanes. The author then combined the different oblique decision trees constructed using MPSVM into an ensemble using bagging. This creates the oblique random forest using MPSVM.

Rastogi and David (2019) takes the MPSVM-based oblique random forest technique a step further to replace the MPSVM technique with a regularized MPSVM to prevent overfitting issues and deal with the small sample size problems. The author clearly states that datasets with small sample sizes would make it difficult to compute the hyperplane. Ganaie et al. (2022) takes this a step further by constructing deeper trees using double random forest. This is done to bootstrap the data at every non-terminal node along with using PCA to transform the features, which leads to increased diversity in the forest as there is a bootstrapped sample at each non-terminal node.

Katuwal et al. (2020) proposes a method that first converts a multi-class classification problem to binary problem using one-vs-all technique and then constructs a random forest of oblique decision trees that implement different linear classifiers at every non-terminal nodes, like Support Vector Machines (SVM), MPSVM, Linear Discriminant Analysis (LDA), Least Squares SVM (LSSVM), Ridge Regression, and Logistic Regression. By combining different classifiers rather than using a single one, it allows us to utilize the strengths of each classifier. Menze et al. (2011) proposes a technique that constructs a random forest of oblique decision trees where the split at each non-terminal node is constructed either using ridge regression or linear discriminate analysis. However, one of the issues with this technique is that it is only suited for binary classification problems, and for a multi-class problem, we must implement a technique such as one-vs-all to convert the multi-class problem into a binary problem.

As we discussed earlier, the time it takes to train an oblique decision tree is a major issue, hence constructing the corresponding oblique random forest would be computationally intensive. Moreover, as most techniques depend on the construction of a variant of SVM for the baseline oblique decision tree within the ensemble, using additional procedures like one-vs-all to adapt the model to a multi-class setting adds significant additional runtime. Furthermore, we want to construct a binary tree model to retain model interpretability.

In this chapter, we propose the Gaussian Mixture Model Oblique Random Forest (GORF) technique for classification and regression, which creates an ensemble of oblique decision trees constructed using the Gaussian Mixture Model (GMM) technique in the decision tree building hierarchy. We use the Gaussian Oblique Decision Tree (GODT) technique proposed in Chapter 3 as our baseline decision tree model, along with bootstrap aggregation and random sub-sample of the feature space to construct the ensemble of GODT. Our technique is suited for both binary and multi-class classification problems as well as for regression tasks. As it does not require any additional methods to convert the multi-class problem to binary, this would save us some computational time. In the next section, we introduce details of the relevant work.

Our contributions in this chapter include the following:

1. We propose a technique that uses our GMM-based oblique decision tree to construct an ensemble, i.e., an Oblique Random Forest for Classification and Regression.
2. The use of GMM constructs a hyperplane fairly quickly, which is generally considered as a task with high computational time, and in some cases even NP-hard. Hence, the construction of the corresponding Random Forest using GODT is fairly quick and has times similar to the axis-parallel counterpart

3. We show empirically that this algorithm has decently high accuracy, and performs either similar to or better than existing established techniques in almost all of the cases on the datasets that we use.

4. We demonstrate the performance of the algorithm on a mix of high-dimensional and large-scale datasets, to display the efficiency and versatility of the technique that we propose. In addition, our technique can be implemented on a mix of classification and regression tasks, whereas, most work in the area of Oblique Random Forests is suitable for only classification tasks.

4.2 Related Work

In this section, we briefly discuss the popular and widely known ensemble techniques. We look at the standard random forest, extremely randomized trees, and rotational forest. The technique that we propose is an ensemble of oblique decision tree, i.e., the ensemble of GODT that we propose in Chapter 3. GODT uses Gaussian Mixture Model technique to construct the hyperplanes necessary to build the decision trees. The random forest literature is vast and we do not aim to cover the whole literature in this section. However, we aim to cover popular Random Forest variants. On the contrary, the Oblique Random Forest literature is limited but diverse. So, we aim to cover literature that is more suitable for our work. Let us first briefly look at the way a standard random forest is constructed.

4.2.1 Standard Random Forests

As outlined in chapter 2, section 2.5, Random Forests (Breiman, 2001) construct an ensemble of decision trees, each trained on a bootstrap sample of the data and a randomly selected subset of features. The final prediction is obtained by majority voting in classification or averaging in regression. This approach effectively reduces variance and improves generalization while maintaining interpretability and robustness. However, the trees in a standard Random Forest relies on axis-parallel splits, which can limit their ability to capture complex or oblique decision boundaries. In contrast, the GORF technique discussed in this chapter extends this idea by incorporating oblique, geometry-based splits, thereby enhancing representational flexibility while retaining the variance-reduction benefits of bagging. For more information on random forests and ensemble based techniques, please refer Chapter 2, section 2.5.

4.2.2 Extremely Randomized Trees (Extra Tree)

Geurts et al. (2006) proposed a new decision tree-based ensemble technique that performs classification and regression. This is an extension to the existing random forest technique that induces more randomness into the tree building process. The idea behind this technique is to randomize the process of performing the axis-parallel splits and the selection of attributes. This technique chooses the attributes and cut-points of the decision tree at random. This method offers the advantage in computational time and generalization over standard techniques.

4.2.2.1 Technique Overview

The idea behind the Extra Tree technique is similar to other ensemble learning techniques, where the aim is to aggregate the predictions from multiple decision trees with an aim to reduce overfitting and build a model with the ability to generalize across unseen datasets. There are two main differences between Extra Tree technique and the standard random forest technique. These are as follows:

- **Random Split Points:** When it comes to selecting the cut-point for the decision tree at each non-terminal node, the standard random forest evaluates the different potential split points and selects the split point that maximizes the model performance, unlike Extra Tree, which chooses the cut-point at random from a range of possible values, adding an additional layer of diversity in the tree building process.
- **Random Feature Selection:** Similar to the standard random forest technique, ExtraTree randomly selects a subset of the original features at each non-terminal node.

However, one other discrepancy is that the ExtraTree uses all of the training data when training the individual decision trees in the model, unlike the standard random forest technique that uses bootstrapping. Algorithm 12 in Appendix A.8 provides the pseudocode for this technique.

4.2.3 Rotation Forests

The primary variation between the different random forest techniques where the underlying decision tree is an axis-parallel decision tree is in the way diversity is induced in the ensemble of the trees and the way the feature set is handled. Rodriguez et al. (2006) proposes the rotation forest technique that constructs ensemble of decision trees based on feature transformation techniques.

4.2.3.1 Technique Overview

The idea behind a rotation forest lies in introducing more diversity into the tree-building process by applying a transformation to the feature subset before training the separate classifiers in the ensemble. When constructing each classifier (in this case axis-parallel decision trees) rotation forest applies principal component analysis on the subset of the feature space for each tree in the ensemble. The main steps of constructing a rotational forest include:

- **Random Feature Selection:** The rotational forest technique randomly partitions the initial features to a disjoint subset, to make sure that they contain different feature combinations for diversity. PCA is then applied to these features to get the principal components. Then a new subset is created with the rotated features, where after the PCA process, all the principal components are retained to preserve variability.

This idea essentially rotates the feature space, which would then present the decision trees within the ensemble with a different perspective of the training data. The feature extraction for each of the decision trees within an ensemble encourages diversity within the model. The standard random forest performs bootstrapping on the training data and performs random feature selection at each non-terminal node. The rotational forest technique, however, depends solely on feature transformation, as it uses the whole dataset during the training process. Algorithm 13 in Appendix A.8 provides the pseudocode for this technique.

4.2.3.2 Gaussian Mixture Model

A Gaussian Mixture Model (GMM) is a parametric probability density function represented as a weighted sum of Gaussian component densities (Reynolds, 2009). Let Z follow a mixture of Normal distributions with mean $\boldsymbol{\mu}_j$, covariance $\boldsymbol{\Sigma}_j$, and component weights ϕ_j for the multi-dimensional model for $j = 1, ..., M$ where M corresponds to the number of components. Please refer Chapter 3, section 3.3.1 for more information on GMM.

The reason we want to focus on creating oblique decision tree ensembles is because this may potentially create oblique random forests that are useful to capture complex patterns in the data and at the same time make them more robust. Oblique decision trees are known to capture complex patterns when compared to axis-parallel decision trees, hence the corresponding ensemble would be robust at doing this (Menze et al., 2011). Oblique decision trees also have the tendency to sometimes overfit due to the flexibility that it offers to capture complex structures (Barros et al., 2011). This is

sometimes viewed as a limitation when constructing a single tree. However, when constructing an ensemble technique like a random forest, it is encouraged to construct decision trees for baseline models that overfit. The ensemble framework overcomes the high variance as a result of individual overfitted trees through aggregation. Therefore, it is ideal to use an oblique decision tree technique as part of an ensemble as something like a random forest needs the baseline trees to overfit. So, in the next section, we explore extending our work from Chapter 3 to construct an ensemble.

4.3 The proposed method

In this section, we introduce the novel Gaussian Mixture Model Oblique Random Forest technique (GORF) for classification and regression, which is an extension of the Gaussian Mixture Model Oblique Decision Tree technique for classification and regression, which we proposed in Chapter 3. The Gaussian Mixture Model Oblique Decision Tree (GODT) technique that we proposed in Chapter 3 is suitable for both classification and regression, and is an oblique decision tree technique that utilizes Gaussian mixture models at each non-terminal node to build soft clusters of two components to replicate a binary tree structure.

What we propose here is to construct an ensemble of these oblique decision trees, i.e., a random forest of GODT. The popular random forest techniques like standard random forest (Breiman, 2001), extremely randomized trees (Geurts et al., 2006), and rotation forests (Rodriguez et al., 2006) all use axis-parallel decision trees as their baseline classifiers. However, in the GORF technique, our aim is to construct an ensemble of decision trees, where the baseline classifier is an oblique decision tree, specifically the GODT from Chapter 3.

When constructing the ensemble, we mainly employ bagging along with random sub-sampling of the feature space as our chosen ensemble technique, with an aim to construct a random forest of GODT, namely Gaussian Oblique Random Forest (GORF) for classification and regression. This is the major difference between our GORF technique and existing techniques in the literature that construct an ensemble of decision trees. Our method builds the ensemble using GODT as the baseline classifier, where the GODT utilizes Gaussian mixture model at each non-terminal node of the base classifier to build soft clusters of two components to replicate the binary tree structure. We purely focus on creating sub-spaces of the initial data using Gaussian mixture model. In the limited literature of oblique random forests, most of the oblique random forest techniques try to build a hyperplane and then optimize the hyperplane to best separate the classes. However, as we saw in Chapter 3 and the literature review of Chapter 4, optimizing the hyperplane in an oblique decision tree generally causes high computational times. This prompted us to construct ensembles of the GODT

technique. Our motivation is to construct GORF with decently high accuracy and quicker computational times. Moreover, the techniques in the literature purely focus on classification tasks. We build a technique suitable for both classification and regression. The details of this technique are stated in Algorithm 8.

In this section, we explore the different components involved in constructing an ensemble of the GODT model. We look at six different elements that compose the GORF technique.

4.3.1 Bootstrap Aggregation (Bagging)

Bootstrap Aggregation, or Bagging algorithm by Breiman (1996a) is one of the earliest introductions to ensemble learning. Please refer Chapter 2 or section 4.2.1 for a detailed reference of bootstrap aggregation.

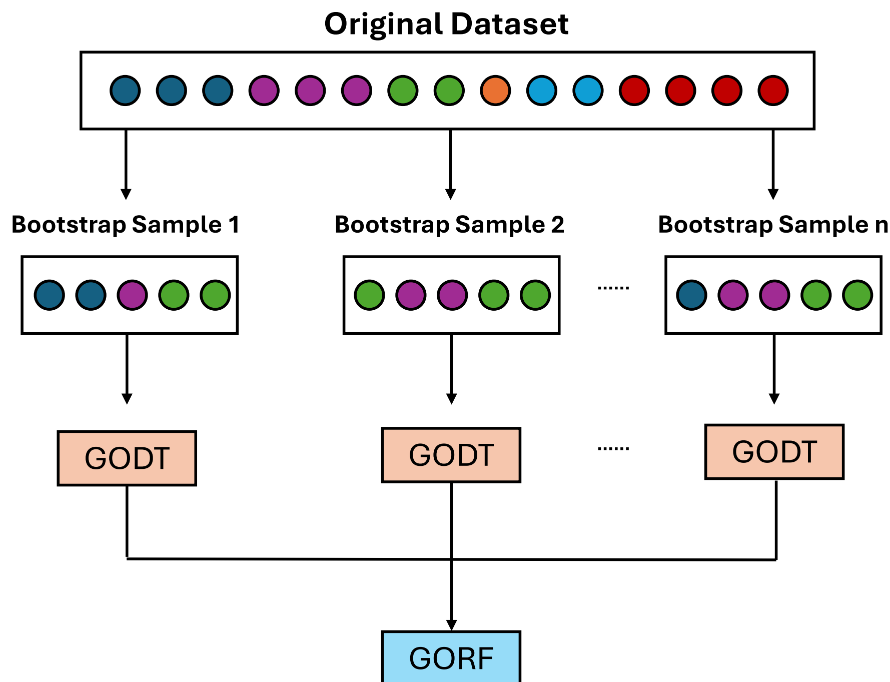


FIGURE 4.1: Ensemble of GODT to create GORF

Figure 4.1 displays the implementation of GORF through an ensemble of GODT. In the context of GORF, we construct base classifiers, in this case GODT, on each bootstrapped sample, and we aggregate the predictions to have a final prediction from the forest. The aggregation technique varies from classification and regression. In classification, we use majority voting as our aggregation technique and in regression, we use averaging as our aggregation technique. This is similar to the existing standard random forest technique.

4.3.2 Feature sub-sampling technique

The primary concept of a bagging-based random forest is that it induces randomness into the base classifier in an aim to achieve a model with low bias and low variance. One way to induce this randomness is by bagging, which we discussed above. Another way that randomness is induced is by feature sub-sampling, where we call this the *mtry* parameter. The value of *mtry* is used to inject randomness in the tree-building process. When the value of $mtry = p$, i.e., all the features in the dataset, no randomness is produced between the individual baseline classifiers (in this case the decision trees). In the standard random forest, a different subset of *mtry* is selected when splitting each non-terminal node. In a standard random forest (Breiman, 2001), a general rule of thumb is to tune the *mtry* hyperparameter and have the *mtry* hyperparameter value approximately set to $\sqrt{p}$ for classification and $p/3$ for regression (Probst et al., 2019). These are just starting values which could further be tuned. However, Fernández-Delgado et al. (2014); Zhang and Suganthan (2017) show that tuning the *mtry* parameter generally produces better performing models. So, in our case, when constructing the GORF, we follow the same procedure of the standard random forest and we tune the *mtry* hyperparameter, making sure it is approximately close to $\sqrt{p}$ for classification and $p/3$ for regression. This would help us strike the right balance between the linear combination of the features, model complexity and generalization performance. We want to have randomness in the baseline classifiers along with making sure we choose enough features to be able to build a linear hyperplane that understands the relationship in the data at each non-terminal node of the baseline classifier.

The GORF framework employs GODT as its base classifier. We therefore begin by briefly revisiting the construction of GODT for classification and regression in the following sections.

4.3.3 GODT Base Learner Recap

The Gaussian Oblique Decision Tree (GODT) serves as the base learner in our Gaussian Oblique Random Forest (GORF). A full description of the hyperplane construction, GMM-based node splitting, non-split condition, and hyperparameter choices for both classification and regression is given in Chapter 3 (see Sections 3.3.1 to 3.3.7). Here, we briefly summarise only the aspects relevant to the construction of the GORF.

In GODT, oblique decision boundaries of the form $\mathbf{w}^\top \mathbf{x} = d$ are constructed at each non-terminal node using a Gaussian Mixture Model (GMM), with the normal vector $\mathbf{w}$ and cut-point d derived from the component means, shared covariance, and prior mixture weights (see Equations (3.2)–(3.6) in Section 3.3). Splitting continues while

node purity (for classification) or node mean squared error (for regression) does not satisfy the non-splitting conditions defined in Section 3.3.5. The node-level behaviour and tree-level hyperparameters of GODT-C and GODT-R are therefore directly inherited by the ensemble.

Algorithm 8 GORF with Bagging and Feature Sub-sampling

Input: Training dataset $\mathcal{D} \in \mathbb{R}^{n \times (p+1)}$
Number of trees $n_estimators$
Bootstrap flag $bootstrap \in \{\text{True}, \text{False}\}$
Number of samples per tree $max_samples$
Number of features per tree $max_features$

Output: Ensemble model $E = \{T_1, T_2, \ldots, T_{n_estimators}\}$

1: Initialize empty ensemble: $E \leftarrow [\,]$
2: **for** $i = 1$ to $n_estimators$ **do**
3:     **if** $bootstrap = \text{True}$ **then**
4:         Sample $max_samples$ instances from $\mathcal{D}$ *with replacement* to form $\mathcal{D}_i$
5:     **else**
6:         Sample $max_samples$ instances from $\mathcal{D}$ *without replacement* to form $\mathcal{D}_i$
7:     **end if**
8:     Randomly select $max_features$ features from the p available features to form subset F_i
9:     Restrict all instances in $\mathcal{D}_i$ to the features in F_i
10:     Train decision tree: $T_i \leftarrow \text{GODT}(\mathcal{D}_i)$
11:     Add T_i to ensemble: $E \leftarrow E \cup \{T_i\}$
12: **end for**
13: **return** E

4.3.4 Hyperparameters for Gaussian Oblique Random Forest (GORF)

GORF introduces a set of forest-level hyperparameters in addition to the tree-level hyperparameters already defined for GODT in Section 3.3.4. At the tree-level, each GODT-C or GODT-R model uses the same $(\delta, \beta, M, min_samples_leaf)$ configuration as described previously. Here, we focus on the hyperparameters that are specific to the ensemble.

For GORF classification (GORF-C) and regression (GORF-R), the main forest-level hyperparameters are:

- *n_estimators*: the number of trees in the forest. Increasing this generally improves stability and accuracy up to a point, at the cost of higher computational time.

- $bootstrap$: indicates whether trees are trained on bootstrap samples (with replacement) or on subsamples without replacement. In our experiments, we always use bootstrapping with replacement.
- $max_samples$: the proportion (or number) of training instances used to grow each tree. We typically use 80% of the training data, to induce sufficient diversity.
- $max_features$: the number of randomly selected features considered at each split. For classification, we tune $max_features$ around $\sqrt{p}$, and for regression around $p/3$, following standard Random Forest practice.

4.3.5 Non-splitting Condition in GORF

Each GODT in the ensemble uses the same non-splitting criteria as defined in Chapter 3 (Section 3.3.5.1 and 3.3.5.2). For classification, node purity is measured by the proportion of the majority class, and splitting stops once this exceeds the threshold δ. For regression, splitting is halted when the node-level mean squared error (MSE) falls below δ. These stopping rules are applied independently to each tree in the forest, ensuring that GORF inherits the same interpretability and control over model complexity as the single-tree GODT. Please refer Section 3.3.5.1 and 3.3.5.2 for more information on this.

4.3.6 Initialization

Another major consideration when constructing clusters is cluster initialization. In chapter 3, we discussed cluster centroid initialization when implementing GMM. This is because, the performance of a clustering technique, which converges to a local maximum of the likelihood function, depends highly on initial cluster centroids. If a clustering technique has a poor initialization, this may lead to empty clusters, slower convergence, and a high chance of getting stuck in bad local minima (Celebi et al., 2013). Techniques like K-means converge to a local minimum, but GMM converges to a local maximum due to the Expectation-Maximization (EM) algorithm. Moreover, Panić et al. (2020) demonstrated that careful initialization when using the EM algorithm is necessary, even when applied in highly favorable settings, and performing random initialization would converge to bad critical points (Jin et al., 2016).

The cluster centroid initialization for the GODT technique provides us with consistent results at every run, provided that we are using the same sample of data. However, when constructing an ensemble, mainly GORF, we conducted an experiment to see if random initialization versus initializing the cluster centroids using Algorithm 6, has an impact on computational times and accuracy.

The Tables A.8 and A.9 in Appendix A.4 display the results for GORF classification using random initialization versus initializing using Algorithm 6 on a sample of 5 different datasets. We observed that for GORF-C, initialization using Algorithm 6 does not have a massive impact on the accuracy for normal-sized datasets. However, when working with high-dimensional datasets, initialization using Algorithm 6 largely improves the accuracy as shown in the Madelon dataset example. Moreover, initialization using Algorithm 6 compared to random initialization does add slightly to the computational time. However, on the contrary, on large and high-dimensional datasets, random initialization looks to add extra time to the computational times. This may be due to poor initialization leading to slow convergence. So, we proceed with initialization using Algorithm 6.

The three separate parameters for the GMM as part of the GORF are the mean, covariance, and the prior mixture components. And, all these three parameters require initialization as part of the implementation. Firstly, the means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are initialized using K-means with a further step of K-means centroids initialized using Algorithm 6. Secondly, for the diagonal covariance matrix $\boldsymbol{\Sigma}$, the diagonal elements are initialized as the variance of each respective feature. Finally, the prior mixture components ϕ are set to uniform for initialization, i.e., if we have $M = 2$ in our case, each component receives an initial equal weight of $\frac{1}{2}$.

What we achieve at the end is a technique to build an ensemble of oblique decision trees for classification and regression, without computationally intensive hyperplane optimization.

4.4 Experimental Results and Discussions

In this section, we will discuss further regarding the data preparation, experimental setup and the final results for both GORF for classification and regression. We refer to the Gaussian Oblique Random Forest for Classification technique as GORF-C, Gaussian Oblique Decision Tree for Classification technique as GODT-C, Gaussian Oblique Random Forest for Regression technique as GORF-R, and Gaussian Oblique Decision Tree for Regression technique as GODT-R.

4.4.1 Classification Results

In this subsection, we will look at the different machine learning datasets that we will be working with, along with the experimental setup, and the classification performance for GORF-C.

4.4.1.1 Machine Learning Datasets

We conduct our experiments on 22 benchmark datasets that have come mainly from the UCI repository (Dua and Graff, 2017), and we also use some datasets from other websites (Chang and Lin). Most of these datasets have been evaluated in detail in previous studies on oblique decision trees and other machine learning techniques. So, we use these datasets to evaluate the performance of our ensemble technique. Moreover, as we used these datasets for comparison in the Chapter 3 on GODT, it will be a good practice for us to use the same datasets for future comparisons.

Among the 22 datasets that we use, 14 are multi-class datasets while the remaining 8 are binary datasets. The details of the datasets are provided in Table 3.1. We do not perform any sort of dimensionality reduction, as we want to observe the performance of our model on small, large-scale, and high-dimensional datasets. However, a very small number of datasets have been scaled between $[-1, 1]$ at the source (Chang and Lin), so we use them in the same form without performing any further pre-processing. We split the dataset into a 75:25 train-test split, where a random subset of 20% of the training data is used for validation. We use test accuracy as our performance measure. The test accuracy looks at the total number of correct predictions on the test data out of the total number of observations in the test data as a percentage.

4.4.1.2 Experimental Setup

When we test the performance of our GORF-C model, we compare it against 4 different techniques, mainly the Standard Random Forest for Classification (RF-C) (Breiman, 2001), Classification and Regression Trees (CART) (Breiman et al., 1984c), Gaussian Oblique Decision Trees for Classification (GODT-C) from Chapter 3, and SVM. These models have been implemented in MATLAB 2019b on a computer which is Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz 3.40 GHz with 8.00 GB RAM. When it comes to the models that we implemented, the main reason we compare against CART and GODT-C is to see whether there is an improvement in performance for the RF-C from CART and whether there is an improvement in performance of GORF-C from GODT-C. However the main performance comparison of GORF-C in terms of test accuracy and computational times is against the RF-C. The main comparison of the GORF is against the standard random forest as both are tree-based ensemble techniques. The literature in the area of oblique random forest is limited and there are no established techniques with existing optimized code repository. So, we compare it against the tested and proven standard random forest by Breiman (2001).

For the RF-C technique, we use the `fitcensemble()` to construct an ensemble of CART decision trees and the function `predict()` to test the model performance. For the CART technique, we utilize the `fitctree()` to learn a single CART decision tree and

the function `predict()` to test the model performance. For the SVM we use the `fitcecoc()` to train the model and the function `predict()` to test the model performance. For RF-C, CART, and SVM, we use the Statistics and Machine Learning toolbox in MATLAB. So, CART constructs an axis-parallel decision boundary, RF-C constructs an ensemble of CART, and the remaining techniques construct a linear decision boundary. Moreover, the above techniques can be implemented on both binary and multi-class data. The SVM uses Error-Correcting Output Codes (ECOC) to extend the binary classification problem to multi-class classification task.

When we are training the GORF-C, GODT-C, RF-C, CART, and SVM techniques, we would require validation data to tune the hyperparameters. Mainly, this is a necessity for GORF-C, GODT-C, RF-C, and CART as you would have hyperparameters such as minimum leaf size, number of features to subsample, number of trees in the ensemble, and other hyperparameters. Specifically, as the GORF-C technique is constructed using multiple GODT-C, we use the validation dataset to tune the purity parameter δ and the regularization parameter β. Moreover, the GORF-C technique has hyperparameters like number of features to subsample and number of trees in the ensemble to tune.

We select the parameter combination that provides us with the highest accuracy on the validation dataset. This parameter combination is again used when training the final model. In most cases, the δ parameter is close to one. Alongside these hyperparameters, we have some further hyperparameters like minimum leaf size in GODT-C technique along with the number of trees, number of features to subsample at every split, proportion of sample to bootstrap for GORF-C, to keep it consistent with the standard random forest technique. We also record the training times of the different algorithms. For these, we run the algorithm 10 times and record the average time taken to train on the dataset. The training computation times for the different techniques are illustrated in Table 4.2. The hyperparameter tuning strategy used here is similar to section 3.4.1.3. We have one addition to this which is the parameter range for the random forest. On top of the parameters from the tree based technique, we have the forest level hyperparameters. For both Gaussian Oblique Random Forests and Standard Random Forests, we have a grid search range of 50, 100, 200, and 500 for the number of trees. We tune the $mtry$ parameter to be approximately $\sqrt{p}$ for Classification and $p/3$ for Regression. And, finally, our grid search range for the minimum leaf size is 1,5, 10 and 20. This design allows the effect of feature subsampling on ensemble diversity and predictive performance to be assessed directly. For more information on hyperparameter tuning strategy and hyperparameter ranges, please refer section 3.4.1.4 and 3.4.1.5.

Dataset	GORF-C	RFC	GODT-C	CART	SVM
	High-dimensional dataset				
Gisette	95.50	91.80	91.80	93.50	**96.80**
Madelon	**79.17**	76.83	76.00	74.83	57.33
Usps	**96.10**	95.22	91.86	87.15	91.48
	Large-scale dataset				
Acoustic	72.83	**76.51**	70.55	68.20	67.00
Combined	77.9	**82.10**	76.31	78.70	80.61
Poker	56.94	**61.01**	56.68	52.64	50.19
	Normal-sized dataset				
Abalone	55.39	**55.87**	55.15	51.16	52.67
Adult	**76.38**	76.36	**76.38**	74.39	74.22
Breast Cancer	**97.81**	94.89	**97.81**	95.62	96.35
Dermatology	**97.83**	93.48	95.65	91.30	95.65
Drug Consumption	77.49	77.49	**78.13**	73.89	17.41
Ecoli	80.95	80.95	**85.71**	78.57	79.76
Glass	72.09	**76.74**	67.44	55.81	58.14
Heart Failure	81.67	71.67	**85.00**	75.00	55.00
Ionosphere	**91.43**	90.48	84.76	85.71	86.67
Iris	**100.00**	**100.00**	**100.00**	**100.00**	**100.00**
Letter	93.46	**94.34**	82.12	78.40	84.14
Movementlibras	**82.22**	73.33	76.67	66.67	71.11
Mushroom	**100.00**	**100.00**	**100.00**	99.92	97.21
Satimages	**90.44**	88.97	89.12	86.01	87.10
Segmentation	**92.45**	90.57	88.68	83.02	86.79
Wine	**100.00**	97.22	**100.00**	94.44	91.67

TABLE 4.1: Test accuracies of the models.

4.4.1.3 Classification performance

In this section, we look at the classification performance of GORF-C when compared to other established, popular, and proven techniques like the RF-C and SVM. Specifically, we compare the test accuracies of these models in Table 4.1. In the table, we also include the accuracies for the GODT-C technique that we proposed in Chapter 3, and we also include accuracies of CART. The reason that this has been done is to give us a good indication on the performance of the ensemble as compared to a single tree.

Looking at Table 4.1, we will look at the performance of the individual models. We compare how the GORF-C is performing against RF-C. We also look at how the ensembles are performing when compared to the single tree-based techniques, i.e., GORF-C compared against GODT-C and RF-C compared against CART. Firstly, if we compare GORF-C and RF-C in isolation, we see that GORF-C performs better than RF-C in 12 out of 22 cases. The RF-C performs better in 6 out of 22 cases. They both perform exactly the same in 4 out of 22 cases. Now, if we compare the individual tree-based techniques as compared to the ensemble-based techniques, we observe that both RF-C and GORF-C have accuracies better than CART and GODT-C respectively. An interesting observation that we make is that in examples like the Abalone, Poker, Adult, Breast cancer, Satimages, and Wine datasets, the performance of GODT-C is

very similar to GORF-C. Which shows that the performance of a single-tree technique would give us a performance similar to ensemble technique in the GODT-C case.

Dataset	GORF-C	RF-C	GODT-C	CART	SVM
	High-dimensional dataset				
Gisette	1897.9786	131.9266	185.7200	4.7725	157.6600
Madelon	589.1086	10.9072	2.4563	0.2688	43.6181
Usps	119.8796	27.5523	5.7451	1.7287	3.1617
	Large-scale dataset				
Acoustic	678.9786	198.7093	19.4608	4.3866	299.2567
Combined	1127.8796	5.7043	90.0542	8.1299	372.3168
Poker	346.7380	740.5240	8.8307	0.6860	81.3286
	Normal-sized dataset				
Abalone	32.4718	3.1569	0.5214	1.0086	1.5474
Adult	5.7154	6.6409	3.0967	0.7297	784.6169
Breast Cancer	9.3929	4.3945	0.4846	0.6409	0.5246
Dermatology	14.6959	2.7227	0.0508	0.0583	1.2145
Drug Consumption	80.0657	5.6637	0.2104	0.0494	94.5625
Ecoli	5.7612	1.8013	0.4881	0.6271	1.5283
Glass	18.8164	3.9908	0.3594	0.6261	0.6638
Heart Failure	9.3600	3.5376	0.5346	0.6115	3.7935
Ionosphere	14.3184	2.6319	0.3569	0.6476	0.4570
Iris	5.4850	3.0563	1.3062	0.7477	0.5610
Letter	98.6785	11.6182	1.0739	0.0966	4.3940
Movementlibras	26.0444	3.7931	0.6789	0.6660	1.1943
Mushroom	53.0158	5.3252	0.3424	0.2271	73.3111
Satimages	268.0892	5.4701	0.6664	0.1048	0.6918
Segmentation	46.1106	3.7387	0.7485	0.6578	0.8060
Wine	5.4099	2.1302	0.1110	0.6660	0.5936

TABLE 4.2: Training times of the techniques in seconds.

On the datasets where RF-C performs better, our technique is not far behind. We particularly observe that GORF-C performs better on high-dimensional dataset. This may be due to the nature of the oblique decision tree that we use as the baseline model for the ensemble. Similarly, we observe that RF-C performs better on the large-scale datasets. When we look at the SVM with linear kernel, we can observe that it performs better on high-dimensional datasets like Gisette. However, on this dataset, the performance of the ensembles are not far away, mainly GORF-C with an accuracy of 95.50. So, in general, looking across all the datasets, GORF-C comes up as a reliable performer with consistent performance across a variety of datasets. The macro-averaged precision, recall, and F1-scores in Appendix A.5 convey the same message.

If we now refer to the training computational times of the different techniques in Table 4.2, the first observation that we clearly make is that the computational time to train

Method	Rank	Raw WSR p	Bonferroni p
RF-C	2	1.989×10^{-1}	7.955×10^{-1}
GODT-C	3	1.682×10^{-2}	6.729×10^{-2}
SVM	4	1.866×10^{-4}	7.466×10^{-4}
CART	5	7.980×10^{-5}	3.192×10^{-4}

TABLE 4.3: Wilcoxon signed-rank test p-values (raw and Bonferroni-corrected, $m = 4$) comparing GORF-C to each classifier

Method	Average Friedman Rank	Mean Accuracy (%)
GORF-C	1.80	84.91
RF-C	2.52	83.29
GODT-C	2.61	82.99
SVM	3.89	76.24
CART	4.18	79.32

TABLE 4.4: Average Friedman ranks and mean test accuracies across all datasets.

an ensemble is generally higher when compared against constructing a single tree. This is because, the ensemble learner consists of multiple repetitions of this baseline model. We observe when it comes to computational times that RF-C is quicker than GORF-C in most cases but not always, based on Table 4.2. However, one consideration here is that the RF-C code that we used from MATLAB toolbox is fully optimized for efficiency. We definitely have an opportunity here to optimize our code to remove some bottlenecks and achieve computational efficiency. However, having said that, the computational time of the GORF-C is decent and achieves us the performance in Table 4.1 in decent computational times.

We perform an experiment to compare the change in performance with change in ensemble size for GORF-C and RF-C in Appendix A.10, Tables A.18 to A.20. We compare this on three datasets, mainly Madelon (high-dimensional), USPS (large-scale), and Satimages (normal-sized) datasets. On the high-dimensional dataset, we observe that the performance of GORF-C starts at a good accuracy compared to RF-C, and ends up at a higher accuracy. On the large-scale dataset, the performance of the GORF-C and RF-C does not vary a lot. They both start at the same level and end at a similar level. On the normal-sized dataset, the GORF-C again starts at a better points ands at a better point when looking at the accuracy. So, we observe based on this experiment, that the GORF-C may require lower ensemble size to achieve the same performance as the RF-C, potentially leading to computational time saving.

4.4.1.4 Statistical Comparison

Similar to chapter 3, we use the Wilcoxon signed-rank test to determine the significance amongst each pair of the classifiers used. In chapter 3, we used OC1 as a comparison model. Due to this, we have had to remove a few datasets when performing the test, as OC1 failed to return results on some datasets. However, as we

are not using OC1 as a comparison model in this chapter, we do not come across such a situation. So, we use results on all of the datasets for the tests. For the Wilcoxon signed-rank test, we have the following hypotheses:

- **Null Hypothesis (H_0)**: There is no significant difference in the performance (e.g., accuracy) between GORF-C and the other classifiers.
- **Alternative Hypothesis (H_1)**: GORF-C performs significantly different than the other classifiers

Table 4.3 presents the results for the Wilcoxon signed-rank test, which was used to conduct a pairwise comparison of GORF-C and each of the other classifiers used. The test was performed using a two-sided hypothesis to assess whether there were statistically significant differences in accuracy distributions. A significance level of $\alpha = 0.05$ was used to determine statistical significance. As shown in the Table 4.3, GORF-C exhibits significant differences in performance compared to GODT-C, CART, and SVM, as indicated by their p-values being below the 0.05 threshold. However, the comparison with RF-C did not yield a statistically significant result, suggesting that the observed performance differences between GORF-C and RF-C are not strong enough to reject the null hypothesis. After Bonferroni correction, all the adjusted p-values are below 0.05 except for RF-C and GODT-C, so we do not reject the null hypothesis for comparisons with RF-C and GODT-C. Future work may explore a broader set of datasets to determine whether the performance of RF-C and GODT-C consistently approaches that of GORF-C.

Table 4.4 shows the average Friedman ranks and mean classification accuracies of all models across the evaluated datasets. A lower Friedman rank indicates consistently higher performance across datasets. GORF-C achieved the lowest average rank (1.80) and the highest mean accuracy (84.91%), suggesting that it outperforms the other classifiers both in average predictive performance and stability. These findings further support the selection of GORF-C as a strong and competitive baseline classifier in this comparative evaluation.

4.4.2 Regression Results

In this subsection, we will look at the performance of our Gaussian Oblique Random Forest for Regression (GORF-R) technique in comparison to established popular techniques like the CART-based Random Forest for Regression (RF-R) (Breiman, 2001). Earlier in this chapter, we looked at the modifications that we make to the GORF-C to make it suitable for a regression task, mainly, adjustments to the architecture like the non-split condition, hyperparameters, and etc. In this subsection, we will also look

into the machine learning datasets that we will be using, along with the experimental setup and performance on regression tasks.

4.4.2.1 Machine Learning Datasets

We conduct our experiments on 20 datasets that are a combination of open-source datasets from the UCI repository (Dua and Graff, 2017) and other datasets from industrial partners. The details of the datasets are provided further in the Table 3.8. We use the same datasets that we used for testing the GODT-R technique from Chapter 3, as this would make it easy for us to compare the GORF-R technique with GODT-R technique. In order to test the performance of our model, we work with different sizes of datasets, both large-scale and normal-sized. In order train and test the performance of our technique, we split the dataset into a 75:25 train-test split, where a random subset of 20% of the training data is used for validation. To measure how well our technique is performing in comparison to other machine learning techniques, we use performance metrics like the Mean Absolute Error (MAE). Furthermore, we do not perform any dimensionality reduction on the data that we use. However, we do perform some pre-processing to remove irrelevant features like ID's, timestamps, and other features that are not helpful for the prediction of the targets. This would ensure that the model is being trained on data that is most relevant.

4.4.2.2 Experimental Setup

Our experimental setup remains similar to the previous chapters. We implement the GORF-R technique in MATLAB 2019b on a computer which is Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz 3.40 GHz with 8.00 GB RAM. We compare our GORF-R method against our GODT-R technique from Chapter 3, Random Forest Regression (RF-R), and Classification and Regression Trees (CART). Both these techniques are available as part of MATLAB's Statistics and Machine Learning Toolbox, and have been implemented in the same computer configuration. We specifically chose Breiman (2001) Random Forest (RF-R) as our major comparison because this is a very well-established and popular technique in the Machine Learning and Data Science community.

In order to implement the RF-R technique in MATLAB, we use the `fitrensemble()` to construct an ensemble of CART decision trees and the function `predict()` to test the model performance. For the CART technique, we utilize the `fitrtree()` to learn a single CART decision tree and the function `predict()` to test the model performance. For the SVM we use the `fitrsvm()` to train the model and the function `predict()` to test the model performance. For RF-R, CART, and SVM, we use the Statistics and Machine Learning toolbox in MATLAB. So, CART constructs an axis parallel decision

boundary, RF-R constructs an ensemble of CART, and the SVM constructs a decision boundary using the Radial Basis Function (RBF) kernel.

Dataset	GORF-R	RF-R	GODT-R	CART	SVM
	Large-scale dataset				
mcev10	2.7905	2.7942	2.8775	2.9675	2.7636
mcev100	0.9413	0.8843	1.0809	0.9895	0.9184
mcev1000	0.4133	0.2832	0.6768	0.5315	0.3465
bejoingpm	53.17508	53.57647	52.3552	52.9147	61.7168
	Normal-sized dataset				
electrical grid	0.023991	0.024848	0.0266	0.0249	0.0240
combined cycle	3.8770	3.705549	3.3705	3.1326	9.8323
air quality	0.152023	0.135759	0.1655	0.1143	6.8039
bias correction	0.9045	0.967931	0.9863	1.1327	2.5138
gas turbine	0.58842	0.662541	0.8551	0.9775	11.7125
parkinsons	0.033187	0.029498	0.0379	0.0320	0.0517
airfoil self noise	3.637154	2.309608	4.5527	3.9664	5.4898
concrete	6.9027	5.4933	9.8687	7.6510	12.1005
can combustion	0.0533	0.0761	0.0708	0.0811	0.0936
energy efficiency	1.480398	1.37714	1.4787	1.1785	1.8973
Istanbul stock exchange	0.0046	0.0046	0.0052	0.0052	0.0061
forest fires	18.6758	18.12884	16.8986	18.4209	9.7672
3d aeroplane	0.8144	0.9844	0.9471	1.1973	1.0948
auto mpg	2.675558	2.73879	2.7072	3.0593	4.5314
computer hardware	13.93087	12.20959	12.7967	33.9584	51.9094
ale	0.241396	0.285731	0.1795	0.2034	0.3149

TABLE 4.5: This table presents the (Mean Absolute Error) MAE for each technique on the test dataset

We also need validation datasets to tune the hyperparameters of the techniques that we are implementing. These parameter combinations are used when training the final model to achieve the best performance. GORF-R, GODT-R, RF-R, CART, and SVM all require validation data for hyperparameter tuning. Some hyperparameters for GORF-R that require tuning include, the purity parameter δ, the regularization parameter β, minimum leaf size, number of trees, and number of features to consider at each split. The RF-R requires minimum leaf size, number of trees, and number of features, amongst others, to tune. We select the parameter combination that provides us with the highest accuracy on the validation dataset. This parameter combination is again used when training the final model. For more information on hyperparameter tuning strategy and hyperparameter ranges, please refer section 3.4.1.3, 3.4.1.4, and 4.4.1.2.

We will now look at the performance of all of the techniques on datasets suitable for regression tasks. We also record the time it takes to train each technique. We run the

algorithms 10 times and record the average time taken to train on each dataset. The training computation times for the different techniques are illustrated in Table 4.6.

Let us start by comparing the performance of the techniques on different datasets, as presented in Table 4.5. If we first solely compare the performance of the ensemble against the single-tree technique, we can begin with comparing the performance of GORF-R against GODT-R. In general, the performance of the ensemble is better than a single-tree through the reduction of the MAE. This is also reflected in the CART technique as compared to RF-R. There are a couple of instances (like the bejoingpm and combined cycle example) where the MAE of GORF-R slightly increases compared to GODT-R. This may potentially be due to us overfitting when constructing a single decision tree, which again makes the case for creating an ensemble. Even RF-R replicates a similar trend. This emphasizes the importance of ensemble techniques that are more robust in comparison to constructing a single tree.

Dataset	GORF-R	RF-R	GODT-R	CART	SVM
	Large-scale dataset				
mcev10	98.5605	64.39905	1.322474	0.205576	51.45576
mcev100	110.5768	71.6574	0.821766	0.079358	51.96682
mcev1000	102.5033	70.7543	1.630908	1.114834	39.50536
bejoingpm	48.0377	11.37719	0.08671	4.3329	85.90265
	Normal-sized dataset				
electrical grid	30.1905	6.113394	1.977073	0.579835	2.605253
combined cycle	40.5768	2.183155	2.916071	0.055392	3.877759
Air Quality	41.1948	0.794389	2.309924	1.048259	2.019167
bias correction	10.8679	2.015208	1.2444	0.06893	1.4361
gas turbine	27.381	0.902252	1.140194	0.18974	1.694454
parkinsons	2.7603	5.160458	1.788394	0.033035	1.762865
airfoil self noise	11.21336	3.44826	0.233709	0.018554	0.264911
concrete	22.2668	0.2908	0.196753	0.205458	0.058164
can combustion	0.3719	1.4255	0.427625	0.167912	0.55365
energy efficiency	10.0098	3.837124	0.433456	0.030224	1.097657
Istanbul stock exchange	14.4114	0.2496	0.726321	0.550912	0.079763
forest fires	14.5812	2.160955	0.098277	0.042507	4.616932
3d aeroplane	0.0837	0.0654	0.333668	0.866374	0.284676
auto mpg	17.9717	2.6879	0.149007	0.112017	4.582627
computer hardware	5.4786	1.606171	1.014906	0.135679	0.319657
ale	4.1111	1.658027	0.065921	0.04196	0.1193

TABLE 4.6: Training Times for each technique in seconds

There seems to be an extremely balanced performance when comparing the GORF-R technique to the RF-R technique. The GORF-R performs better than RF-R in 9 out of the 20 datasets, and the RF-R performs better than GORF-R in 10 out of 20 datasets, and the both techniques perform exactly the same on the Istanbul stock exchange dataset. Even when RF-R performs better, the performance of GORF-R is not too

distant. The GORF-R technique does not perform poorly in any of the 20 datasets. Hence, we can see that we have an oblique random forest technique for regression tasks that performs decently well across multiple datasets.

If we now refer to the training computational times of the different techniques in Table 4.6, the computational time to train an ensemble is longer than training a single tree. Additionally, training an oblique random forest seems to take longer. When we compare the computational times of GORF-R against RF-R, excluding a couple of cases, GORF-R generally takes longer than RF-R. One consideration to make here is that the RF-R technique that we compare has been implemented in C or C++ for highest performance optimization and is regularly optimized. We have an opportunity here to optimize our code to remove potential bottlenecks and achieve computational efficiency as part of future development.

We managed to implement the GORF technique for both classification (on a combination of binary, multi-class) and regression datasets. We observe that on high-dimensional, low-dimensional, and normal-sized datasets, the GORF performs consistently better or in close proximity to the best model in computational accuracy. Considering the computational times, GORF has reasonable computational times with a high potential for code optimization as part of future work. This exhibits that we have a versatile oblique random forest technique with the capability to have a decent accuracy and computational times across different varieties of datasets.

Chapter 5

Gaussian Mixture Model Quadratic Random Forest for Classification

5.1 Introduction

In Chapter 3, we explored the concept of building a decision tree technique to constructs linear hyperplanes at each non-terminal node. We explored this using Gaussian Mixture Models, to construct the Gaussian Oblique Decision Trees (GODT) technique for both classification and regression. In Chapter 4, we extended this technique to construct an ensemble of GODT techniques, building the Gaussian Oblique Random Forest (GORF) technique for both classification and regression. We observed that both the GODT and GORF techniques have good accuracies across a wide variety of datasets.

While the area of axis-parallel decision trees and oblique decision trees are widely explored, as we performed our literature review, we understood that the area of quadratic decision trees was relatively under-explored compared to axis-parallel decision tree techniques or even oblique decision tree techniques. Moreover, we do not come across standard implementations of quadratic decision tree techniques, which supports our findings here. This has motivated us to explore a potential technique that constructs quadratic decision boundaries at each non-terminal node to implement the decision tree.

The decision tree is a hierarchical technique that implements recursive partitioning of the input space into much smaller subsets which are pure (i.e., data corresponding to a single class). The decision tree technique has been widely popular for over four decades, due to the simplicity in its implementation and interpretation. Some popular applications of the decision tree techniques are in network security (Benferhat et al., 2013), computer vision (Nowozin et al., 2011), bio-informatics (Chu and Li, 2014;

Remita et al., 2017), medical diagnosis (Pereira et al., 2017), credit risk (Baesens et al., 2003), and many more examples.

The decision tree technique is also popular due to its simplicity and interpretability. When constructing a decision tree technique, at each non-terminal node, an impurity measure is optimized to achieve sub-space of the initial data that is pure. This would then provide us with a model that would give good prediction performance when presented with unseen test data, that requires target values being predicted. The decision tree technique is mainly divided into *axis-parallel* decision trees to construct orthogonal decision boundaries and *oblique* decision trees to construct linear decision boundaries. However, we take this a step further and propose *quadratic* decision trees to construct quadratic decision boundaries.

The *axis-parallel* decision tree technique constructs orthogonal decision boundaries by considering only a single feature during the node-splitting process. Every non-terminal node is assigned a split rule that is a binary-valued function of the feature vector. At each non-terminal node, the feature to split on and the value to split on is decided based on the impurity measure. The combination of feature and value to split on, leads to a lower impurity score to be selected as the split condition. When a node majorly contains data corresponding to a single class, it is converted to a terminal or a leaf node, where the majority class value is assigned as the class of the leaf node.

Some popular and widely used examples of the axis-parallel decision tree techniques include Iterative Dichotomiser 3 (ID3) (Quinlan, 1986), Classification And Regression Trees (CART) (Breiman et al., 1984b), and C4.5 (Quinlan, 2014). The impurity measures used for a classification task for ID3, Classification and Regression Tree (CART), and C4.5 are Information gain, Gini index, and Gain ratio respectively, which are used to greedily optimize the features to split on and the values to split on. Axis-parallel decision trees can also be implemented for regression tasks, where measures such as mean squared error or variance reduction is used to decide whether to split a node any further or to convert it into a leaf node.

Axis-parallel decision trees are easy to interpret due to the simpler split rules and are quicker to compute as it is faster to evaluate a single feature, and search for the best threshold. However, if we have features depicting linear or non-linear relationships requiring linear or curved decision boundaries, axis-parallel decision trees would generate deep trees and potentially a sub-space where the data is over-fragmented. Looking at all the features collectively would yield us a better understanding of the hidden relationships between them. As Yıldız and Alpaydın (2012) suggests, axis-parallel decision trees using a single feature to split a space with an orthogonal boundary would result in larger trees and poor generalization.

Oblique decision trees, in contrast to axis-parallel decision trees, would consider a linear combination of features at each non-terminal node. The oblique decision tree

constructs a linear hyperplane rather than an orthogonal hyperplane at each non-terminal node to separate the classes. This is done in an aim to improve the purity of the nodes incrementally at subsequent nodes. Murthy et al. (1994) shows in their contributions that when the underlying problem requires oblique splits, using axis-parallel splits would lead to less accurate trees. Moreover, the axis-parallel trees can provide us with complex tree structures and increased computational cost, when the inherent decision boundaries are not axis-parallel (Yang et al., 2019).

Some popular techniques in the area of oblique decision trees include CART-Linear Combination (CART-LC) (Breiman et al., 1984c) which was a variant of the CART technique that constructs CART with linear combination. Another popular technique includes Oblique Classifier 1 (OC1) (Murthy et al., 1994), which is an extension to CART-LC, that proposes to use random restarts and perturbations to escape the local minima. In Chapter 3, we also proposed an oblique decision tree technique called Gaussian Oblique Decision Tree (GODT) for classification and regression, that uses Gaussian mixture model clustering to construct oblique hyperplanes. We observe that this has a good performance across various datasets.

At each non-terminal node, oblique decision trees use linear combinations of features, which means that they assume that the decision boundary is linear. Even though this may allow them to capture the interactions between features better than axis-parallel decision boundaries, they may fall short in representing inherently curved or more complex boundaries. By incorporating quadratic terms into the decision function, *quadratic decision trees* allow for a direct modelling of curved boundaries. This is very useful in scenarios where the true decision boundary is non-linear, potentially reducing the number of splits required compared to an oblique decision tree.

While quadratic decision tree techniques address some limitations of oblique decision trees, they also bring in new challenges. A potential quadratic decision tree requires estimation of a symmetric matrix $\mathbf{Q}$, a weights vector $\mathbf{w}$, and a bias term d. This is clearly an increase in the number of hyperparameters that require estimation as compared to an oblique decision tree. Moreover, the higher number of hyperparameters increase the risk of overfitting, particularly in high-dimensional spaces or when the training dataset is small. We will explore the idea of an ensemble of these quadratic decision trees to address this issue later in the chapter. Although oblique decision trees are computationally demanding when compared to axis-parallel decision trees due to the need to find optimized linear hyperplanes, it is still simpler to find a linear hyperplane as compared to a quadratic decision boundary, as the latter would involve search over a larger parameter space. So, these are some considerations to make when proposing a quadratic decision tree technique.

To provide a concise overview, in Chapter 2 and Chapter 3, we have explored the standard decision tree technique (e.g., ID3, CART, and C4.5) where at each

non-terminal node, only one feature is considered at a time (e.g., $\mathbf{x}_i < \theta, \mathbf{x}_i \geq \theta$). In Chapter 3, we also considered the oblique decision tree technique e.g., Gaussian Oblique Decision Tree, that we propose, which considers a linear combination of features at each non-terminal node (e.g., $\mathbf{w}^\top \mathbf{x} < d$ or $\mathbf{w}^\top \mathbf{x} \geq d$) to generate linear decision boundaries or a linear hyperplane, where $\mathbf{x}$ is the feature vector, $\mathbf{w}$ is the vector corresponding to the coefficients of the linear term, and d is the cut-point. Here, we take this forward to propose a decision tree technique that generates a quadratic decision boundary or a hypersurface of the form $\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{w}^\top \mathbf{x} + b = 0$ where $\mathbf{x}$ is the feature vector, $\mathbf{Q}$ is the symmetric matrix representing the quadratic term, $\mathbf{w}$ is the vector corresponding to the coefficients of the linear term, and b is the bias term that adjusts the position of the decision boundary in the feature space.

We propose a novel quadratic decision tree technique called Gaussian Quadratic Decision Tree (GQDT) for classification that balances good accuracy with decent computational efficiency, while also ensuring robustness and scalability across a different variety of datasets. We also take this a step further to propose an ensemble of GQDT, to propose the Gaussian Quadratic Random Forest (GQRF) for classification. Our contributions are as follows:

1. Through the literature review, we observe that quadratic decision tree techniques have been very rarely explored. So, we propose a Gaussian Mixture Model (GMM)-based technique that constructs a quadratic decision tree.

2. Our process of constructing a quadratic decision boundary is fairly quick, and the computational times are very close to its oblique counterpart, taking into account that this has an increase in the number of hyperparameters that require estimation.

3. We demonstrate that the proposed technique has a decently high accuracy, and performs similar to or better than existing techniques in most cases tested here, including the GODT technique that we proposed in Chapter 3. Furthermore, we demonstrate this on a combination of large-scale, high-dimensional, and normal-sized datasets. We also show this performance on a mix of binary and multi-class classification tasks.

4. Finally, we extend this technique to construct an ensemble of quadratic decision tree, i.e., an ensemble of GQDT called Gaussian Quadratic Random Forest (GQRF) for classification and we demonstrate that it has a good accuracy when compared to other ensemble techniques like the Standard Random Forest (RF) (Breiman, 2001) and Gaussian Oblique Random Forest (GORF) proposed in Chapter 4.

5.2 Related Work

In this section, we briefly discuss the literature of decision tree techniques that we compare in our experiments section. We have discussed most of these techniques in chapters 2, 3, and 4, so we briefly discuss these here. We discuss the CART technique by Breiman et al. (1984c) and the Standard Random Forest technique by Breiman (2001).

Both CART and Standard Random Forests are well established tree-based algorithms that serve as important benchmarks for the proposed GQDT-C and GQRF-C approaches. Their detailed formulations, training procedures, and mathematical foundations have been presented previously in Chapter 3. This section focuses on highlighting the aspects of these methods that are most relevant to our work.

5.2.1 CART

Classification and Regression Tree (CART) (Breiman et al., 1984c) constructs binary decision trees using axis-parallel splits based on impurity measures such as the Gini index for classification and mean squared error for regression. The resulting partitions are rectangular regions in the feature space, which makes the model highly interpretable and efficient to train. However, these axis-aligned boundaries can be restrictive when the true decision surfaces are oblique or non-linear. The GODT-C method proposed in chapter 3 builds oblique hyperplanes that can represent more flexible partitions while maintaining the interpretability of tree structures. Please refer chapter 3, section 3.2.1 for more information on CART.

5.2.2 Random Forests

Random Forests (Breiman, 2001) extend the CART framework through ensemble learning. Multiple CART trees are trained on bootstrap samples of the data, and their predictions are aggregated, typically by majority voting for classification or averaging for regression. This bagging approach reduces variance and enhances generalization performance, but at the cost of interpretability and an increase in model complexity. Since each tree in a standard Random Forest still relies on axis-parallel splits, it may not capture complex class boundaries effectively. In contrast, the proposed GORF, which is an ensemble of GODT technique proposed in chapter 3, retains the stabilizing effect of bagging while introducing oblique, geometry-aware splits that can model more intricate decision surfaces. Please refer chapter 2, section 2.5 for more information on Bagging and Random Forests.

5.2.3 Summary

In summary, CART and Random Forests provide strong baselines for tree-based modelling and ensemble learning. Their simplicity and robustness makes them valuable points of comparison. However, their reliance on axis-parallel partitions limits their ability to capture correlations between features and complex, high-dimensional boundaries. The GODT methods developed in this thesis address these limitations by generalizing the splitting mechanism to use data-adaptive hyperplanes, thereby improving flexibility without compromising the core advantages of tree-based models. Please refer chapter 2 section 2.5 and chapter 3 section 3.2.1 for more information on CART and Ensemble Learning.

5.3 The Proposed Method

In this section, we introduce the methodology of two techniques that are related together. Firstly, we introduce the novel Gaussian Quadratic Decision Tree technique for Classification (GQDT-C), that looks at constructing decision trees, that have a quadratic decision boundary (second-order polynomial function of the features) rather than axis-parallel or oblique. At each non-terminal node, we construct a quadratic decision-boundary or a quadratic hypersurface to split the node into pure binary nodes. The quadratic hypersurface would generate non-linear splits to separate the nodes. We utilize the Gaussian Mixture Model (GMM) to learn the coefficients of the hypersurface. We discuss this in more detail in later sections. We take this single non-linear decision tree technique a step further by constructing an ensemble of these decision trees. We call this the Gaussian Quadratic Random Forest technique for Classification (GQRF-C). Due to this being a novel area, the literature is very limited. Moreover, non-linear decision boundary decision tree techniques are very rare. So, we compare our work with other tree-based techniques. The details of how to construct GQDT-C technique is described in Algorithm 10 and GQRF-C technique is described in Algorithm 11. This shows the architecture and illustration of GQDT-C and GQRF-C. The algorithm is very similar to Algorithm 5 in Chapter 3 and Algorithm 8 in Chapter 4, with the difference being in the outputs used from GMM to construct the quadratic decision tree.

The construction of GQDT-C has multiple dependencies like the generation of the quadratic decision boundary, hyperparameters involved, non-split condition, and other dependencies like the initialization of GMM parameters. Furthermore, we also have considerations when constructing the ensemble, like the feature sub-sampling, bootstrapping and etc. So, we will look at these in the next subsections.

5.3.1 Gaussian Mixture Model

A Gaussian Mixture Model (GMM) is a parametric probability density function represented as a weighted sum of Gaussian component densities (Reynolds, 2009). Let Z follow a mixture of Normal distributions with mean $\boldsymbol{\mu}_j$, covariance $\boldsymbol{\Sigma}_j$, and component weights ϕ_j for the multi-dimensional model for $j = 1, ..., M$ where M corresponds to the number of components. Please refer Chapter 3, section 3.3.1 for more information on GMM.

The parameters in a GMM are estimated using Expectation-Maximization (EM) iteratively. Our aim is to construct a quadratic decision boundary, of the form $\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{w}^\top \mathbf{x} + b = 0$. We can observe from Algorithm 10 that GQDT-C technique uses GMM to first construct soft clusters and then use the parameter estimates from the GMM to construct the quadratic decision boundaries. Referring back to equation 3.2, based on the algorithm 10, for our method with $M = 2$ we have two mean vectors $\boldsymbol{\mu_1}$ and $\boldsymbol{\mu_2}$, we have two covariance matrices $\boldsymbol{\Sigma_1}$ and $\boldsymbol{\Sigma_2}$, and we have ϕ_1 and ϕ_2 which describe the prior probabilities of the components. In the case of GQDT-C, $\mathbf{Q}$, $\mathbf{w}^\top$, and b are calculated using the outputs of GMM.

5.3.2 Calibration of the Gaussian Mixture Model

At each non-terminal node of the GQDT-C, the data reaching the node are modelled using a two-component Gaussian mixture model (GMM). The GMM is fitted to the feature matrix $\mathbf{X} \in \mathbb{R}^{m \times p}$ using maximum likelihood estimation via the Expectation–Maximization (EM) algorithm, where m corresponds to the number of observations at a non-terminal node.

We assume a mixture model of the form

$$p(\mathbf{x}) = \sum_{j=1}^{2} \phi_j \, \mathcal{N}(\mathbf{x} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j),$$

where ϕ_j are the mixture weights, $\boldsymbol{\mu}_j$ are the component means, and $\boldsymbol{\Sigma}_j$ are component-specific covariance matrices.

The EM algorithm alternates between computing posterior responsibilities $\gamma_{ij} = p(z_i = j \mid \mathbf{x}_i)$ in the E-step and updating the mixture parameters in the M-step. The mixture weights and means are updated as

$$\phi_j = \frac{1}{m} \sum_{i=1}^{m} \gamma_{ij}, \qquad \boldsymbol{\mu}_j = \frac{\sum_{i=1}^{m} \gamma_{ij} \mathbf{x}_i}{\sum_{i=1}^{m} \gamma_{ij}}.$$

Unlike the linear GODT formulation in Chapter 3, GQDT-C employs *component-specific full covariance matrices*, estimated as

$$\boldsymbol{\Sigma}_j = \frac{\sum_{i=1}^{m} \gamma_{ij}(\mathbf{x}_i - \boldsymbol{\mu}_j)(\mathbf{x}_i - \boldsymbol{\mu}_j)^\top}{\sum_{i=1}^{m} \gamma_{ij}} + \beta \mathbf{I},$$

where $\beta > 0$ is a regularization parameter added to ensure numerical stability and positive definiteness of the covariance matrices.

The EM procedure and parameter updates follow standard formulations (see Bishop, 2006a; Reynolds, 2009) and are identical to those used in Chapter 3, except for the use of class-specific full covariance matrices.

Given the calibrated GMM parameters $(\boldsymbol{\mu}_1, \boldsymbol{\mu}_2, \boldsymbol{\Sigma}_1, \boldsymbol{\Sigma}_2, \phi_1, \phi_2)$, we now describe how these parameters induce a quadratic decision boundary at each non-terminal node.

5.3.3 Hypersurface Generation

Unlike Chapter 3, where a shared covariance assumption leads to a linear decision boundary, the present setting allows component-specific covariance matrices, resulting in a quadratic hypersurface. In this section, we explore incorporating quadratic decision boundaries with a second-degree polynomial term into the tree-building process. This allows the decision tree to create non-linear, complex decision boundaries in the feature space, which would make GQDT-C suitable for datasets where the data cannot be split efficiently using neither a linear nor axis-parallel decision boundary.

The aim of GQDT-C technique is to maximize the class separation in the data, i.e., to ensure the quadratic decision boundary splits the data to achieve classes either side of the decision boundary that eventually correspond to a single class. The reason we aim for this is so that when presented with new test data, the data would go down the hierarchy of the decision tree to allocate the data points to the appropriate class, leading to improved prediction accuracy.

The hypersurface constructed by GQDT-C has the equation in the form

$$\mathbf{x}^\top \mathbf{Q} \mathbf{x} + \mathbf{w}^\top \mathbf{x} + b = 0, \tag{5.1}$$

where:

- $\mathbf{x}$ is the feature vector
- $\mathbf{w}$ is the vector corresponding to the coefficients of the linear term.

- $\mathbf{Q}$ is the symmetric matrix representing the quadratic term
- b is the bias term, that determines the position of the decision boundary.

When constructing the hypersurface, the $\mathbf{Q}$ controls the non-linear shape of the boundary, and the $\mathbf{w}$ controls the linear part of the hypersurface that influences the orientation and the direction of the boundary in space. Once we have the quadratic decision boundary, when presented with new data observations (test data), we aim to assign data to either side of the hypersurface depending on the following:

$$\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{w}^\top \mathbf{x} + b \geq 0 \quad \text{or} \quad \mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{w}^\top \mathbf{x} + b < 0. \tag{5.2}$$

Let us now look at how the coefficient of the quadratic term, coefficient of the linear term, and the bias term are calculated. We have discussed earlier that at each non-terminal node, we implement the GMM. At every run of the GMM, this would return us the means of the two clusters created, $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$, along with the full covariance matrix for each of the two components, $\boldsymbol{\Sigma}_1$ and $\boldsymbol{\Sigma}_2$, and the prior probabilities of the components ϕ_1 and ϕ_2. We will use this information to calculate the values of $\mathbf{Q}$, $\mathbf{w}$, and b (Hart et al., 2000, Chapter 2, Section 2.6.3):

Unlike the linear case in Chapter 3, where a shared covariance matrix leads to a linear decision boundary, the use of component-specific covariance matrices results in a quadratic decision boundary. For completeness, we now derive the resulting quadratic hypersurface.

5.3.3.1 Derivation of the quadratic hypersurface

Assumptions: We assume the node data follow a two-component Gaussian mixture model with component-specific full covariance matrices. The covariance matrices are regularized so that $\boldsymbol{\Sigma}_j$ are symmetric positive definite (hence invertible) for $j = 1, 2$.

The quadratic decision boundary arises from the Bayes-optimal rule for two Gaussian components, as in quadratic discriminant analysis (QDA). This derivation follows the standard form of quadratic discriminant analysis for Gaussian class-conditional densities with unequal covariance matrices.The decision boundary is defined by equality of log-posterior probabilities:

$$\log p(z = 1 \mid \mathbf{x}) - \log p(z = 2 \mid \mathbf{x}) = 0.$$

Algorithm 10 GQDT(Dataset, Purity Condition, Regularization Parameter)

Input: Training data $\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$, purity condition value δ, regularization parameter value β

Output: Fully constructed decision tree, cluster centroids $\boldsymbol{\mu}_1^{(j)}$ and $\boldsymbol{\mu}_2^{(j)}$, covariance matrices $\boldsymbol{\Sigma}_1^{(j)}$ and $\boldsymbol{\Sigma}_2^{(j)}$ for $j = 1, ..., k$ where k is the total number of layers in the decision tree.

1: Initialise an empty tree **T** and an empty set of nodes Ψ
2: Assign all data in $\mathcal{D}$ to the *RootNode* and insert the *RootNode* into Ψ
3: **if** the *RootNode* satisfies the non-splitting condition **then**
4: Convert the *RootNode* into a *LeafNode* and assign the majority class label
5: **return T**
6: **else**
7: **for** $i = 1$ to k **do** ▷ for a sufficiently large k
8: Select a *Node* N from Ψ
9: Let the feature matrix in N be $\mathbf{X} \in \mathbb{R}^{m\times p}$ and the corresponding labels be $\mathbf{y} \in \mathbb{R}^{m\times 1}$ for $m \subseteq n$
10: **if** $m \leq m_{\min}$ **then**
11: Convert N into a *LeafNode* and assign the majority class
12: **continue**
13: **end if**
14: Initialise the centroids for the Gaussian Mixture Model (GMM) using K-means on the data in N, where the initial K-means centroids are obtained via Algorithm 6
15: Fit a 2-component Gaussian mixture model to **X** by maximum-likelihood estimation using the EM algorithm, with full covariance matrix $\boldsymbol{\Sigma}_1^{(j)}$ and $\boldsymbol{\Sigma}_2^{(j)}$ per component and regularization parameter β added to each diagonal entry
16: Extract the estimated mixture weights π_1, π_2, component means $\boldsymbol{\mu}_1, \boldsymbol{\mu}_2$, and full covariance matrix $\boldsymbol{\Sigma}_1^{(j)}$ and $\boldsymbol{\Sigma}_2^{(j)}$
17: **for** $j = 1, 2$ **do** ▷ for the two mixture components
18: **if** the non-splitting condition is satisfied in the child region **then**
19: Convert the non-terminal *Node* into a *LeafNode* and assign the majority class label
20: **else**
21: Add the non-terminal *Node* to Ψ
22: **end if**
23: **end for**
24: **end for**
25: **end if**

Substituting the Gaussian densities and expanding the quadratic forms in $\mathbf{x}$ yields a

second-order polynomial. Grouping the quadratic, linear, and constant terms gives a decision function of the form

$$\log \frac{p(z=1 \mid \mathbf{x})}{p(z=2 \mid \mathbf{x})} = \log \frac{\phi_1}{\phi_2} - \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_1)^\top \boldsymbol{\Sigma}_1^{-1}(\mathbf{x}-\boldsymbol{\mu}_1) + \frac{1}{2}(\mathbf{x}-\boldsymbol{\mu}_2)^\top \boldsymbol{\Sigma}_2^{-1}(\mathbf{x}-\boldsymbol{\mu}_2). \quad (5.3)$$

Expanding the quadratic forms yields

$$-\frac{1}{2}\mathbf{x}^\top (\boldsymbol{\Sigma}_1^{-1} - \boldsymbol{\Sigma}_2^{-1})\mathbf{x} + (\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2)^\top \mathbf{x} + (w_{10} - w_{20}), \quad (5.4)$$

which is a second-order polynomial in $\mathbf{x}$. Grouping terms gives the quadratic decision function

$$\mathbf{x}^\top \mathbf{Q}\mathbf{x} + \mathbf{w}^\top \mathbf{x} + b = 0.$$

We now identify $\mathbf{Q}$, $\mathbf{w}$, and b in terms of the fitted GMM parameters (Hart et al., 2000, Chapter 2, Section 2.6.3).

Quadratic term: Define

$$\mathbf{W}_j = -\frac{1}{2}\boldsymbol{\Sigma}_j^{-1}, \qquad j = 1, 2. \quad (5.5)$$

Then the quadratic coefficient matrix is

$$\mathbf{Q} = \mathbf{W}_1 - \mathbf{W}_2 = -\frac{1}{2}\left(\boldsymbol{\Sigma}_1^{-1} - \boldsymbol{\Sigma}_2^{-1}\right). \quad (5.6)$$

Linear term: Define

$$\mathbf{w}_j = \boldsymbol{\Sigma}_j^{-1}\boldsymbol{\mu}_j, \qquad j = 1, 2, \quad (5.7)$$

which yields the linear coefficient vector

$$\mathbf{w} = \mathbf{w}_1 - \mathbf{w}_2 = \boldsymbol{\Sigma}_1^{-1}\boldsymbol{\mu}_1 - \boldsymbol{\Sigma}_2^{-1}\boldsymbol{\mu}_2. \quad (5.8)$$

Bias term: The constant term is

$$w_{j0} = -\frac{1}{2}\boldsymbol{\mu}_j^\top \boldsymbol{\Sigma}_j^{-1}\boldsymbol{\mu}_j - \frac{1}{2}\ln|\boldsymbol{\Sigma}_j| + \ln(\phi_j), \quad (5.9)$$

so that $b = w_{10} - w_{20}$.

If $\boldsymbol{\Sigma}_1 = \boldsymbol{\Sigma}_2$, then $\mathbf{Q} = 0$ and the quadratic term cancels. The boundary reduces to a linear hyperplane, similar to Chapter 3 (shared-covariance) case.

5.3.4 Non-splitting condition for GQDT-C

We construct GQDT-C using GMM. When presented with a set of data, we implement GMM on the data to construct two clusters. Then, we decide whether to split the sub-regions any further by analyzing the purity of the sub-region. If the purity of the sub-region is greater than a threshold value, we convert that sub-region into a leaf node, if not, we iterate further. In our case, we use the threshold δ to decide whether to split a node further or to convert it into a leaf node. Each single GQDT and GQDT in the ensemble uses the same non-splitting criteria as GODT defined in Chapter 3 for classification and regression (Section 3.3.5).

5.3.5 Initialization

We use GMM clustering to create sub-regions of the dataset we are presented with. One major consideration when using GMM or any type of clustering is the cluster initialization. In our case, we need to consider the cluster centroid initialization. The performance of a clustering technique that converges to a local maximum of the likelihood function depends highly on the initial cluster centroids. There are a few disadvantages of not initializing the cluster centroids, including reasons such as creation of empty clusters, slower convergence, and a higher chance of getting stuck in a bad local minima (Celebi et al., 2013). Unlike techniques like the K-means clustering, which converges to a local minimum, techniques like GMM converges to a local maximum due to the EM algorithm. Moreover, Panić et al. (2020) demonstrated that careful initialization when using the EM algorithm is necessary, even when applied in highly favorable settings, and performing random initialization would converge to bad critical points (Jin et al., 2016).

We compare our GQDT-C technique with the GODT-C technique and the CART technique. Firstly, the standard CART technique produces consistent results at every run. This is given that we use the same sample of data. On the other hand, as the GQDT-C and GODT-C techniques are GMM dependent, where the GMM is dependent on the EM algorithm. At every run, the cluster assignment varies due to random initialization of the cluster centroids. In Chapter 3 Algorithm 6, we described how the GODT-C technique's cluster centroids are initialized. The initialization of the GQDT-C technique's cluster centroids are similar.

We initialize the cluster centroids of the GMM technique using the K-means technique, where the clusters centroids of the K-means technique are initialized using Algorithm 6. We perform an experiment on a small number of datasets to observe how initializing using algorithm 6 against random initialization impacts the performance and computational times of the GQDT technique (Appendix A.6, Table A.14). We observe that initializing the cluster centroids using Algorithm 6 generally

improved the performance with very similar computational times. However, in the high-dimensional example, random initialization added extra computational time with decrease in accuracy. So, we choose to initialize the GMM cluster centroids using Algorithm 6 .

In our technique, we have three separate parameters for the GMM, i.e., the mean, covariance, and the prior mixture components that require initialization as part of the implementation. Firstly, the means $\boldsymbol{\mu}_1$ and $\boldsymbol{\mu}_2$ are initialized using K-means with a further step of K-means centroids initialized using algorithm 6 (please refer Chapter 3, section 3.3.6). We should also note that as part of algorithm 6, the data is centered at every non-terminal node. Secondly, for the full covariance $\boldsymbol{\Sigma}_1$ and $\boldsymbol{\Sigma}_2$, the initial covariance matrices are set to the empirical covariance of the data, capturing both variances and feature correlations. Finally, the prior mixture components ϕ are set to uniform for initialization, i.e., if we have $M = 2$ in our case, each component receives an initial equal weight of $\frac{1}{2}$.

As discussed earlier in this chapter, we take our GQDT-C technique a step further to implement the ensemble of this technique. We specifically decide to create an ensemble using bootstrap aggregation and feature sub-sampling. We discuss this in the next section.

5.3.6 Ensemble of GQDT-C

In the previous section, we proposed the GQDT-C technique that uses GMM to construct quadratic decision boundaries at each non-terminal node to build a top-down decision tree. We observe in the experiments section of 5.4 that this performs competitively as a standalone learner. However, as observed in the literature review, ensemble techniques are recognized to reduce variance, improve robustness, improve generalization, mainly on complex datasets. To take advantage of these benefits, we take our GQDT-C technique forward to build an ensemble of this technique called Gaussian Quadratic Random Forest for Classification (GQRF-C). The GQRF-C is based on two key ensemble strategies, namely bootstrap aggregation (bagging) and feature sub-sampling. This ensemble strategy is based on the standard random forest technique by Breiman (2001). We discuss bagging in detail in Chapter 2 and Section 2.5.

Algorithm 11 GQRF with Bagging and Feature Sub-sampling

Input: Training dataset $\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$
Number of trees $n_estimators$
Bootstrap flag $bootstrap \in \{\text{True}, \text{False}\}$
Number of samples per tree $max_samples$
Number of features per tree $max_features$

Output: Ensemble model $E = \{T_1, T_2, \ldots, T_{n_estimators}\}$

1: Initialize empty ensemble: $E \leftarrow [\,]$
2: **for** $i = 1$ to $n_estimators$ **do**
3: **if** $bootstrap = \text{True}$ **then**
4: Sample $max_samples$ instances from $\mathcal{D}$ *with replacement* to form $\mathcal{D}_i$
5: **else**
6: Sample $max_samples$ instances from $\mathcal{D}$ *without replacement* to form $\mathcal{D}_i$
7: **end if**
8: Randomly select $max_features$ features from the p available features to form subset F_i
9: Restrict all instances in $\mathcal{D}_i$ to the features in F_i
10: Train decision tree: $T_i \leftarrow \text{GQDT}(\mathcal{D}_i)$
11: Add T_i to ensemble: $E \leftarrow E \cup \{T_i\}$
12: **end for**
13: **return** E

In addition to bagging, feature sub-sampling is another way to induce randomness into the ensemble process. We call this the *mtry* parameter. The value of *mtry* is used to inject randomness in the tree-building process. In the standard random forest by Breiman (2001), a different subset of *mtry* is selected when splitting each non-terminal node. In a standard random forest, a general rule of thumb is to tune the *mtry* hyperparameter and have the *mtry* hyperparameter value approximately set to $\sqrt{p}$ for classification. These are proposed starting values that could be tuned. However, Fernández-Delgado et al. (2014); Zhang and Suganthan (2017) show that tuning the *mtry* parameter generally produces better performing models. In our case, the GQRF-C employs quadratic splits, so each candidate decision boundary can combine variables through both squared and cross-product terms. Because the class-separating information often lies in pairs of features acting together, sampling only $\sqrt{p}$ features at each node sometimes excludes one member of a critical pair and degrades accuracy. When tuning our model, the default $\sqrt{p}$ setting for *mtry* gave weaker models, whereas increasing *mtry*, and in some cases letting every node consider the full feature set consistently improved the performance. As a result, we tune *mtry* carefully in the quadratic random-forest implementation, and we are prepared to use the full feature set in some cases when the data requires it. So, in order to induce other forms of randomness, for datasets where we increase the *mtry* value close to the full feature

set, we randomly initialize the GMM cluster centroids and we do not use the technique mentioned in algorithm 6. This is so that we can induce further randomness into the ensemble.

5.3.7 Hyperparameters for GQDT-C and GQRF-C

When constructing the GQDT-C technique, we will have to consider various hyperparameters to tune the model. We construct both a GQDT-C and then an ensemble of GQDT-C using bootstrap aggregation and feature sub-sampling. We will first consider the different features that require tuning when constructing the GQDT-C and then we look at the features for the ensemble of GQDT-C, i.e., GQRF-C.

When constructing the GQDT-C, at the tree-level, we have four main hyperparameters, which includes the δ parameter to measure the purity of the node/split/sub-region during the classification task. So, when we implement GMM on a subset of data, we use this purity measure to decide whether we split a node further or not. The β parameter that sets the regularization for the GMM, the M value which looks at the number of components when implementing the GMM, and $min_samples_leaf$ looks at the minimum number of samples required to be at the leaf node. Please refer Chapter 3 for more details on the δ and β parameters. There are more parameters that we can tune when building a tree-based technique, but these are the main parameters we consider. Let us look at each one of these:

- Firstly, we look at the M hyperparameter, which looks at the number of components for the GMM. For the GQDT-C technique, we are looking to construct a binary tree technique. As we aim for a binary split, we work with the approximation of having only two components for the GMM. This would help us split a region into two, using a single decision boundary at each step. So, $M = 2$. We are fully aware that there may be more components, but these will be explored in future work.
- Secondly, δ is the purity hyperparameter that takes the value between $(0, 1]$ and requires tuning. As $\delta \to 1$, this would mean that each non-terminal node is fully pure and data in the node belongs to only a single class. This is a parameter that we tune on the validation dataset. The aim of the purity threshold δ is to prevent unnecessary splitting when the node already has a high purity.
- The β is the regularization hyperparameter crucial for a GMM to ensure that the GMM is stable and well-defined. The β value is added to the diagonal of the covariance matrices to make sure that they remain positive definite and numerically stable. This helps prevent variances from becoming too small, which could otherwise make the matrix nearly singular. This is a parameter that

we tune typically by testing small positive values to make sure that the covariance matrices stay positive definite without overly distorting the original estimated variance.

- The $min_samples_leaf$ considers the minimum number of samples required to be at the leaf node. When constructing a decision tree, this is a parameter that we tune to make sure that the tree is not allowed to grow very deep to avoid overfitting. The idea behind a random forest is to make sure that each individual decision tree is deep, which would make the trees have low bias and high variance. Then, when the random forest aggregates the predictions from the multiple deep decision trees, this would reduce the variance leading to an ensemble model with low bias and lower variance. Breiman (2001); Friedman et al. (2001) discuss the same idea. So, when constructing GQDT-C as part of GQRF-C, we make sure to construct deep decision trees. However, if we are constructing a single GQDT, we tune this hyperparameter to avoid overfitting.

There are further hyperparameters to consider when we construct a GQRF-C, i.e., a random forest where the baseline decision trees are GQDT-C. Let us now consider the forest-level hyperparameters that require tuning when constructing a GQRF-C. These are as follows:

- A random forest is made up of multiple decision trees. And, this is a parameter that we have to control. The $n_estimators$ controls the number of trees that we build as part of the forest. The general consideration is the constructing more trees would generally improve the performance of the random forest model. However, constructing a large number of trees quickly adds to the computational cost. Beyond a certain number, constructing more trees would not significantly add to the accuracy. So, we try to find a balance in the trade-off between accuracy and computational time, when tuning this hyperparameter.
- As part of the random forest technique, when constructing the underlying decision tree techniques, each tree is constructed on a bootstrap sample of the dataset with replacement. The $bootstrap$ hyperparameter suggests whether to sample the dataset with replacement or to use the full dataset.
- If we consider to bootstrapping the sample, we will have to decide on the $max_samples$ when constructing each tree. We use randomly selected 80% of the training dataset with replacement, so this parameter is set to 0.8 for all the datasets. This will induce randomness in the ensemble (Breiman, 2001).
- Finally, we look at the $max_features$, i.e., the $mtry$ hyperparameter, which looks at the random number of features to consider at every split. As discussed earlier in this chapter, this is a hyperparameter that we tune.

5.4 Experimental Results

In this section, we look at the performance of the Gaussian Quadratic Decision Tree technique for classification (GQDT-C) and the performance of Gaussian Quadratic Random Forest technique for classification (GQRF-C). Before looking at the performance, we look at the datasets used, experimental setup, and then the final results comparison of the techniques. We compare GQRF-C and GQDT-C with GORF-C, GODT-C (from chapter 3 and 4), standard random forest for classification technique (RF-C), and CART. This will give us a good idea of how each tree-based technique is performing.

5.4.1 Machine Learning Datasets

In order to test the performance of GQRF-C and GQDT-C, we implement the technique on 22 benchmark datasets that have mainly come from the UCI repository (Dua and Graff, 2017), with some datasets coming from other websites (Chang and Lin). The majority of datasets that we use here have already been evaluated in previous studies in the area of machine learning. In the 22 datasets that we work with in our experiments, it contains a mix of multi-class and binary datasets. We work with 14 multi-class datasets and 8 binary datasets. Furthermore, we work with a mix of high-dimensional, large-scale, and normal-sized datasets. Please refer Table 3.1 for more information on the dataset. This provides us with a good variety of datasets to test the versatility of the techniques. We use the same datasets as previous chapters to provide us with a consistent comparison across techniques, as all the techniques that we propose are non-axis-parallel tree-based techniques and their respective ensembles. The different techniques that we use have hyperparameters that require tuning. So, we would require a 75:25 train-test split, where a random subset of 20% of the training data is used for validation. We use test accuracy as our performance measure. The test accuracy looks at the total number of correct predictions on the test data out of the total number of observations in the test data, as a percentage.

5.4.1.1 Experimental Setup

In this section, we consider the different techniques that we use to compare with GQRF-C and GQDT-C and the configuration of the system we implement this in. We will compare GQRF-C and GQDT-C with tree-based techniques, mainly, GORF-C, GODT-C, RF-C, and CART. We implement these techniques in MATLAB 2019b on a computer which is Intel(R) Core(TM) i5-7500 CPU @ 3.40GHz 3.40 GHz with 8.00 GB RAM. The GODT-C and GORF-C are oblique decision tree and oblique random forest techniques that we proposed in chapter 3 and chapter 4 respectively. The RF-C and

CART are axis-parallel random forest and decision tree techniques respectively. For the RF-C technique, we use the `fitcensemble()` to construct an ensemble of CART decision trees and the function `predict()` to test the model performance. For the CART technique, we utilize the `fitctree()` to learn a single CART decision tree and the function `predict()` to test the model performance. For GQRF-C, GQDT-C, GORF-C, and GODT-C, we wrote and implemented the code from scratch in MATLAB.

When we train all the techniques mentioned above, they require hyperparameters to be tuned. The hyperparameters that we tune depend on the type of technique that we are working with. For the GMM-based techniques that we propose, we use the validation dataset to tune the purity parameter δ and the regularization parameter β. Along with these, we have parameters like minimum leaf size, maximum depth, and in the case of the ensemble techniques, the number of trees, number of features to subsample, proportion of sample to bootstrap and etc. Furthermore, in addition to the accuracies, we also record the training times of the different algorithms. For these, we run the algorithm 10 times and record the average time taken to train on the dataset. The hyperparameter tuning strategy and range is similar to chapter 3 and 4 for GQDT, GQRF and the baseline techniques. Please refer section 3.4.1.3 and 4.4.1.2 for more information on the hyperparameter tuning strategy.

Dataset	**GQDT-C**	**GODT-C**	**CART**
	High-dimensional dataset		
Gisette	92.30	91.80	**93.50**
Madelon	57.33	**76.00**	74.83
Usps	**95.38**	91.86	87.15
	Large-scale dataset		
Acoustic	**74.04**	70.55	68.20
Combined	78.56	76.31	**78.70**
Poker	54.67	**56.68**	52.64
	Normal-sized dataset		
Abalone	55.07	**55.15**	51.16
Adult	**76.38**	**76.38**	74.39
Breast Cancer	**97.81**	**97.81**	95.62
Dermatology	81.52	**95.65**	91.30
Drug Consumption	77.71	**78.13**	73.89
Ecoli	79.76	**85.71**	78.57
Glass	**69.77**	67.44	55.81
Heart Failure	78.33	**85.00**	75.00
Ionosphere	**96.19**	84.76	85.71
Iris	**100.00**	**100.00**	**100.00**
Letter	**88.92**	82.12	78.40
Movementlibras	**81.11**	76.67	66.67
Mushroom	**100.00**	**100.00**	99.92
Satimages	**89.36**	89.12	86.01
Segmentation	**94.34**	88.68	83.02
Wine	94.44	**100.00**	94.44

TABLE 5.1: Test accuracies of the tree-based techniques, i.e., GQDT-C, GODT-C, and CART.

5.4.1.2 Classification Performance

In this section, we compare the classification performance of the different techniques. We mainly want to observe how the GQRF-C and GQDT-C perform as compared to other tree-based techniques. To remind ourselves again, the GQDT-C technique constructs non-linear decision boundaries at each non-terminal node to split the data, where this decision boundary can be in the form of an ellipsoids, or other quadrics depending on the class distribution and the covariance structure. The GQRF-C technique takes this a step further to construct an ensemble of GQDT-C technique.

We first look at the performance of the individual tree-based techniques in Table 5.1. We then compare the performance of the ensembles, along with observing how the performance of the ensemble varies in comparison to a single tree-based technique. Firstly, if we compare the single tree-based techniques, i.e., GQDT-C, GODT-C, and CART, which are techniques that construct a quadratic decision boundary, linear decision boundary, and axis-parallel decision boundaries respectively. Out of the 22 datasets, we see that GQDT-C performs better in 8 out of the 22 cases, GODT-C performs better in 9 out of 22 cases, CART performs better in 1 out of 22 cases, GQDT-C and GODT-C return the same accuracy in 3 out of the 22 cases, and then all the three techniques return the same accuracy in 1 out of 22 cases. The GQDT-C performance on the Madelon dataset is away from the best performance of 76.00 accuracy produced by GODT-C. We think this may be due to the nature of the dataset. As the dataset has all feature values that are categorical, a quadratic decision boundary may be less suitable here. In general, we see that CART is the third-best performing technique in classification, however, we observe that CART is not far away from the best performing model in 4 cases. On the datasets where GQDT-C does not perform better, it misses out on the top spot on another 4 datasets by a very small margin. Where the GQDT-C technique performs really well on 6 datasets, it is the better performer quite comfortably. So, we observe that the GQDT-C has a consistently decent classification performance across all of the datasets.

Now, let us have a look at the computational times of the tree-based techniques in Table 5.2. Looking at the normal-sized datasets, we observe that the computational times of the GQDT-C is similar to GODT-C. There were a couple of instances like the Letter dataset and Satimages where the computational times of GQDT-C was even quicker than GODT-C. These were the datasets where even GQDT-C performed better from a classification accuracy point of view. We did also come across scenarios in the high-dimensional and large-scale datasets, where the GQDT-C took longer than GODT-C, like the Madelon dataset. This was the dataset where the computational time was nearly double that of GODT-C. Looking back at the classification accuracy, this was the dataset where GQDT-C performed slightly poorly. So, this may be a dataset where a quadratic decision boundary is probably not the most suitable option,

Dataset	GQDT-C	GODT-C	CART
	High-dimensional dataset		
Gisette	310.4756	185.7200	4.7725
Madelon	17.4218	2.4563	0.2688
Usps	22.9973	5.7451	1.7287
	Large-scale dataset		
Acoustic	99.9128	19.4608	4.3866
Combined	196.9527	90.0542	8.1299
Poker	14.3056	8.8307	0.6860
	Normal-sized dataset		
Abalone	0.2546	0.5214	1.0086
Adult	0.1326	3.0967	0.7297
Breast Cancer	0.0713	0.4846	0.6409
Dermatology	0.1892	0.0508	0.0583
Drug Consumption	0.4746	0.2104	0.0494
Ecoli	0.1679	0.4881	0.6271
Glass	0.1462	0.3594	0.6261
Heart Failure	0.0882	0.5346	0.6115
Ionosphere	0.1204	0.3569	0.6476
Iris	0.1273	1.3062	0.7477
Letter	4.6927	1.0739	0.0966
Movementlibras	0.4257	0.6789	0.6660
Mushroom	0.2273	0.3424	0.2271
Satimages	1.3554	0.6664	0.1048
Segmentation	0.1703	0.7485	0.6578
Wine	0.1229	0.1110	0.6660

TABLE 5.2: Training times of the tree-based techniques, i.e., GQDT-C, GODT-C, and CART in seconds.

leading to longer times for finding a decision boundary that would lead to a pure class. Another consideration is that as both of these datasets are high-dimensional, and computing full covariance matrices for the two individual components as part of GMM might have been computationally expensive. However, apart from a few scenarios, the GQDT-C computational times are fairly quick. Additionally, it is worth noting that even though the computational times of GQDT-C are fairly quick, the computational times of CART is the quickest. This is expected assuming that CART is implementing axis-parallel decision boundaries and the library used to implement CART has been fully optimized in MATLAB.

We have to also note the versatility of the GQDT-C that we propose. We implement the GQDT-C technique on a mix of large-scale, high-dimensional, binary, multi-classed, and normal-sized dataset. We observe that in general, across all of these datasets, this technique in most cases consistently performs better or similar to the best performing model. We do also have decent computational times that are not extremely slow. This shows that we have a technique that is versatile with a reliable accuracy and computational times.

We take our GQDT-C technique a step further by creating an ensemble of GQDT-C technique using bagging and feature sub-sampling. Single tree-based techniques

Dataset	GQRF-C	GORF-C	RF-C
	High-dimensional dataset		
Gisette	92.48	**95.50**	91.80
Madelon	57.22	**79.17**	76.83
Usps	**96.45**	96.10	95.22
	Large-scale dataset		
Acoustic	76.34	72.83	**76.51**
Combined	78.49	77.90	**82.10**
Poker	56.78	56.94	**61.01**
	Normal-sized dataset		
Abalone	**57.46**	55.39	55.87
Adult	**76.38**	**76.38**	**76.38**
Breast Cancer	**97.81**	**97.81**	94.89
Dermatology	**97.83**	**97.83**	93.48
Drug Consumption	**77.49**	**77.49**	**77.49**
Ecoli	**82.14**	80.95	80.95
Glass	69.77	72.09	**76.74**
Heart Failure	78.33	**81.67**	71.67
Ionosphere	**98.10**	91.43	90.48
Iris	**100.00**	**100.00**	**100.00**
Letter	**94.44**	93.46	94.34
Movementlibras	**85.56**	82.22	73.33
Mushroom	**100.00**	**100.00**	**100.00**
Satimages	**91.50**	90.44	88.97
Segmentation	**92.45**	**92.45**	90.57
Wine	**100.00**	**100.00**	97.22

TABLE 5.3: Test accuracies of the ensemble-based techniques, GQRF-C, GORF-C, RF-C.

generally has the tendency to overfit even after tuning the hyperparameters like the minimum leaf size, maximum depth, and etc. Hence, the ensemble of each of GQDT-C, GODT-C, and CART, i.e., GQRF-C, GORF-C, and RF-C would give us a robust performance measure of the techniques on each of the datasets and a good indication of performance without overfitting. We will now compare the accuracies and computational times of the ensemble-based techniques in Tables 5.3 and 5.4 respectively.

Looking at the classification accuracies of the ensemble-based techniques, one key observation that we make is that the performance of the GQRF-C technique is either similar to or better than the GQDT-C. Moreover, we observe that there are many datasets where GQRF-C performs similar to GORF-C and GQRF-C technique has accuracies either very close to or better than the best performing model. Out of the 22 datasets, GQRF-C performs best in 7 out of the 22 datasets, GORF-C performs best in 3 out of the 22 datasets, GQRF-C and GORF-C performs similar in 8 out of the 22 datasets, RF-C performs better in 4 out of the 22 datasets and RF-C performs similar to GQRF-C and GORF-C in 4 out of the 22 datasets. On the datasets where RF-C performs better like the acoustic, combined, poker, and Glass datasets, the GQRF-C does not perform poorly. So, we have come up with a technique that has a reliably good performance across all varieties of datasets. Please refer Appendix A.5 for the

macro-averaged precision, macro-averaged recall, and macro-averaged F1-scores. Please refer Appendix A.7 for the statistical comparison.

Dataset	**GQRF-C**	**GORF-C**	**RF-C**
	High-dimensional dataset		
Gisette	2098.5847	1897.9786	131.9266
Madelon	896.1643	589.1086	10.9072
Usps	1032.5674	119.8796	27.5523
	Large-scale dataset		
Acoustic	2065.6753	678.9786	198.7093
Combined	3099.4657	1127.8796	5.7043
Poker	365.4745	346.7380	740.5240
	Normal-sized dataset		
Abalone	28.8697	32.4718	3.1569
Adult	309.1223	5.7154	6.6409
Breast Cancer	6.8460	9.3929	4.3945
Dermatology	19.0097	14.6959	2.7227
Drug Consumption	42.1312	80.0657	5.6637
Ecoli	15.6674	5.7612	1.8013
Glass	13.4278	18.8164	3.9908
Heart Failure	9.0878	9.3600	3.5376
Ionosphere	13.1085	14.3184	2.6319
Iris	8.7523	5.4850	3.0563
Letter	378.6194	98.6785	11.6182
Movementlibras	38.1632	26.0444	3.7931
Mushroom	24.5786	53.0158	5.3252
Satimages	101.2542	268.0892	5.4701
Segmentation	16.2322	46.1106	3.7387
Wine	8.7103	5.4099	2.1302

TABLE 5.4: Training times of the ensemble-based techniques, GQRF-C, GORF-C, RF-C in seconds.

If we consider the computational times of the GQRF-C technique, similar to the GORF-C, the computational times are definitely high on some datasets, but does not exceed 1 hour. It is very difficult for us to compare this against the RF-C, as the RF-C in the Statistics and Machine Learning toolbox in MATLAB relies on pre-compiled libraries written in C/C++ or FORTRAN for many of its operations. So, the techniques that we propose and implement in MATLAB have the opportunity to be further optimized. So, if we implement our code in something like C/C++, we can come up with very comparable computational times to compare against existing techniques. This assumption is based on the known performance characteristics of C and C++, which are highly optimized for speed and low-level memory management. Given that the existing pipeline was developed in MATLAB, which is an interpreted language that is generally slow for computationally intensive tasks, re-implementing the same logic in C or C++ would likely lead to significant performance gains (Moler, 2004).

However, to reiterate, existing computational times are not extremely computationally intensive assuming we developed the end-to-end pipeline in MATLAB. Re-developing the pipelines in C/C++ would be as part of the future work.

In Appendix A.11 figure A.1, we compare the training time as the training sample increases on the Madelon dataset for RF-C, GORF-C, and GQRF-C. RF-C scales efficiently with sample size, while GORF-C shows increased but controlled computational cost. GQRF-C exhibits a steep growth in training time for large sample sizes, suggesting that the additional optimization complexity significantly impacts scalability. Appendix A.11 figure A.2 shows log–log plots of training time against feature dimensionality as the feature dimensionality changes for the Madelon dataset. RF-C exhibits the most favorable scaling, with training time increasing slowly as the number of features grows. GORF-C incurs higher computational cost due to oblique split optimization, but its scaling with feature dimension remains moderate. In contrast, GQRF-C shows a sharp increase in training time for large feature sizes.

5.5 Computational Complexity of the Proposed Methods

This section presents a detailed Big-O complexity analysis of the proposed Gaussian Oblique Decision Tree (GODT), Gaussian Quadratic Decision Tree (GQDT), Gaussian Oblique Random Forest (GORF), and Gaussian Quadratic Random Forest (GQRF) methods. The structure of this section follows:

- explicit comparison with classical CART and Random Forests;
- clear complexity dependence on: N (number of samples), p (number of features), m_{try} (feature-subset size), T (number of trees), I (EM iterations);
- explicit consideration of Algorithm 6-based initialization (PCA, median split, and K-means);
- explicit treatment of the cost of constructing linear and quadratic hypersurfaces.

We assume balanced binary trees, giving a depth of $O(\log N)$, consistent with standard analyses of decision trees (Louppe, 2014). At each non-terminal node of each proposed method, a fresh Gaussian mixture model (GMM) is fitted. Thus the complexity depends crucially on the cost of initialisation, EM iterations, and hypersurface construction.

5.5.1 Classical Decision Tree Baselines

5.5.1.1 CART

At each node, CART evaluates axis-parallel splits over all p features. Evaluating a candidate split requires scanning or sorting feature values. The per-node cost is therefore:

$$O(Np).$$

With tree depth $O(\log N)$, the total training cost is:

$$\boxed{T_{\text{CART}} = O(Np \log N)}.$$

Prediction requires following a single root-to-leaf path:

$$O(\log N).$$

5.5.1.2 Classical Random Forest (RF)

A Random Forest grows T independent CART trees, each using feature sub-sampling. At each node, only m_{try} features are evaluated.

Thus:

$$\boxed{T_{\text{RF}} = O(TNm_{\text{try}} \log N)}.$$

Prediction requires:

$$O(T \log N).$$

5.5.2 Cost Components in the Proposed Methods

Every non-terminal node in GODT, GQDT, GORF and GQRF performs:

1. **Algorithm 6 based initialization** (PCA $\rightarrow$ median split $\rightarrow$ K-means),
2. **GMM fitting** (EM with $M = 2$ components),
3. **Construction of linear or quadratic hypersurfaces**.

Below we compute the cost of each component.

5.5.2.1 Algorithm 6 based Initialization

Algorithm 6 performs PCA, computes a median split, and runs K-means with $k = 2$.

The MATLAB `pca` function uses a full singular-value decomposition. This costs:

$$O(\min(Np^2, pN^2)).$$

In most machine-learning settings $N > p$, giving:

$$O(Np^2).$$

Projection along the first principal component costs $O(Np)$; the median can be computed in $O(N)$. Hence:

$$O(Np).$$

High-quality initialization ensures rapid convergence, so the cost of K-means is:

$$O(Np).$$

Since $O(Np^2)$ dominates:

$$\boxed{T_{\text{init}} = O(Np^2)},$$

which is the total initialization cost. This is the dominant cost in GODT and an important contributor in GQDT.

5.5.2.2 Gaussian Mixture Model (GMM)

Each proposed method refits a two-component GMM at every node.

GODT with diagonal, shared covariance: E-step: $O(Np)$, M-step: $O(Np)$. For I EM iterations:

$$\boxed{T_{\text{GMM,GODT}} = O(INp)}.$$

GQDT with full, separate covariances: Full covariance matrices require quadratic operations in p.

E-step: $O(Np^2)$, M-step: $O(Np^2)$.

$$\boxed{T_{\text{GMM,GQDT}} = O(INp^2)}.$$

5.5.2.3 Hypersurface Construction

GODT with linear hyperplane. Calculation of:

$$\mathbf{w} = \Sigma^{-1}(\mu_2 - \mu_1), \qquad p = \text{cut-point},$$

costs:

$$O(p).$$

GQDT with quadratic hypersurface. Assembly of:

$$\mathbf{Q} = -\tfrac{1}{2}(\Sigma_1^{-1} - \Sigma_2^{-1}), \qquad \mathbf{w}, \qquad b,$$

requires:

$$O(p^2).$$

5.5.3 Per-node and Per-tree Complexity

5.5.3.1 GODT

Per-node cost:

$$T_{\text{node}}^{\text{GODT}} = O(Np^2) + O(INp).$$

A balanced tree has depth $O(\log N)$, giving:

$$\boxed{T_{\text{GODT}} = O(Np^2 \log N) + O(INp \log N)}.$$

5.5.3.2 GQDT

Per-node cost:

$$T_{\text{node}}^{\text{GQDT}} = O(Np^2) + O(INp^2).$$

Thus:

$$\boxed{T_{\text{GQDT}} = O(Np^2 \log N) + O(INp^2 \log N)}.$$

The GMM term usually dominates for large I or high p.

5.5.4 Ensemble Methods: GORF and GQRF

Each tree uses only m_{try} randomly sampled features. We substitute m_{try} directly into the complexity expressions.

5.5.4.1 GORF

Initialization (Algorithm 6 with m_{try} features):

$$T_{\text{init}} = O(Nm_{\text{try}}^2).$$

GMM (diagonal, shared covariance):

$$T_{\text{GMM}} = O(INm_{\text{try}}).$$

Total per-tree cost:

$$\boxed{T_{\text{GORF-tree}} = O(Nm_{\text{try}}^2 \log N) + O(INm_{\text{try}} \log N)}.$$

Total forest cost (with T trees):

$$\boxed{T_{\text{GORF}} = O(TNm_{\text{try}}^2 \log N) + O(TINm_{\text{try}} \log N)}.$$

5.5.4.2 GQRF

GQRF performs Algorithm 6 and GMM using full covariance matrices, but only within the m_{try}-dimensional subspace.

Thus per-tree:

$$\boxed{T_{\text{GQRF-tree}} = O(Nm_{\text{try}}^2 \log N) + O(INm_{\text{try}}^2 \log N)}.$$

Total forest:

$$\boxed{T_{\text{GQRF}} = O(TNm_{\text{try}}^2 \log N) + O(TINm_{\text{try}}^2 \log N)}.$$

5.5.5 Comparison Table

Model	Training Complexity
CART	$O(Np \log N)$
Random Forest (RF)	$O(TNm_{\text{try}} \log N)$
GODT	$O(Np^2 \log N + INp \log N)$
GORF	$O(TNm_{\text{try}}^2 \log N + TINm_{\text{try}} \log N)$
GQDT	$O(Np^2 \log N + INp^2 \log N)$
GQRF	$O(TNm_{\text{try}}^2 \log N + TINm_{\text{try}}^2 \log N)$

TABLE 5.5: Computational complexity of classical and proposed models.

5.5.6 Interpretation of Computational Behaviour

The complexity expressions allow us to reason about how the proposed methods behave across different data regimes.

Low-dimensional datasets (small p): When p is small (e.g. $p < 20$), the quadratic and linear terms in p become inexpensive. In this regime:

- GODT becomes close in cost to CART, often differing only by a constant factor.
- GQDT remains more expensive than GODT (due to full covariances) but is still practical.
- GORF and GQRF are both efficient because m_{try} is small.

Thus, the proposed methods are highly scalable for low-dimensional data.

Moderate-dimensional datasets (tens to hundreds of features): In this regime, PCA and GMM begin to dominate. The cost differences become clearer:

- GODT's diagonal-covariance GMM scales linearly in p inside the EM step, while PCA contributes a quadratic p^2 term.
- GQDT's full-covariance GMM introduces a quadratic p^2 factor in every iteration.
- GORF and GQRF reduce dimensional dependence from p to m_{try}, which is an advantage when $m_{\text{try}} \ll p$.

High-dimensional datasets ($p \gg 1000$): Here dimensionality dominates the complexity landscape:

- PCA's $O(Np^2)$ becomes very expensive.
- GQDT and GQRF incur $O(INp^2)$, making them the most costly.
- GODT remains cheaper than GQDT due to diagonal covariance.
- GORF and GQRF are significantly more manageable if $m_{\text{try}} \ll p$, as the quadratic term becomes m_{try}^2 instead of p^2.

Thus, feature sub-sampling is essential for scalability in high-dimensional settings.

Small sample size (small N): When N is small:

- All GMM-based methods are fast because EM iterations scale in N.
- PCA is inexpensive as long as p is not extremely large.

Hence GODT and GQDT both train quickly.

Large sample size (large N): When N is large:

- The cost of both PCA and GMM scales linearly in N at every node.
- Ensemble methods multiply this by T.
- CART and RF remain the fastest when N is very large.
- GORF offers a good compromise because $m_{\text{try}}^2 \ll p^2$.

Overall Summary

- CART and RF are the most computationally efficient across all settings.
- GODT is efficient for low to moderate dimensional data and scales better than GQDT.
- GQDT and GQRF offer increased modelling capacity but at the cost of quadratic dependence on p.
- Feature sub-sampling in GORF and GQRF dramatically reduces dimensionality burden.
- For high-dimensional or very large datasets, GORF is the best-balanced among the proposed methods.

Hence, the Gaussian Quadratic Decision Tree and Gaussian Quadratic Random Forest techniques that we proposed in this chapter are suitable techniques for both classification and regression. However, early experiments that we performed suggested that the performance of the quadratic decision tree-based technique on regression tasks was not as expected. So, this chapter was focussed on classification. However, we believe spending more time on the regression task, mainly looking at the non-split condition could help the technique perform well on regression. We also compare the computational complexities of the different tree-based techniques and their respective ensembles. In the next chapter, we conclude this thesis and discuss future work.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

In this thesis, we proposed several computationally efficient non-orthogonal tree-based algorithms that performed well on a wide variety of datasets. We proposed oblique and quadratic decision tree-based techniques along with their respective ensembles.

In Chapter 3, we proposed oblique decision tree-based algorithms to address the limitations with traditional axis-parallel/orthogonal decision trees. We observed in the literature that traditional axis-parallel decision trees have the tendency to not capture non-axis-aligned or complex relationships. This sometimes creates decision trees that are overly complicated or less interpretable, potentially leading to overfitting. Moreover, existing oblique decision tree-based techniques proposed in the literature have computationally intensive procedures to optimize the best oblique decision boundary. To address these restrictions, we have developed an innovative technique to construct oblique decision trees for both classification and regression tasks. This technique efficiently handles the inherent computational intensity involved in performing an oblique split, at the same time without sacrificing the accuracy and providing a decent accuracy.

When constructing a single decision tree as the classification or regression model, there is a high-tendency to overfit due to the lack of optimized tuning of the model. So, in Chapter 4, we extend the techniques proposed in Chapter 3 to introduce an ensemble of the oblique decision tree-based algorithms for classification and regression. These algorithms adapt the ensemble framework proposed by Breiman (2001) on the oblique decision tree technique proposed in Chapter 3. When implemented across a variety of datasets, this technique demonstrated a higher robustness, better accuracy, and generalization across different types of datasets. We compared this technique across various existing machine learning models and this demonstrated good performance.

One observation we made when performing our literature review was a limited exploration of non-linear decision boundary-based decision tree techniques. In Chapter 5, we proposed a new quadratic decision tree algorithm focussed on classification tasks. The proposed decision tree technique is a classic top-down algorithm that constructs quadratic decision boundaries at each non-terminal node. The proposed quadratic decision tree would offer us an opportunity to construct a decision tree that models non-linear relationships. We observed that the quadratic decision tree technique constructs the decision boundaries in decent computational time addressing potential computational issues. In Chapter 5, we also introduced the corresponding ensemble of the quadratic decision tree using existing ensemble framework proposed by Breiman (2001). We implement the proposed quadratic decision tree and its corresponding ensemble on a variety of datasets, and observed a good performance at decent computational times. We also compared these techniques with other tree-based techniques to evaluate the performance across different algorithms.

This thesis introduced a family of Gaussian mixture-based oblique decision tree methods designed to improve interpretability and predictive performance while avoiding computationally intensive optimization. The work brings together ideas from decision-tree learning, mixture modelling, and ensemble methods to construct a flexible and transparent modelling framework.

The main contributions of this thesis are as follows:

1. A novel Gaussian Mixture Model–based oblique decision tree (GODT) that constructs hyperplane splits without explicit optimization.
2. A unified extension of GODT for both classification and regression, including tailored stopping rules based on class purity and prediction error.
3. The Gaussian Oblique Random Forest (GORF), an ensemble method that leverages bagging and random subspace selection to reduce variance in practice while retaining interpretability.
4. A novel Gaussian Mixture Model–based quadratic decision tree (GQDT) that constructs quadratic decision boundaries without explicit optimization.
5. The Gaussian Quadratic Random Forest (GQRF), an ensemble of GQDT models that combines bagging and random subspace selection to provide flexible decision boundaries within an interpretable tree-based framework.
6. A comprehensive empirical evaluation demonstrating competitive performance of GODT, GQDT, GORF, and GQRF relative to axis-parallel decision trees, existing oblique tree-based models, and other popular decision-boundary-based

machine learning techniques on both public benchmark datasets and datasets provided by industrial partners.

Overall, the research contributions presented in this thesis represent substantial advancements in the domain of decision-tree-based methods and ensembles, extending the work of decision tree techniques and its corresponding ensembles beyond their axis-parallel counterparts. The techniques that we proposed in the area of oblique decision trees and quadratic decision trees, and its corresponding ensembles would provide valuable and practical tools for complex real-world applications.

6.2 Future Work

As we made some significant progress, there are several promising areas that are available for future exploration. Due to the lack of time, we were not able to explore these further. Some potential future areas include the following:

1. **Developing a hybrid decision tree technique that performs combination of quadratic, linear, and axis-parallel decision splits**
   One observation we made when implementing axis-parallel, oblique, or quadratic decision trees is that each of these decision tree techniques performed well on some types of datasets. This gave us the idea to explore hybrid split techniques for future creation or modification of decision trees. There is an opportunity to explore techniques that either fit orthogonal, oblique, or quadratic splits at each non-terminal node depending on the quality of each type of split. Such a technique could offer greater flexibility, with a potential to enhance the accuracy of the model and create one generic model for all types of datasets.

2. **Build a random forest of the hybrid quadratic, linear or axis-parallel decision tree technique**
   Taking the hybrid decision tree technique that we proposed above, we propose future work that involves building a random forest ensemble of the hybrid tree technique. We hope that the corresponding ensemble of the hybrid technique to potentially outperforms traditional ensemble techniques due to the flexibility of constructing a quadratic, oblique, or an axis-parallel split at each non-terminal node.

3. **Exploring rotational forest and extremely randomized tree ensemble for the above models**

In our thesis, we have explored the idea of ensemble using bagging and feature sub-sampling. Future work can also explore creating an ensemble of the decision tree techniques that we proposed and the hybrid decision tree technique we proposed above using the ensemble architecture in rotational forests and extremely randomized trees. This may potentially introduce further randomness into the ensemble which can further assist in achieving better performance.

4. **Try using measures that work with class imbalance**
   Another important area is an opportunity in the future that involves looking into the class imbalance of the datasets. Having class imbalance in the datasets is common, but looking at specialized split techniques that are designed for imbalanced datasets may further assist in improving model performance and generalization when implemented on real-world datasets.

5. **Multi-way splitting decision tree**
   The oblique decision tree and quadratic decision tree techniques that we propose in this thesis perform a binary split, i.e., every node splits into two nodes. As the techniques that we propose are based on Gaussian Mixture Model, we fit a Gaussian Mixture Model (GMM) with two components. However, there is an opportunity here to explore multi-way splitting by implementing a GMM with more than two components.

6. **Comparison with neural network models**
   An important direction for future research is the comparison of the proposed Gaussian Oblique Decision Tree with neural network-based models, including both shallow and deep architectures. Neural networks are capable of learning highly non-linear decision boundaries and are known to perform well on large-scale datasets. A systematic comparison would require careful architectural design, regularization, and hyperparameter tuning to ensure a fair evaluation, which was beyond the scope of the present thesis. Investigating how GODT and GQDT compares to, or can be integrated with, neural network models remains an interesting avenue for future work.

7. **Consistency of GODT**
   Although consistency, i.e., convergence of the learned predictor to the Bayes optimal classifier as the sample size tends to infinity, has been established for several classical decision tree models, proving consistency for GODT is challenging. The GODT construction relies on an EM-based procedure whose convergence is not guaranteed in general, preventing a direct application of classical consistency proofs that assume stable or deterministic tree partitioning. Consequently, establishing consistency of GODT requires new theoretical tools and is left as an important topic for future work.

Appendix A

A.1 Additional Results for GODT Classification

We present in Table A.1, the accuracy of the GODT-C technique when we use full covariance (Full Cov accuracy), the respective computational times (Full Cov Comp Time), accuracy of the GODT-C technique when we use diagonal covariance (Diag Cov accuracy), and the respective computational times (Diag Cov Comp Time).

Dataset	Full Cov accuracy	Diag Cov accuracy	Full Cov Comp Time	Diag Cov Comp Time
Abalone	53.95%	55.15%	1.2973	0.3107
Adult	76.38%	76.37%	0.4028	0.0950
Madelon	63.17%	76.00%	21.0409	7.0117
Satimages	65.89%	89.12%	11.0512	5.2859
Wine	66.67	100.00%	0.183260575	0.1133

TABLE A.1: Table comparing the performance of GODT-C technique on some datasets when using full covariance compared to diagonal covariance when implementing the GMM. We record the test accuracies and computational times after running the same model for 10 times. The computational times is recorded in seconds

We present in Table A.2 and A.3, the accuracy of the GODT-C technique when we initialize the cluster centroids using Algorithm 6, the respective computational times, accuracy of the GODT-C technique when we randomly initialize the cluster centroids, and the respective computational times.

Dataset	Accuracy (Initialized)	Accuracy (Random)
Abalone	55.15%	54.21%±0.49%
Adult	76.37%	76.37±1.17e-14
Madelon	76.00%	66.55±1.82
Satimages	89.12%	88.13±1.42
Wine	100.00%	99.44±1.76

TABLE A.2: Test accuracy of the GODT-C technique with initialization using Algorithm 6 vs random initialization

Dataset	Initialized Times	Random Times
Abalone	0.3107±0.03403	0.2684±0.0532
Adult	0.0950±0.0437	0.1356±0.0865
Madelon	7.0117±0.1344	8.3354±1.5254
Satimages	5.2859±0.4821	4.5041±0.8483
Wine	0.1133±0.1487	0.0188±0.0028

TABLE A.3: Computational Times of the GODT-C technique with initialization using Algorithm 6 vs random initialization

A.2 Further Results for GODT-R

	GODT-R	CART	SVM
		Large-scale dataset	
mcev10	13.17214	13.9506	**12.15083**
mcev100	1.892569	1.580603	**1.334719**
mcev1000	0.811914	0.561925	**0.207444**
bejoingpm	$\mathbf{5.62 \times 10e^3}$	$5.71{\times}10e^3$	$7.56{\times}10e^3$
		Normal-sized dataset	
electrical grid	$1.04{\times}10e^{-3}$	$9.30{\times}10e^{-4}$	$\mathbf{8.35 \times 10^{-4}}$
combined cycle	20.10043	**16.59275**	$1.49{\times}10e^2$
air quality	0.04792	**0.02077**	$1.12{\times}10e^3$
bias correction	**1.7713**	2.2429	9.5561
gas turbine	**1.6855**	2.3274	229.8265
parkinsons	0.002589	**0.001978**	0.005166
airfoil self noise	34.33897	**25.53171**	45.75893
concrete	$1.53{\times}10e^2$	**97.94772**	$2.39{\times}10e^2$
can combustion	**0.025272**	0.034229	0.076521
energy efficiency	4.647594	**3.169584**	$8.32{\times}10e^{-4}$
istanbul stock exchange	$\mathbf{4.90 \times 10e^{-5}}$	$\mathbf{4.90 \times 10e^{-5}}$	$6.97{\times}10e^{-5}$
forest fires	$3.13{\times}10e^3$	$3.00{\times}10e^3$	$\mathbf{2.98}{\times}10e^3$
3d aeroplane	**2.172796**	4.219792	1.987157
auto mpg	**13.5363**	16.4997	32.16147
computer hardware	$\mathbf{9.37 \times 10e^2}$	$9.43{\times}10e^3$	$6.54{\times}10e^3$
ale	**0.0624**	0.0806	0.1958

TABLE A.4: This table presents the Mean Squared Error (MSE) for each technique on the regression dataset

A.3 Macro-averaged Precision, Macro-averaged Recall, and Macro-averaged F1 Scores for GODT-C

Datasets	GODT-C	CART	SVM	LDA	OC1
	High-dimensional dataset				
Gisette	91.80%	91.55%	96.86%	89.08%	**
Madelon	76.14%	75.98%	57.33%	58.67%	68.43%
Usps	91.57%	86.16%	91.96%	82.29%	89.11%
	Large-scale dataset				
Acoustic	69.40%	66.71%	66.31%	59.18%	*
Combined	74.63%	76.95%	79.04%	67.07%	*
Poker	15.26%	10.33%	5.01%	5.01%	13.10%
	Normal-sized dataset				
Abalone	53.68%	51.73%	37.58%	52.20%	50.30%
Adult	38.19%	49.14%	43.01%	38.19%	62.46%
Breast Cancer	96.51%	94.19%	94.44%	96.51%	91.56%
Dermatology	93.75%	88.76%	93.45%	89.88%	73.35%
Drug Consumption	32.65%	20.35%	15.36%	21.19%	20.59%
Ecoli	77.48%	67.99%	76.94%	73.50%	62.93%
Glass	68.38%	49.36%	40.74%	29.47%	66.75%
Heart Failure	80.64%	72.90%	54.23%	67.16%	72.80%
Ionosphere	86.20%	84.66%	86.90%	81.96%	92.46%
Iris	100.00%	100.00%	100.00%	100.00%	100.00%
Letter	82.54%	78.97%	84.62%	60.00%	***
Movementlibras	80.32%	69.37%	73.75%	59.10%	2.83%
Mushroom	100.00%	99.92%	97.23%	88.05%	99.55%
Satimages	87.84%	83.49%	85.03%	79.16%	81.36%
Segmentation	87.84%	83.10%	88.06%	77.47%	89.88%
Wine	100.00%	94.87%	92.86%	100.00%	90.48%

TABLE A.5: Macro-averaged Precision of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code.

Datasets	GODT-C	CART	SVM	LDA	OC1
	High-dimensional dataset				
Gisette	91.80%	91.50%	96.80%	88.80%	**
Madelon	76.00%	74.83%	57.33%	58.67%	65.14%
Usps	90.54%	85.75%	91.06%	81.76%	88.67%
	Large-scale dataset				
Acoustic	69.79%	66.72%	65.52%	60.20%	*
Combined	76.68%	78.41%	80.53%	68.00%	*
Poker	12.20%	11.17%	10.00%	10.00%	12.79%
	Normal-sized dataset				
Abalone	55.06%	51.10%	52.38%	53.90%	48.68%
Adult	50.00%	49.84%	48.98%	50.00%	66.76%
Breast Cancer	98.45%	95.44%	97.42%	98.45%	90.25%
Dermatology	93.80%	84.72%	92.78%	87.78%	74.54%
Drug Consumption	16.53%	19.51%	22.28%	24.98%	20.30%
Ecoli	74.59%	61.48%	69.82%	79.43%	58.02%
Glass	61.46%	59.58%	51.11%	33.96%	69.05%
Heart Failure	81.82%	78.98%	55.40%	70.74%	77.83%
Ionosphere	80.66%	75.93%	83.86%	80.30%	92.96%
Iris	100.00%	100.00%	100.00%	100.00%	100.00%
Letter	82.13%	78.50%	84.20%	59.70%	***
Movementlibras	79.36%	69.50%	73.47%	57.67%	18.33%
Mushroom	100.00%	99.92%	97.20%	87.33%	99.55%
Satimages	87.02%	83.99%	83.85%	79.36%	80.41%
Segmentation	85.92%	82.86%	88.16%	77.76%	91.02%
Wine	100.00%	93.33%	91.11%	100.00%	88.89%

TABLE A.6: Macro-averaged Recall of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code.

Datasets	GODT-C	CART	SVM	LDA	OC1
	High-dimensional dataset				
Gisette	91.80%	91.50%	96.80%	88.78%	**
Madelon	75.97%	74.55%	57.33%	58.67%	63.49%
Usps	90.94%	85.93%	91.98%	81.61%	87.14%
	Large-scale dataset				
Acoustic	69.39%	66.71%	65.48%	57.62%	*
Combined	75.38%	77.57%	79.67%	65.94%	*
Poker	11.84%	10.66%	6.68%	6.68%	12.91%
	Normal-sized dataset				
Abalone	52.05%	51.02%	43.00%	52.69%	49.18%
Adult	43.30%	45.46%	43.61%	43.30%	61.95%
Breast Cancer	97.41%	94.78%	95.74%	97.41%	90.87%
Dermatology	93.60%	85.67%	92.80%	88.20%	72.12%
Drug Consumption	16.48%	19.67%	9.96%	21.34%	20.31%
Ecoli	75.28%	62.32%	71.12%	74.67%	58.03%
Glass	61.97%	49.77%	44.44%	31.50%	67.57%
Heart Failure	81.19%	72.82%	52.00%	67.71%	73.53%
Ionosphere	82.32%	75.96%	85.00%	80.98%	92.70%
Iris	100.00%	100.00%	100.00%	100.00%	100.00%
Letter	82.20%	78.58%	84.27%	58.65%	***
Movementlibras	77.69%	65.67%	71.33%	57.03%	4.76%
Mushroom	100.00%	99.92%	97.21%	87.33%	99.55%
Satimages	87.38%	83.70%	84.28%	78.77%	80.50%
Segmentation	85.83%	81.58%	86.44%	76.80%	90.01%
Wine	100.00%	93.52%	91.15%	100.00%	88.36%

TABLE A.7: Macro-averaged F1-Score of the models. The * means that no result were returned for the OC1 technique or the results were not returned after running for more than an hour. The ** means that no results were returned after the technique returned an error saying no splits could be found. The *** means that there was another error returned saying left count and right count not correctly set, which is an error in the code.

A.4 GORF-C Initialization Comparisons

Dataset	Accuracy (Initialized)	Accuracy (Random)
Abalone	54.66%	54.75%
Adult	76.38%	76.37%
Madelon	79.00%	69.33%
Satimages	90.44%	90.36%
Wine	100.00%	99.44%

TABLE A.8: Test accuracy of the GORF-C technique with initialization using Algorithm 6 vs random initialization

Dataset	Initialized Times	Random Times
Abalone	42.81±1.57	39.26±1.34
Adult	2.22±0.13	48.68±1.98
Madelon	255.08±2.48	557.53±4.68
Satimages	268.08±3.67	200.87±3.12
Wine	5.41±0.13	7.53±1.37

TABLE A.9: Computational Times of the GORF-C technique with initialization using Algorithm 6 vs random initialization

A.5 Macro-averaged Precision, Macro-averaged Recall, and Macro-averaged F1-Scores for GORF-C, GQDT-C, GQRF-C

Dataset	GQRF-C	GQDT-C	GORF-C	RF-C
	High-dimensional dataset			
Gisette	91.65%	91.67%	95.54%	91.80%
Madelon	57.36%	57.77%	79.10%	63.34%
Usps	96.13%	95.16%	95.91%	94.40%
	Large-scale dataset			
Acoustic	76.47%	73.54%	72.34%	76.40%
Combined	76.11%	76.45%	77.46%	82.25%
Poker	11.28%	10.66%	19.03%	25.05%
	Normal-sized dataset			
Abalone	54.76%	55.45%	35.33%	55.87%
Adult	38.19%	38.19%	38.19%	46.52%
Breast Cancer	96.51%	96.51%	95.90%	93.56%
Dermatology	96.48%	79.56%	98.48%	93.94%
Drug Consumption	11.07%	11.07%	11.07%	11.07%
Ecoli	73.20%	71.47%	67.59%	84.90%
Glass	44.44%	55.65%	54.07%	58.78%
Heart Failure	79.09%	79.09%	82.00%	68.29%
Ionosphere	98.55%	95.48%	91.96%	95.27%
Iris	100.00%	100.00%	100.00%	100.00%
Letter	94.11%	89.37%	93.13%	93.14%
Movementlibras	85.19%	84.24%	84.36%	72.64%
Mushroom	100.00%	100.00%	100.00%	100.00%
Satimages	91.35%	88.35%	90.95%	88.57%
Segmentation	92.01%	93.99%	92.86%	89.23%
Wine	100.00%	95.00%	100.00%	97.22%

TABLE A.10: Macro-averaged Precision of the techniques

Dataset	GQRF-C	GQDT-C	GORF-C	RF-C
	High-dimensional dataset			
Gisette	90.89%	90.75%	95.50%	91.80%
Madelon	57.33%	57.67%	79.33%	63.33%
Usps	93.45%	94.65%	95.33%	94.05%
	Large-scale dataset			
Acoustic	76.34%	73.87%	72.76%	76.59%
Combined	77.89%	78.34%	78.54%	83.90%
Poker	12.67%	11.48%	11.28%	13.19%
	Normal-sized dataset			
Abalone	55.48%	56.65%	52.98%	56.27%
Adult	50.00%	50.00%	50.00%	43.32%
Breast Cancer	98.45%	98.45%	97.20%	94.19%
Dermatology	96.48%	76.94%	98.15%	89.26%
Drug Consumption	14.29%	14.29%	14.29%	14.29%
Ecoli	68.74%	67.25%	62.79%	72.53%
Glass	55.42%	53.45%	55.42%	76.87%
Heart Failure	61.36%	61.36%	72.73%	72.73%
Ionosphere	97.37%	96.45%	87.40%	90.79%
Iris	100.00%	100.00%	100.00%	100.00%
Letter	92.88%	88.96%	92.91%	93.00%
Movementlibras	85.38%	83.70%	83.93%	70.17%
Mushroom	100.00%	100.00%	100.00%	100.00%
Satimages	90.02%	86.99%	89.85%	85.52%
Segmentation	93.06%	95.10%	93.06%	90.46%
Wine	100.00%	94.44%	100.00%	96.67%

TABLE A.11: Macro-averaged Recall of the techniques

Dataset	GQRF-C	GQDT-C	GORF-C	RF-C
	High-dimensional dataset			
Gisette	91.35%	91.82%	95.50%	91.80%
Madelon	57.98%	57.34%	79.33%	63.33%
Usps	93.78%	94.87%	95.59%	94.17%
	Large-scale dataset			
Acoustic	76.12%	73.12%	72.14%	76.36%
Combined	75.24%	77.65%	77.98%	82.95%
Poker	11.65%	10.93%	10.47%	12.57%
	Normal-sized dataset			
Abalone	52.74%	55.45%	42.37%	55.35%
Adult	43.30%	43.30%	43.30%	49.99%
Breast Cancer	97.41%	97.41%	96.52%	93.87%
Dermatology	96.48%	76.64%	98.23%	90.48%
Drug Consumption	12.47%	12.47%	12.47%	12.47%
Ecoli	69.53%	68.58%	64.40%	76.41%
Glass	46.63%	55.71%	54.27%	64.39%
Heart Failure	62.48%	62.48%	75.45%	68.51%
Ionosphere	97.91%	95.92%	90.29%	92.45%
Iris	100.00%	100.00%	100.00%	100.00%
Letter	93.14%	89.05%	92.93%	93.00%
Movementlibras	84.23%	82.69%	82.89%	74.07%
Mushroom	100.00%	100.00%	100.00%	100.00%
Satimages	90.45%	87.46%	90.30%	86.47%
Segmentation	92.22%	94.25%	92.06%	89.50%
Wine	100.00%	94.57%	100.00%	96.80%

TABLE A.12: Macro-averaged F1 Score of the techniques

A.6 Further Results for GQDT-C and GQRF-C

Dataset	Full Cov accuracy	Diag Cov accuracy	Full Cov Comp Time	Diag Cov Comp Time
Abalone	55.07%	54.43%	1.1303±0.7564	0.3255±0.0867
Adult	76.38%	76.38%	7.3118±0.2630	5.7113±0.1134
Madelon	57.38%	68.67%	15.8317±0.7222	7.2157±0.5744
Satimages	89.36%	87.49%	7.6720±0.8612	7.2692±0.7283
Wine	94.44%	91.67%	0.1204±0.0032	0.0949±0.0054

TABLE A.13: Table comparing the performance of GQDT-C technique on some datasets when using full covariance (Full Cov) compared to diagonal covariance (Diag Cov) when implementing the GMM. We record the test accuracies and computational times (Comp Time) after running the same model for 10 times. The computational times are recorded in seconds

Dataset	Initialized Accuracy	Random Accuracy	Initialized Comp Time	Random Comp Time
Abalone	55.07%	52.91%±0.7653%	1.1303±0.7564	0.5906±0.0354
Adult	76.38%	76.40%±0.0004%	7.3118±0.2630	6.8905±0.5792
Madelon	57.38%	53.53%±0.0328%	15.8317±0.7222	28.8311±1.3590
Satimages	89.36%	85.31%±0.0103%	7.6720±0.8612	14.5317±0.1147
Wine	94.44%	88.89%±0.0340%	0.1204±0.0032	0.1673±0.0486

TABLE A.14: Table comparing the performance of GQDT-C technique on some datasets with random initialization of the cluster centroids vs initialization using algorithm 6. We record the test accuracies and computational times (Comp Time) after running the same model for 10 times. The computational times are recorded in seconds

A.7 Statistical Comparison of the GQDT-C and the GQRF-C techniques

Group	Compared Model	Rank	WSR p
Tree-Based	GODT-C vs GQDT-C	1.66	$8.8129 \times 10e^{-1}$
Tree-Based	CART vs GQDT-C	2.59	$1.4269 \times 10e^{-2}$
Ensemble-Based	GORF-C vs GQRF-C	1.93	$4.4451 \times 10e^{-1}$
Ensemble-Based	RFC vs GQRF-C	2.32	$1.4540 \times 10e^{-1}$

TABLE A.15: Wilcoxon Signed-Rank test results comparing GQDT-C and GQRF-C to other classifiers in their respective groups

Group	Method	Avg. Friedman Rank	Mean Accuracy (%)
Tree-Based	GQDT-C	1.75	82.41
Tree-Based	GODT-C	1.66	82.99
Tree-Based	CART	2.59	79.32
Ensemble-Based	GQRF-C	1.75	84.41
Ensemble-Based	GORF-C	1.93	84.91
Ensemble-Based	RFC	2.32	83.29

TABLE A.16: Average Friedman ranks and mean classification accuracies for tree-based and ensemble-based classifiers

Similar to Chapter 3 and Chapter 4, we evaluate the performance of our GQDT-C and GQRF-C using the Wilcoxon signed-rank tests. The results are summarized in Tables A.15 and A.16. The hypothesis for the Wilcoxon signed-rank tests are as follows:

- **Null Hypothesis (H_0)**: There is no statistically significant difference in accuracy between the reference model (GQDT-C which is the tree-based model and GQRF-C which is the ensemble based model) and each of the other classifiers in its group.
- **Alternative Hypothesis (H_1)**: The reference model performs significantly differently from the other classifiers in its group.

Tree-Based Models. GODT-C achieved the highest average Friedman rank (1.66) and mean accuracy (82.99%), GQDT-C followed closely with a rank of 1.75 and a comparable mean accuracy of 82.41%. The Wilcoxon signed-rank test between GODT-C and GQDT-C yielded a non-significant p-value ($p = 0.881$), suggesting that the performance difference between them is not statistically meaningful. Moreover, GQDT-C significantly outperformed CART ($p = 0.014$), reinforcing its strong relative performance. These findings suggest that although GQDT-C did not achieve the top rank, its performance is statistically comparable to the best-performing tree-based model.

Ensemble-Based Models. For the ensemble-based group, GORF-C recorded the highest mean accuracy (84.91%), GQRF-C obtained the best average Friedman rank

(1.75), indicating consistent performance across datasets. However, the Wilcoxon tests comparing GQRF-C to both GORF-C and RF-C did not yield significant results, suggesting no strong statistical evidence to favor one model over the others in this group. Therefore, although GQRF-C appears reliable with good accuracy, its performance cannot be considered statistically superior within the ensemble category. Future work could explore whether GQRF-C performance remains consistent across a wider range of datasets, tasks, and evaluation metrics, as well as investigating fine-tuning the model.

A.8 Pseudocode for Extremely Randomized Trees and Rotation Forests

Algorithm 12 Extremely Randomized Trees (Geurts et al., 2006)

Input: Training dataset $\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$
Number of trees B
Number of random splits per node K
Stopping criterion
Output: Ensemble model $E = \{T_1, T_2, \ldots, T_{n_estimators}\}$

```
1:  for b = 1 to B do
2:      T_b ← GROWTREE(D)
3:  end for
4:  return {T_b}_{b=1}^B
5:  function GROWTREE(S)
6:      if STOPPINGCRITERION(S) then
7:          return LeafNode(predict = majority_label(S))
8:      end if
9:      Randomly select feature subset A (e.g. of size √p)
10:     for each feature a ∈ A do
11:         for k = 1 to K do
12:             Sample split threshold θ_{a,k} uniformly in range of a over S
13:             Compute impurity decrease Δ_{a,k} for split (a, θ_{a,k})
14:         end for
15:     end for
16:     Choose (a*, θ*) = arg max_{a,k} Δ_{a,k}
17:     Partition S into S_L, S_R by (a*, θ*)
18:     L ← GROWTREE(S_L)
19:     R ← GROWTREE(S_R)
20:     return Node(feature = a*, threshold = θ*, left = L, right = R)
21: end function
```

Algorithm 13 Rotation Forest (Rodriguez et al., 2006)

Input: Training dataset $\mathcal{D} \in \mathbb{R}^{n\times(p+1)}$
Number of trees B
Number of features per tree, e.g., p
Output: Ensemble model $E = \{T_1, T_2, \ldots, T_{n_estimators}\}$

1: **for** $b = 1$ to B **do**
2: Randomly partition feature set $\{1, \ldots, d\}$ into K subsets $\{F_{b,k}\}_{k=1}^{K}$
3: Initialize rotation matrix $R_b \leftarrow$ identity($d \times d$)
4: **for** $k = 1$ to K **do**
5: $\mathcal{D}_{b,k} \leftarrow$ project $\mathcal{D}$ onto features in $F_{b,k}$
6: Optionally apply bootstrap sample to $\mathcal{D}_{b,k}$
7: Compute PCA on $\mathcal{D}_{b,k}$: obtain eigenvectors $P_{b,k}$
8: Insert $P_{b,k}$ into the corresponding block of R_b
9: **end for**
10: Rotate dataset: $\widetilde{\mathcal{D}} \leftarrow \{(R_m\mathbf{x}_i, y_i)\}$
11: Train base tree $T_m \leftarrow$ DecisionTree($\widetilde{\mathcal{D}}$)
12: **end for**
13: **return** E

A.9 Tree depth of GODT-C and CART for different datasets

Datasets	GODT	CART
	High-dimensional dataset	
Gisette	12	27
Madelon	9	13
Usps	13	23
	Large-scale dataset	
Acoustic	60	57
Combined	19	37
Poker	15	24
	Normal-sized dataset	
Abalone	12	24
Adult	36	41
Breast Cancer	8	7
Dermatology	14	10
Drug Consumption	18	21
Ecoli	9	6
Glass	6	9
Heart Failure	12	7
Ionosphere	6	7
Iris	6	4
Letter	19	24
Movementlibras	8	10
Mushroom	3	7
Satimages	20	18
Segmentation	7	9
Wine	10	4

TABLE A.17: Average tree depth of GODT and CART across datasets. Datasets are grouped by dimensionality and sample size. Lower values indicate more compact trees.

A.10 Ensemble size of GORF-C and RF-C three different datasets of different sizes and dimensions

Number of Trees	GORF	RF
10	70.17	64.83
25	72.83	69.87
50	73.83	73.83
100	76.00	74.50
200	76.17	74.33
300	78.33	75.33
500	78.83	76.00
800	79.33	76.17
1000	79.59	76.19

TABLE A.18: Predictive performance (accuracy, %) vs ensemble size on the Madelon dataset for GORF and Random Forest.

Number of Trees	GORF	RF
10	93.12	93.49
25	93.49	95.05
50	94.09	95.38
100	93.71	95.75
200	93.87	95.75
300	93.89	95.70
500	94.11	95.70
800	95.43	95.75
1000	96.76	95.75

TABLE A.19: Predictive performance (accuracy, %) vs ensemble size on the USPS dataset for GORF and Random Forest.

Number of Trees	GORF	RF
10	91.61	88.66
25	91.92	89.67
50	91.84	90.44
100	91.92	90.75
200	92.07	90.29
300	91.84	89.98
500	92.23	90.60
800	92.15	90.29
1000	92.66	90.60

TABLE A.20: Predictive performance (accuracy, %) vs ensemble size on the Satimages dataset for GORF and Random Forest.

A.11 Log-Log plot of increasing sample size and feature size against training time

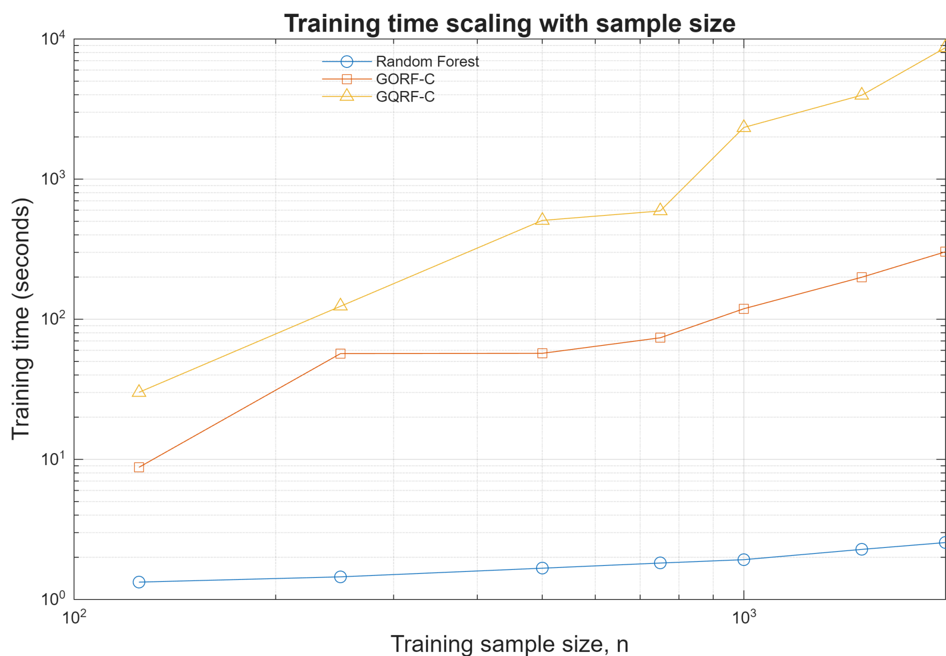


FIGURE A.1: Log–log plot of training time vs training sample size for RF, GORF-C, and GQRF-C. Here, we gradually increase the training sample size on the Madelon dataset.

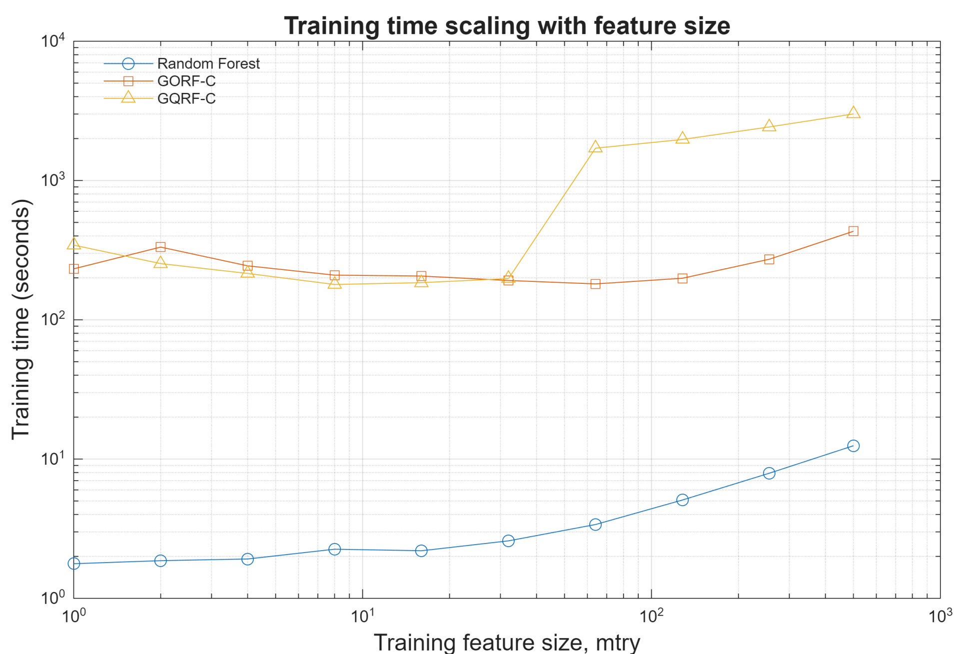


FIGURE A.2: Log–log plot of training time vs number of features considered per split (*mtry*) for RF, GORF-C, and GQRF-C. Here, we gradually increase the *mtry* on the Madelon dataset.

References

Steven Abney. Bootstrapping. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 360–367. Association for Computational Linguistics, 2002.

Firas Abuzaid, Joseph K Bradley, Feynman T Liang, Andrew Feng, Lee Yang, Matei Zaharia, and Ameet S Talwalkar. Yggdrasil: An optimized system for training deep decision trees at scale. In *NIPS*, pages 3810–3818, 2016.

Stavros P Adam, Stamatios-Aggelos N Alexandropoulos, Panos M Pardalos, and Michael N Vrahatis. No free lunch theorem: A review. *Approximation and optimization*, pages 57–82, 2019.

Fatemeh Alamdar, Fatemeh Sheykh Mohammadi, and Ali Amiri. Twin bounded weighted relaxed support vector machines. *IEEE Access*, 7:22260–22275, 2019.

Mordechai Averbuch, Tom H Karson, Benjamin Ben-Ami, Oded Maimon, and Lior Rokach. Context-sensitive medical information retrieval. *MedInfo*, 11(Pt 1):282–6, 2004.

Bart Baesens, Tony Van Gestel, Stijn Viaene, Maria Stepanova, Johan Suykens, and Jan Vanthienen. Benchmarking state-of-the-art classification algorithms for credit scoring. *Journal of the operational research society*, 54(6):627–635, 2003.

Robert E Banfield, Lawrence O Hall, Kevin W Bowyer, and W Philip Kegelmeyer. A comparison of decision tree ensemble creation techniques. *IEEE transactions on pattern analysis and machine intelligence*, 29(1):173–180, 2006.

Rodrigo C. Barros, Ricardo Cerri, Pablo A. Jaskowiak, and André C. P. L. F. de Carvalho. A bottom-up oblique decision tree induction algorithm. In *2011 11th International Conference on Intelligent Systems Design and Applications*, pages 450–456, 2011. .

Salem Benferhat, Abdelhamid Boudjelida, Karim Tabia, and Habiba Drias. An intrusion detection and alert correlation approach based on revising probabilistic classifiers using expert knowledge. *Applied intelligence*, 38(4):520–540, 2013.

Dimitris Bertsimas, Jack Dunn, and Aris Paschalidis. Regression and classification using optimal decision trees. In *2017 IEEE MIT undergraduate research technology conference (URTC)*, pages 1–4. IEEE, 2017.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006a.

Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006b.

Ferdinand Bollwein and Stephan Westphal. Oblique decision tree induction by cross-entropy optimization based on the von mises–fisher distribution. *Computational Statistics*, 37(5):2203–2229, 2022.

L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984a.

L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984b.

Leo Breiman. Bagging predictors. *Machine learning*, 24:123–140, 1996a.

Leo Breiman. Bias, variance, and arcing classifiers. Technical report, Department of Statistics, University of California, Berkeley, 1996b. URL `https://statistics.berkeley.edu/sites/default/files/tech-reports/460.pdf`.

Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984c.

Carla E Brodley and Paul E Utgoff. *Multivariate versus univariate decision trees*, volume 92. Citeseer, 1992.

Carla E Brodley and Paul E Utgoff. Multivariate decision trees. *Machine learning*, 19(1): 45–77, 1995.

Shashi Buluswar and Bruce A Draper. Nonparametric classification of pixels under varying outdoor illumination. In *Intelligent Robots and Computer Vision XIII: Algorithms and Computer Vision*, volume 2353, pages 529–536. International Society for Optics and Photonics, 1994.

M Emre Celebi, Hassan A Kingravi, and Patricio A Vela. A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert systems with applications*, 40(1):200–210, 2013.

B Chandra and P Paul Varghese. Moving towards efficient decision tree construction. *Information Sciences*, 179(8):1059–1069, 2009.

B Chandra, Ravi Kothari, and Pallath Paul. A new node splitting measure for decision tree construction. *Pattern Recognition*, 43(8):2725–2731, 2010.

Chih-Chung Chang and Chih-Jen Lin. Libsvm data: Classification, regression, and multi-label. URL https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/.

Xiaobo Chen, Jian Yang, Qiaolin Ye, and Jun Liang. Recursive projection twin support vector machine via within-class variance minimization. *Pattern Recognition*, 44 (10-11):2643–2655, 2011.

Na Chu and Jie Li. Methodology study of classification algorithm in tcm zheng diagnosis. In *2014 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 22–25. IEEE, 2014.

Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Machine learning*, 20 (3):273–297, 1995.

B. V. Dasarathy and B. V. Sheela. A composite classifier system design: Concepts and methodology. *Proceedings of the IEEE*, 67(5):708–713, 1979.

Glenn De'ath and Katharina E Fabricius. Classification and regression trees: a powerful yet simple technique for ecological data analysis. *Ecology*, 81(11): 3178–3192, 2000.

Arthur P. Dempster, Nan M. Laird, and Donald B. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)*, 39(1):1–22, 1977. .

Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

Thomas G Dietterich and Ghulum Bakiri. Solving multiclass learning problems via error-correcting output codes. *Journal of artificial intelligence research*, 2:263–286, 1994.

Dheeru Dua and Casey Graff. UCI machine learning repository, 2017. URL http://archive.ics.uci.edu/ml.

Lixin Fan. Accurate robust and efficient error estimation for decision trees. In *International Conference on Machine Learning*, pages 239–247. PMLR, 2016.

Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *The journal of machine learning research*, 15(1):3133–3181, 2014.

Artur J Ferreira and Mário AT Figueiredo. Boosting algorithms: A review of methods, theory, and applications. *Ensemble machine learning: Methods and applications*, pages 35–85, 2012.

Jerome Friedman, Trevor Hastie, and Robert Tibshirani. *The elements of statistical learning*, volume 1. Springer series in statistics New York, 2001.

Juergen Gall and Victor Lempitsky. Class-specific hough forests for object detection. *Decision forests for computer vision and medical image analysis*, pages 143–157, 2013.

MA Ganaie, Muhammad Tanveer, and Ponnuthurai N Suganthan. Oblique decision tree ensemble via twin bounded svm. *Expert Systems with Applications*, 143:113072, 2020.

Mudasir A Ganaie, Muhammad Tanveer, Ponnuthurai N Suganthan, and Václav Snásel. Oblique and rotation double random forest. *Neural Networks*, 153:496–517, 2022.

Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine Learning*, 63(1):3–42, 2006.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

Lars Kai Hansen and Peter Salamon. Neural network ensembles. *IEEE transactions on pattern analysis and machine intelligence*, 12(10):993–1001, 1990.

Peter E Hart, David G Stork, and Richard O Duda. *Pattern classification*. Wiley Hoboken, 2000.

Trevor Hastie and Robert Tibshirani. Classification by pairwise coupling. *The annals of statistics*, 26(2):451–471, 1998.

David Heath, Simon Kasif, and Steven Salzberg. Induction of oblique decision trees. In *IJCAI*, volume 1993, pages 1002–1007. Citeseer, 1993.

Jennifer A Hoeting, David Madigan, Adrian E Raftery, and Chris T Volinsky. Bayesian model averaging: a tutorial (with comments by m. clyde, david draper and ei george, and a rejoinder by the authors. *Statistical science*, 14(4):382–417, 1999.

Chih-Wei Hsu and Chih-Jen Lin. A comparison of methods for multiclass support vector machines. *IEEE transactions on Neural Networks*, 13(2):415–425, 2002.

Alan Julian Izenman. Linear discriminant analysis. In *Modern multivariate statistical techniques: regression, classification, and manifold learning*, pages 237–280. Springer, 2013.

Robert A Jacobs, Michael I Jordan, Steven J Nowlan, and Geoffrey E Hinton. Adaptive mixtures of local experts. *Neural computation*, 3(1):79–87, 1991.

Chi Jin, Yuchen Zhang, Sivaraman Balakrishnan, Martin J Wainwright, and Michael I Jordan. Local maxima in the likelihood of gaussian mixture models: Structural results and algorithmic consequences. *Advances in neural information processing systems*, 29, 2016.

Thorsten Joachims. Text categorization with support vector machines: Learning with many relevant features. In *European conference on machine learning*, pages 137–142. Springer, 1998.

Rakesh Katuwal, Ponnuthurai Nagaratnam Suganthan, and Le Zhang. Heterogeneous oblique random forest. *Pattern Recognition*, 99:107078, 2020.

Ron Kohavi, David H Wolpert, et al. Bias plus variance decomposition for zero-one loss functions. In *ICML*, volume 96, pages 275–283. Citeseer, 1996.

UH-G Krebel. Pairwise classification and support vector machines. *Advances in kernel methods: support vector learning*, pages 255–268, 1999.

Ludmila I Kuncheva and Juan J Rodríguez. An experimental study on rotation forest ensembles. In *International workshop on multiple classifier systems*, pages 459–468. Springer, 2007.

Philippe Lagacherie and Susan Holmes. Addressing geographical data errors in a classification tree for soil unit prediction. *International Journal of Geographical Information Science*, 11(2):183–198, 1997.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553): 436–444, 2015.

Wei-Yin Loh and Yu-Shan Shih. Split selection methods for classification trees. *Statistica sinica*, pages 815–840, 1997.

Wei-Yin Loh and Nunta Vanichsetakul. Tree-structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83(403):715–725, 1988.

Asdrúbal López-Chau, Jair Cervantes, Lourdes López-García, and Farid García Lamont. Fisher's decision tree. *Expert Systems with Applications*, 40(16):6283–6291, 2013.

Gilles Louppe. *Understanding random forests: From theory to practice*. Universite de Liege (Belgium), 2014.

Olvi L Mangasarian and Edward W Wild. Multisurface proximal support vector machine classification via generalized eigenvalues. *IEEE transactions on pattern analysis and machine intelligence*, 28(1):69–74, 2005.

Naresh Manwani and PS Sastry. Geometric decision tree. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(1):181–192, 2011.

MathWorks. fitgmdist — fit gaussian mixture model to data. MATLAB Documentation, a. Accessed 2026-01-14.

MathWorks. gmdistribution — create gaussian mixture model. MATLAB Documentation, b. Accessed 2026-01-14.

MathWorks. Clustering using gaussian mixture models. MATLAB Documentation, c. Accessed 2026-01-14.

Geoffrey McLachlan and David Peel. *Finite Mixture Models*. Wiley, 2000. .

Vlado Menkovski, Ioannis T. Christou, and Sofoklis Efremidis. Oblique decision trees using embedded support vector machines in classifier ensembles. In *2008 7th IEEE International Conference on Cybernetic Intelligent Systems*, pages 1–6, 2008. .

Bjoern H Menze, B Michael Kelm, Daniel N Splitthoff, Ullrich Koethe, and Fred A Hamprecht. On oblique random forests. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2011, Athens, Greece, September 5-9, 2011, Proceedings, Part II 22*, pages 453–469. Springer, 2011.

Cleve Moler. Numerical computing with matlab. *SIAM*, 2004. URL `https://www.mathworks.com/moler/chapters.html`.

Sreerama Murthy, Simon Kasif, Steven Salzberg, and Richard Beigel. Oc1: A randomized induction of oblique decision trees. In *AAAI*, volume 93, pages 322–327. Citeseer, 1993.

Sreerama K Murthy, Simon Kasif, and Steven Salzberg. A system for induction of oblique decision trees. *Journal of artificial intelligence research*, 2:1–32, 1994.

Sai-Cheong Ng and Kwong-Sak Leung. Induction of quadratic decision trees using genetic algorithms and k-d trees. *WSEAS Transactions on Computers*, 3(3):839–845, 2004.

Kamal Nigam, Andrew Kachites McCallum, Sebastian Thrun, and Tom Mitchell. Text classification from labeled and unlabeled documents using em. *Machine learning*, 39: 103–134, 2000.

Nils J Nilsson. Survey of pattern recognition. *Annals of the New York Academy of Sciences*, 161(2):380–401, 1969.

Mohammad Norouzi, Maxwell D Collins, David J Fleet, and Pushmeet Kohli. Co2 forest: Improved random forest by continuous optimization of oblique splits. *arXiv preprint arXiv:1506.06155*, 2015.

Sebastian Nowozin, Carsten Rother, Shai Bagon, Toby Sharp, Bangpeng Yao, and Pushmeet Kohli. Decision tree fields. In *2011 International Conference on Computer Vision*, pages 1668–1675. IEEE, 2011.

Guansong Pang, Huidong Jin, and Shengyi Jiang. Cenknn: a scalable and effective text classifier. *Data Mining and Knowledge Discovery*, 29(3):593–625, 2015.

Branislav Panić, Jernej Klemenc, and Marko Nagode. Improved initialization of the em algorithm for mixture model parameter estimation. *Mathematics*, 8(3):373, 2020.

Jigar Patel, Sahil Shah, Priyank Thakkar, and Ketan Kotecha. Predicting stock market index using fusion of machine learning techniques. *Expert systems with applications*, 42(4):2162–2172, 2015.

Sergio Pereira, Américo Oliveira, Victor Alves, and Carlos A Silva. On hierarchical brain tumor segmentation in mri using fully convolutional neural networks: a preliminary study. In *2017 IEEE 5th Portuguese meeting on bioengineering (ENBENG)*, pages 1–4. IEEE, 2017.

John C Platt, Nello Cristianini, John Shawe-Taylor, et al. Large margin dags for multiclass classification. In *nips*, volume 12, pages 547–553, 1999.

Anantha M Prasad, Louis R Iverson, and Andy Liaw. Newer classification and regression tree techniques: bagging and random forests for ecological prediction. *Ecosystems*, 9:181–199, 2006.

Philipp Probst, Marvin N Wright, and Anne-Laure Boulesteix. Hyperparameters and tuning strategies for random forest. *Wiley Interdisciplinary Reviews: data mining and knowledge discovery*, 9(3):e1301, 2019.

Yanjun Qi, Harpreet K Dhiman, Neil Bhola, Ivan Budyak, Siddhartha Kar, David Man, Arpana Dutta, Kalyan Tirupula, Brian I Carr, Jennifer Grandis, et al. Systematic prediction of human membrane receptor interactions. *Proteomics*, 9(23):5243–5255, 2009.

Zhiquan Qi, Yingjie Tian, and Yong Shi. Robust twin support vector machine for pattern classification. *Pattern Recognition*, 46(1):305–316, 2013.

J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

J Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

Reshma Rastogi and Anubhav David. Oblique random forest via regularized multisurface proximal support vector machine. In *2019 Global Conference for Advancement in Technology (GCAT)*, pages 1–6, 2019. .

Mohamed Amine Remita, Ahmed Halioui, Bruno Daigle, Golrokh Kiani, Abdoulaye Baniré Diallo, et al. A machine learning approach for viral genome classification. *BMC bioinformatics*, 18(1):1–11, 2017.

Ye Ren, Le Zhang, and Ponnuthurai N Suganthan. Ensemble classification and regression-recent developments, applications and future directions. *IEEE Computational intelligence magazine*, 11(1):41–53, 2016.

Douglas A Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741: 659–663, 2009.

Bharat Richhariya and Muhammad Tanveer. A robust fuzzy least squares twin support vector machine for class imbalance learning. *Applied Soft Computing*, 71: 418–432, 2018.

Juan José Rodriguez, Ludmila I Kuncheva, and Carlos J Alonso. Rotation forest: A new classifier ensemble method. *IEEE transactions on pattern analysis and machine intelligence*, 28(10):1619–1630, 2006.

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

Steven L Salzberg. C4. 5: Programs for machine learning by j. ross quinlan. morgan kaufmann publishers, inc., 1993, 1994.

Robert E Schapire et al. A brief introduction to boosting. In *Ijcai*, volume 99, pages 1401–1406. Citeseer, 1999.

Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM computing surveys (CSUR)*, 34(1):1–47, 2002.

Yuan-Hai Shao, Nai-Yang Deng, Zhi-Min Yang, Wei-Jie Chen, and Zhen Wang. Probabilistic outputs for twin support vector machines. *Knowledge-Based Systems*, 33: 145–151, 2012.

Sweta Sharma, Reshma Rastogi, and Suresh Chandra. Large-scale twin parametric support vector machine using pinball loss function. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, 2019.

Renato M Silva, Tiago A Almeida, and Akebo Yamakami. Mdltext: An efficient and lightweight text classifier. *Knowledge-Based Systems*, 118:152–164, 2017.

James V Stone. *Bayes' rule: a tutorial introduction to Bayesian analysis*. Sebtel Press, 2013.

José L Subirats, José M Jerez, Iván Gómez, and Leonardo Franco. Multiclass pattern recognition extension for the new c-mantec constructive neural network algorithm. *Cognitive Computation*, 2(4):285–290, 2010.

Vladimir Svetnik, Andy Liaw, Christopher Tong, J Christopher Culberson, Robert P Sheridan, and Bradley P Feuston. Random forest: a classification and regression tool for compound classification and qsar modeling. *Journal of chemical information and computer sciences*, 43(6):1947–1958, 2003.

Paul E Utgoff and Carla E Brodley. An incremental method for finding multivariate splits for decision trees. In *Machine Learning Proceedings 1990*, pages 58–65. Elsevier, 1990.

Paul E Utgoff and Carla E Brodley. Linear machine decision trees. 1991.

N Denizcan Vanli, Muhammed O Sayin, Mohammadreza Mohaghegh, Huseyin Ozkan, and Suleyman S Kozat. Nonlinear regression via incremental decision trees. *Pattern Recognition*, 86:1–13, 2019.

K VijiyaKumar, B Lavanya, I Nirmala, and S Sofia Caroline. Random forest algorithm for the prediction of diabetes. In *2019 IEEE international conference on system, computation, automation and networking (ICSCAN)*, pages 1–5. IEEE, 2019.

Fei Wang, Quan Wang, Feiping Nie, Weizhong Yu, and Rong Wang. Efficient tree classifiers for large scale datasets. *Neurocomputing*, 284:70–79, 2018.

Fei Wang, Quan Wang, Feiping Nie, Zhongheng Li, Weizhong Yu, and Fuji Ren. A linear multivariate binary decision tree classifier based on k-means splitting. *Pattern Recognition*, 107:107521, 2020.

Ran Wang, Sam Kwong, Xi-Zhao Wang, and Qingshan Jiang. Segment based decision tree induction with continuous valued attributes. *IEEE transactions on cybernetics*, 45 (7):1262–1275, 2014.

DC Wickramarachchi, BL Robertson, Marco Reale, Christopher John Price, and J Brown. Hhcart: An oblique decision tree. *Computational Statistics & Data Analysis*, 96:12–23, 2016.

David H Wolpert. Stacked generalization. *Neural networks*, 5(2):241–259, 1992.

Robert F Woolson. Wilcoxon signed-rank test. *Wiley encyclopedia of clinical trials*, pages 1–3, 2007.

Xindong Wu, Vipin Kumar, J Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J McLachlan, Angus Ng, Bing Liu, Philip S Yu, et al. Top 10 algorithms in data mining. *Knowledge and information systems*, 14:1–37, 2008.

Yitian Xu, Rui Guo, and Laisheng Wang. A twin multi-class classification support vector machine. *Cognitive computation*, 5(4):580–588, 2013.

Bin-Bin Yang, Song-Qing Shen, and Wei Gao. Weighted oblique decision trees. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 5621–5627, 2019.

Olcay Taner Yildiz and Ethem Alpaydin. Linear discriminant trees. *International Journal of Pattern Recognition and Artificial Intelligence*, 19(03):323–353, 2005.

Olcay Taner Yıldız and Ethem Alpaydın. Comparing univariate and multivariate decision trees. 2012.

Cha Zhang and Yunqian Ma. *Ensemble Machine Learning: Methods and Applications*. Springer Publishing Company, Incorporated, 2012. ISBN 1441993258.

Chun-Xia Zhang and Jiang-She Zhang. Rotboost: A technique for combining rotation forest and adaboost. *Pattern recognition letters*, 29(10):1524–1536, 2008.

Le Zhang and Ponnuthurai N. Suganthan. Oblique decision tree ensemble via multisurface proximal support vector machine. *IEEE Transactions on Cybernetics*, 45 (10):2165–2176, 2015. .

Le Zhang and Ponnuthurai Nagaratnam Suganthan. Benchmarking ensemble classifiers with novel co-trained kernel ridge regression and random vector functional link ensembles [research frontier]. *IEEE Computational Intelligence Magazine*, 12(4):61–72, 2017. .

Le Zhang, Jagannadan Varadarajan, Ponnuthurai Nagaratnam Suganthan, Narendra Ahuja, and Pierre Moulin. Robust visual tracking using oblique random forests. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5589–5598, 2017.

Zhi-Hua Zhou and Ji Feng. Deep forest. *arXiv preprint arXiv:1702.08835*, 2017.