Applying Data Mining for the Recognition of Digital Ink Strokes

Samuel Hsiao-Heng Chang

Under supervision of Dr Beryl Plimmer

A thesis submitted in fulfilment of the requirements for the degree of

Master of Engineering in Software Engineering

The University of Auckland, Feb 2010

Abstract

The objective of this research is to improve the recognition of hand drawn diagrams. To accurately recognise an object, a recogniser should be able to utilise the rich information stored in an ink stroke. To reduce the process time required to develop a recogniser, a recogniser should be easily extendable. While recognition techniques based on machine learning satisfied both requirements above, they are still limited; the stroke information utilised is mostly decided heuristically, and there is no systematic comparison between the available algorithms. Therefore this research is focused on improving the existing sketch recognition by combining a rich feature set and WEKA, a data mining tool.

Our review of literature shows the different approaches in sketch recognition, and reveals the strength and weakness of each. It also demonstrates the promising results obtained in related areas with data mining. We analyse the data mining algorithms implemented in WEKA, and select nine from among them. These algorithms are optimised on three diagram sets by altering their settings, and we assume this optimisation can be applied to any diagram set. We then rank and combine these algorithms to obtain higher performance. Furthermore we apply data mining to select better features for Rubine, which originally applied a heuristically decided feature set, to improve its accuracy.

The results of these analyses are implemented into Rata.SSR, a recogniser generator which can generate recognisers with input training examples. It provides a simple interface for training and using the recognisers. We trained several classifiers and evaluated them against existing classifiers including Cali, OneDollarRecogniser, PaleoSketch, DTW and Microsoft Recogniser. The best recogniser among the existing ones produces 89.7% recognition rate in average across four datasets, while our worst recogniser, which uses Bagging algorithm produces 94.9% and our best recogniser, using a Voting mechanism, achieves 98.0% recognition rate. We also demonstrated the Rubine classifier can be improved by attribute selection, which improves accuracy from 87.1% to 96.7%. Therefore the recognition rate is improved by the application of data mining.

Acknowledgements

I would like to acknowledge first and foremost, my supervisor, Dr. Beryl Plimmer, for guiding me not only academically, but also in many other areas. Thank you for training me with many fascinating projects, and giving me opportunities to attend different conferences and learn different things. It is great to be your student.

Thanks to Associate Professor Eibe Frank, for patiently answering my questions on data mining. Your advice solved many questions which confused me for a long time.

To Dad and Mom, thank you for giving me support and encouragement, and caring more about my health and happiness than my grade. To Shirley and Christina, thanks for always being keen to hear my problems and trying to relax my stressed mind by offering holiday plans.

Thanks to Rachel, for all the discussions and suggestions; I will always remember our computation-power hunting time. Thanks to Nilu and John, for sharing the news and jokes in the HCI lab, and motivating me with your hard working attitudes. Thanks to Paul and Andrew, for the useful feedback you provided to this research.

Thanks to Callia and Tim, for all the crazy ideas, friendly teasing and endless laughter, when we really should be studying. Thanks to Sherry, for all the delicious food and the amusements you prepared to remove my stress.

Thanks to all the people who participated in the data collection.

Thanks to all of my teachers, particularly Hui-Chen and Hsiao-Chuan, for you corrected me when I was wrong and guided me to the right path. To my brothers and sisters in Christ, your prayers are supporting me even now.

Lastly, to my Lord and shepherd Jesus Christ, who leads me here and will be guiding my life forever.

wordle.net Thesis Visualization

Hmmm, why there is no Samuel...........?

Table of Contents

1 Introduction 1

1.1. Motivation 2

1.2. Objectives 3

1.3. Outline 5

1.4. Definitions 6

2 Related Work 7

2.1. Digital Ink Shape Recognition 7

2.2. Data Mining in Related Domains 28

2.3. Software Used 31

2.4. Summary 32

3 Overview 35

3.1. Data Collection and Process 35

3.2. Data Mining 37

3.3. Implementation 44

3.4. Evaluation 45

3.5. Summary 45

4 Data Mining 47

4.1. Data Collection and Preparation 47

4.2. Scope of Algorithms 54

4.3. WEKA Algorithms 55

4.4. Combined WEKA Algorithms 108

4.5. Rubine 113

4.6. Summary 116

5 Implementation 117

5.1. Overview 117

5.2. Architecture 121

5.3. Training 125

5.4. Classifying 131

5.5. Extension Opportunities 135

5.6. Summary 137

6 Evaluation 139

6.1. Outline 139

6.2. Experiment against Other Classifiers 143

6.3. Using the ShapeData to Recognise Other Datasets 148

6.4. Summary 152

7 Discussion ... 153
7.1. Data Mining ... 153
7.2. Nature of Diagrams ... 157
7.3. Implementation ... 160
7.4. Limitations ... 161
8 Conclusion and Future Work ... 163
8.1. Conclusion ... 163
8.2. Future Work ... 164
Appendix A: The Questionnaire ... 167
Appendix B: The Information Sheets ... 168
Appendix C: The Instructions ... 170
References ... 171

List of Figures

Figure 1. Two shapes to be recognised 7

Figure 2. The divisions of previous work 9

Figure 3. Example using Grid comparison 10

Figure 4. $1 processes(Wobbrock et al., 2007). (a) Difference in stroke times (b) Rotation of strokes 11

Figure 5. Sketched symbol and their beautified version (Hse & Newton, 2005) 12

Figure 6. Examples of symbol templates(Kara & Stahovich, 2004) 12

Figure 7. The system overview (Ouyang & Davis, 2009) 13

Figure 8. Training samples for letter C (Gross, 1994) 13

Figure 9. The semantic network of a square (Calhoun et al., 2002) 15

Figure 10. Demonstration of the mechanism (Sezgin & Davis, 2005) 16

Figure 11. An example of segmentation (Sezgin et al., 2001) 18

Figure 12. Calculating the curvature sign. The window includes 9 points(Calhoun et al., 2002) 19

Figure 13. Feature area examples(Yu & Cai, 2003) 19

Figure 14. the forward step when finding a possible end segment for a symbol(Gennari et al., 2004) 20

Figure 15. CALI polygons used to estimate feature (Fonseca & Jorge, 2000) 22

Figure 16. Recognition rate vs. training set size (Rubine, 1991) 25

Figure 17. Domains and their sub-domains 36

Figure 18. Comparison between cross validations: (a) based on sample, (b) based on participant 39

Figure 19. ShapeData from participant 14 and 19 48

Figure 20. GraphData from participant 14 and 19 48

Figure 21. Flowchart diagram from participant 14 and 19 48

Figure 22. Class diagram from participant 14 and 19 49

Figure 23. Self ranked familiarity 50

Figure 24. Instruction for drawing ClassData 51

Figure 25. Self ranked complexity of each task 51

Figure 26. Bagging: Bag Size Percent 57

Figure 27. Bagging: Accuracy vs. Graph types 58
Figure 28. Bagging: Accuracy vs. Num Iterations 58
Figure 29. Bagging: Optimise experiment..................59
Figure 30. Bagging: RandomSplitting 60
Figure 31. Bagging:OrderedSplitting..................61
Figure 32. RandomForest: Max depth 62
Figure 33. RandomForest: Number of Features 63
Figure 34. RandomForest: Number of Trees 64
Figure 35. RandomForest: Optimise experiment..................65
Figure 36. RandomForest: RandomSplitting 65
Figure 37. RandomForest: OrderedSplitting..................66
Figure 38. LogitBoost: Likelihood threshold..................68
Figure 39. LogitBoost: NumFolds 69
Figure 40. LogitBoost: NumIterations 69
Figure 41. LogitBoost: Shrinkage..................70
Figure 42. LogitBoost: UseResampling..................70
Figure 43. LogitBoost: Weight threshold 71
Figure 44. LogitBoost: Optimise experiment 72
Figure 45. LogitBoost: RandomSplitting..................73
Figure 46. LogitBoost: OrderedSplitting 73
Figure 47. END: Classifier 75
Figure 48. END:.NumIterations..................75
Figure 49. END: Optimise experiment 76
Figure 50. END: RandomSplitting 77
Figure 51. END: OrderedSplitting..................77
Figure 52. Example ADTree in WEKA(Witten & Frank, 2005)..................78
Figure 53. LADTree: Num of Boosting Iterations..................79
Figure 54. LADTree: Optimise experiment..................79
Figure 55. LADTree: RandomSplitting 80

Figure 56. LADTree:OrderedSplitting....81
Figure 57. LMT: ErrorOnProbabilities....82
Figure 58. LMT:.FastRegression....83
Figure 59. LMT: MinNumInstances....83
Figure 60. LMT: NumBoostingIterations....84
Figure 61. LMT: UseAIC....85
Figure 62. LMT: WeightTrimBeta....85
Figure 63. LMT: Optimise experiment....86
Figure 64. LMT: RandomSplitting....86
Figure 65. LMT: OrderedSplitting....87
Figure 66. Bayes Network: SearchAlgorithm....88
Figure 67. Bayes Network: Optimise experiment....89
Figure 68. Bayes Network: RandomSplitting....89
Figure 69. Bayes Network: OrderedSplitting....90
Figure 70. Example artificial neuron network....90
Figure 71. MultilayerPerceptron: Decay....92
Figure 72. MultilayerPerceptron: HiddenLayers....92
Figure 73. MultilayerPerceptron: LearningRate....93
Figure 74. MultilayerPerceptron: Momentum....93
Figure 75. MultilayerPerceptron: NominalToBinaryFilter....94
Figure 76. MultilayerPerceptron: NormaliseAttributes....94
Figure 77. MultilayerPerceptron: TrainingTime....94
Figure 78. MultilayerPerceptron: ValidationSetSize....95
Figure 79. MultilayerPerceptron: Optimise experiment....96
Figure 80. MultilayerPerceptron: RandomSplitting....96
Figure 81. MultilayerPerceptron: OrderedSplitting....97
Figure 82. A linear support vector machine (Platt, 1998)....98
Figure 83. SMO: BulidLogisticModels....99
Figure 84. SMO: C....99

Figure 85. SMO: FilterType 100
Figure 86. SMO: Kernel 100
Figure 87. SMO: Optimise Experiment 101
Figure 88. SMO: RandomSplitting 102
Figure 89. SMO: OrderedSplitting 102
Figure 90. Accuracy difference for each algorithm 105
Figure 91. Performance gain 106
Figure 92. Voting: Combination and optimisation 109
Figure 93. Voting: Different combination 110
Figure 94. Stacking: Combination 111
Figure 95. Voting, Stacking and the best performing algorithm 112
Figure 96. Accuracy vs number of attributes 114
Figure 97. Orignial Rubine v.s. Attribute Selected Rubine(11) 115
Figure 98. Rata.SSR use case. (a)training (b)classifying 119
Figure 99. System architecture 122
Figure 100. Data formats 123
Figure 101. The Rata.SSR training interface 125
Figure 102. Rata.SSR training interface: General Train 126
Figure 103. Rata.SSR training interface: Algorithm 126
Figure 104. Training a recogniser 127
Figure 105. Simplified structure of Rubine. a) the original implementation, b)the modified version 129
Figure 106. Rata.SSR training interface: Status area showing classifying results 130
Figure 107. Loading a recogniser 131
Figure 108. Classify one stroke at a time 134
Figure 109. Rata.SSR training interface: Paint interface example use. (a. Before b. After) 135
Figure 110. WEKA explorer to setup END algorithm 136
Figure 111. Rata.SSR training interface: Rubine Customise 136
Figure 112. Shapes recognisable by CALI (Fonseca et al., 2002) 141

Figure 113. Result of BN, LB and Bagging on CLASS drawn by participant 17(red means wrongly recognised)....145

Figure 114. Recognition accuracy for individual shape classes 146

Figure 115. Two examples from LogistBoost in ClassData(1-10), strokes in red are the misclassified ones 147

Figure 116. ShapeData comparison: Recognition accuracy for individual shape classes151

Figure 117. Connectors in ClassData....157

List of Tables

Table 1. Filters used by Apte et al. (1993) 21

Table 2. The different datasets 48

Table 3. Algorithms with accuracy over 90%. Final candidates are shaded in red 54

Table 4. Bagging options(Hall et al., 2009) 57

Table 5. RandomForest options(Hall et al., 2009) 62

Table 6. LogitBoost options(Hall et al., 2009) 67

Table 7. END options(Hall et al., 2009) 74

Table 8. END testing time comparison 76

Table 9. LADTree options(Hall et al., 2009) 79

Table 10. LMT options(Hall et al., 2009) 81

Table 11. Bayes network options(Hall et al., 2009) 88

Table 12. MultilayerPerceptron options(Hall et al., 2009) 91

Table 13. SMO options(Hall et al., 2009) 98

Table 14. Results of Default settings vs Attribute selected classifiers 103

Table 15. The comparison between FlowChart and the averaged data from other datasets 106

Table 16. Algorithm rankings 108

Table 17. The algorithms used 109

Table 18. The average accuracy with 10 fold cross validation 110

Table 19. The maximum number of selected attributes 114

Table 20. Rubine testing whole dataset 115

Table 21. Attribute selection from original Rubine 116

Table 22. The description of each groupbox in Rata.SSR training interface 125

Table 23. The different versions of ClassifierClassify 132

Table 24. Evaluation result 144

Table 25. Accuracy (average of the five WEKA algorithms used in evaluation) in all datasets compared to participant skills (Skills in Likert scale, 5 used frequently <-> 1 never used before) 147

Table 26. Different schemes for training example 149

Table 27. Comparison of diagram collection and shape collection 150

Chapter 1
Introduction

Sketches are informal scribbles representing ideas. Widely applied in early design, they allow the designers to capture their thoughts and create concrete representations. Although computers attempt to replace this traditional method of design, designers still prefer pen and paper when exploring and communicating the structure of a design (Forbus, Usher, & Chapman, 2003), which usually leads to better results (Brereton & McGarry, 2000).

Regardless of the power of pen and paper, in the modern world virtually nothing can be designed without computers. Architecture and engineering sketches need to be redrawn with CAD tools, UML and flowcharts need be translated into digital formats, and even the traditional cartoon characters are recreated as 3D models. Many advantages come with the electronic versions of design. For example, piles of design sheets can be stored in a small hard drive; they can be easily found, duplicated, transferred and modified.

Hence, often designers design with pen and paper first, and then sit in front of their computers, dragging lines and boxes to translate the design from paper into the digital world (Bartolo, Farrugia, Camilleri, & Borg, 2008; Schweikardt & Gross, 2000). This redundant process reduced their productivity. A tool capable of converting drawings done with pen and paper to their digital format can save a great deal of time and effort. This idea is explored by off-line recognition studies (Johnson, Gross, Hong, & Do, 2009). These studies are mostly graph-based, because the only information presented is the digitalised image of the original drawing.

Richer information can be brought in with alternative input methods. In 1963, Sutherland (1963) announced his work on Sketchpad, which allows a person to make stylus input with a digital pen. Many devices are developed after that. Today, the market for tablet PC and touch screen continues to grow. These devices allow users to interact with them by input with styluses, similar to pen and paper. When compared with an image scanned from paper, which is used in off-line recognition, more information can be recorded with these styluses including temporal, spatial or even pressure information. Research utilised these features and developed on-line recognition techniques. They allow the interpretation

of digital stylus, and are applied in various areas including architecture, engineering, software modelling and user interface design (Plimmer & Apperley, 2004).

The interpretation of these styluses can be roughly broken into two branches, the eager recognition and the lazy recognition (Johnson et al., 2009). Eager recognition means a stroke is recognised immediately after it is drawn, while lazy recognition means the recognition phase will be triggered later when all information is provided. Most modern recognisers have applied lazy approach, claiming that it cause less distraction to users (Kara & Stahovich, 2004). However, because an eager system can be used for lazy implementation, we aim to support eager recognition for more flexibility under different situations, for example, dynamic editing of graph elements (Purchase, Plimmer, Baker, & Pilcher, 2010). The typical recognition process for eager recognition can be expressed with three steps: the recogniser receives a stylus input stroke, recognises what shape the stroke represents, and returns the name of the shape.

Accuracy in recognising sketched symbols is doubtless the most important element, and also the most complex to achieve. According to Goel (1991) sketches have overloaded semantics, and are ambiguous, dense and replete with information (Shilman, Pasula, Russell, & Newton, 2002). Furthermore much noise can be found with free hand sketches (Sezgin, Stahovich, & Davis, 2001). A tool which accurately recognises sketches can therefore only be implemented by wise use of the information provided by the input stylus. However, it may require considerable time and expert knowledge. The goal of this research is to create a recogniser generator which applies data mining and allows fast and automatic training of reliable recognisers.

1.1. Motivation

A bridge can be built between designing with pen and paper and the use of digital power. However, the core of this bridge is the recognition accuracy. When the purpose of a sketch recogniser is to recognise elements of the sketch, failures in doing this will diminish its value.

No single recogniser can recognise everything, if “everything” means all diagrams including those yet to be created. Even if a recogniser has good performance in a specific diagram domain (for example, a flowchart recogniser), if it is hard coded and cannot be extended to another diagram domain, it becomes less cost effective. Some recognisers

tried to support multiple common shape types which appear in different diagrams. For example, a flowchart recogniser is able to work with directed graph, since all shape used by directed graph can be found in flowcharts. Although this approach increases the number of supported diagram domains, it still cannot cope with every existing diagram; furthermore the diagram which does not use all the supported shapes can display reduced accuracy, because the recogniser is forced to decide from among more candidates than it needs to; lastly much of the information associated with graph cannot be utilised, including for example the ratio of shapes and how they are usually drawn.

Machine learning can reduce the weakness of hard coded methods. By training recognisers with examples, different diagrams can be supported as long as training data is given. However, existing research in sketch recognition rarely compares the performance of different algorithms. Their performance cannot be judged through the reported result since they can be biased toward the data used, and no collected data can represent the complete hypothesis space (Schmieder, Plimmer, & Blagojevic, 2009). Furthermore, many recognisers are created from scratch, when similar problems are already explored in other problem domains.

Data mining has been successfully applied in different domains where rich information is presented. Sketch information is rich where temporal, spatial and pressure information can be extracted from a single stroke. We are hence motivated to explore the possibility of applying data mining techniques to create a meta-program, which can generate accurate recognisers automatically.

1.2. Objectives

The primary objective of this project is to investigate if the application of data mining techniques can improve the sketch recognition. There are many kinds of sketches, and we will focus on sketched diagrams. We define a “diagram” as a defined set of symbols; UML diagram is an example, and gesture recognition also fits the definition; on the other hand, art sketches are not considered as diagrams because there is no defined set of symbols. The hypothesis is that the recognition accuracy can be improved with the application of data mining. Because there are many data mining techniques, we are also interested in finding the best techniques for diagram recognition.

The knowledge in data mining will then be used to build recognisers. Improvement can be broken down into three parts: the accuracy, the extensibility and the cost. Accuracy is the most important element for all recognition systems. Researchers applied extensive effort in developing ultimate recognisers. Yet, to our knowledge there exists no perfect recogniser.

Extensibility is another issue. Different diagrams contain different types of elements. While fine tuning the code can make it achieve high accuracy, such a process will decrease the extensibility. For example if a recogniser is specifically build for recognising UML diagrams, it will require extra effort to make modifications so it can recognise flowcharts.

The main reason extensibility is desired is to reduce the cost of building a classifier, and that mainly depends on the time required to build a recogniser. If a recogniser takes one year to tailor, extensibility will certainly enhance its value. On the other hand, if building a recogniser only requires one day without program code and input from recognition specialists, the built recogniser does not have to be flexible – if it is not suitable, another day's work can replace it. This is the level of automation we are aiming for, to remove the required time and expert knowledge to implement and tailor a recogniser.

We aim to create eager recognisers, which require the recognition to be fast enough so the creative process will not be interrupted. People naturally draw with differing numbers of strokes; for example, a rectangle can be completed with one stroke or four or even more strokes. Most multi-stroke recognisers applied a technique called "joining" which combines different parts of a stroke into one joined shape. Although a multi-stroke recogniser is more applicable in the real world, we decided to restrict the study to single-stroke recognition (where each shape must be completed with only one stroke). This allows us to concentrate on data mining, and the result can be applied perfectly to multi-stroke shapes that have been joined.

Based on these requirements, the objectives are formed:

- To identify the strengths of different data mining algorithms and find the ones which can generate algorithms that are most accurate in the domain of sketched diagram recognition

- To improve the extensibility and to reduce the cost of a recogniser implementation. We intend to automate the process of creating a recogniser, so the tool can be applied in different situations with minimal modification

To prove our argument, a recogniser generator will be implemented which can generate optimised recognisers based on input data. The generator itself must be easy to use and portable, while the generated recognisers must perform at least as well as the existing recognisers.

1.3. Outline

The remainder of this thesis is organised as follows.

Chapter 2 presents the related work of this project. It starts with a review of publications in digital ink shape recognition, which are divided into three different groups according to their approach. This follows with successful applications of data mining in domains other than digital ink shape recognition, including character recognition and speech recognition. The chapter ends with an introduction to the software applications this project uses.

Chapter 3 gives an overview to this project, and discusses several decisions made to data mining and to the building of the recogniser generator. This includes the method of data collection, the design of experiments to evaluate and improve the performance of algorithms, the plans to implement the recogniser generator, and the way recognisers will be evaluated.

Data mining will be discussed in Chapter 4. The chapter starts with the procedure on collecting and processing data. The experiments on different algorithms follow; each is evaluated to show its optimised performance toward diagram recognition. The way to improve Rubine's algorithm (Rubine, 1991) is then discussed.

The implementation of the recogniser generator is explained in Chapter 5. We will discuss how the technology used is selected, explain how the architecture is formed, and describe how the algorithms are implemented into the generator.

Evaluation of the generated classifiers will be presented in Chapter 6. In the first experiment the data mining classifiers are tested against existing classifiers. The second experiment tests the difference between different data collection methods.

Chapter 7 presents a discussion of the observations and contributions. Conclusion and opportunities for future work will be drawn in chapter 8.

1.4. Definitions

The following definitions are used throughout the entire document.

Term	Definition
Stroke	A single stylus action, captured from stylus down to stylus up. Pressure, time and location data are recorded
Shape	A symbol which is not text
Feature	Attribute, measures certain characteristic of the stroke referred to
Sketch	Hand drawn diagram or a collection of strokes
Diagram	Sketches with a defined set of shapes
Algorithm	The mechanism used to generate classifier based on input values
Classifier	The artefact generated by data mining algorithm, which can be used to classify the input data
Recogniser	Tool implemented to classify input strokes
Eager recognition	A stroke is recognised when it is drawn
In-situ-collection	Data collected by requiring participants to draw a complete diagram, all shapes within relate to one another as part of the diagram
Isolated-collection	Collect data by collecting individual shapes separately
TSC-features	Temporal Spatial Context features. These features consider both temporal features such as inter-stroke (previous/next strokes) time and distances, and the attributes of other strokes in the graph such as the closest ones. They are only considered in in-situ-collection

Chapter 2
Related Work

This chapter outlines the related work for the thesis. In section 2.1, past approaches in digital ink recognition are analysed. In section 2.2, approaches in similar areas are explored. Section 2.3 introduces the software tools used in this project.

2.1. Digital Ink Shape Recognition

Humans are good at recognising the similarity between shapes. For example, one can easily judge if the shapes in Figure 1 are the same, and decide what shape they are. The goal of digital ink recognition is to make computers perform the task as well as humans do.

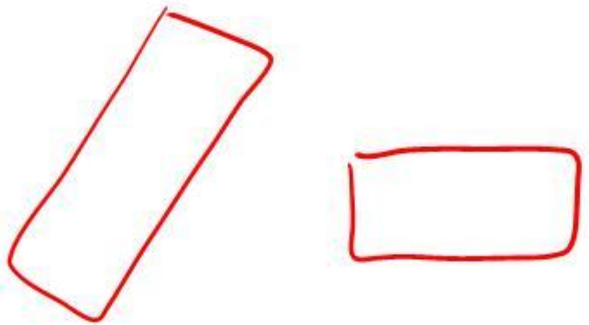


Figure 1. Two shapes to be recognised

The methods for ink recognition are many and varied. In this section we will focus on the differences in their extensibility and the ways information is utilised.

Extensibility indicates how easy it is to make a recogniser recognise a new shape. It is briefly discussed in section 1.2. The level of extensibility is mostly associated with the level of machine learning, because we can only reduce the amount of human work by substituting it with machine work, where we allow machines to use ink information efficiently (Hammond et al., 2008).

The input of a stroke recogniser is the stylus, which is collected by computers through the specialised hardware and transferred into data format defined by the programming language used. In this thesis they are called strokes. These strokes can be processed by recognisers to deduce different kinds of information. For example, one recogniser may

change the strokes into a JPEG picture, which may be used for an off-line algorithm. Another may change the strokes into a number of numeric values suitable for data mining algorithms. How information is used can greatly affect the implementation and attribute of a classifier. Recognisers which rely heavily on data mining usually change a stroke into temporal and geometrical features. We hence decided the other extreme will be close to off-line recognition where only the basic geometric information, such as pixel images, is used, and build a continuum between these extremes.

Based on these two dimensions, the approaches are separated into three categories: hard coded, if it has low extensibility and requires modification of code to allow recognition of new shapes; template comparison, if it can be trained but uses a limited number or type of information; training based, if it can be trained and many different kinds of information are presented. These research are plotted onto a graph, as shown in Figure 2 with the type of feature used on the horizontal axis and the level of extensibility on the vertical axis.

The categorization is based on the published information on the recognisers. If a recogniser applies several approaches which fall into multiple categories, each aspect will be described separately in their related section.

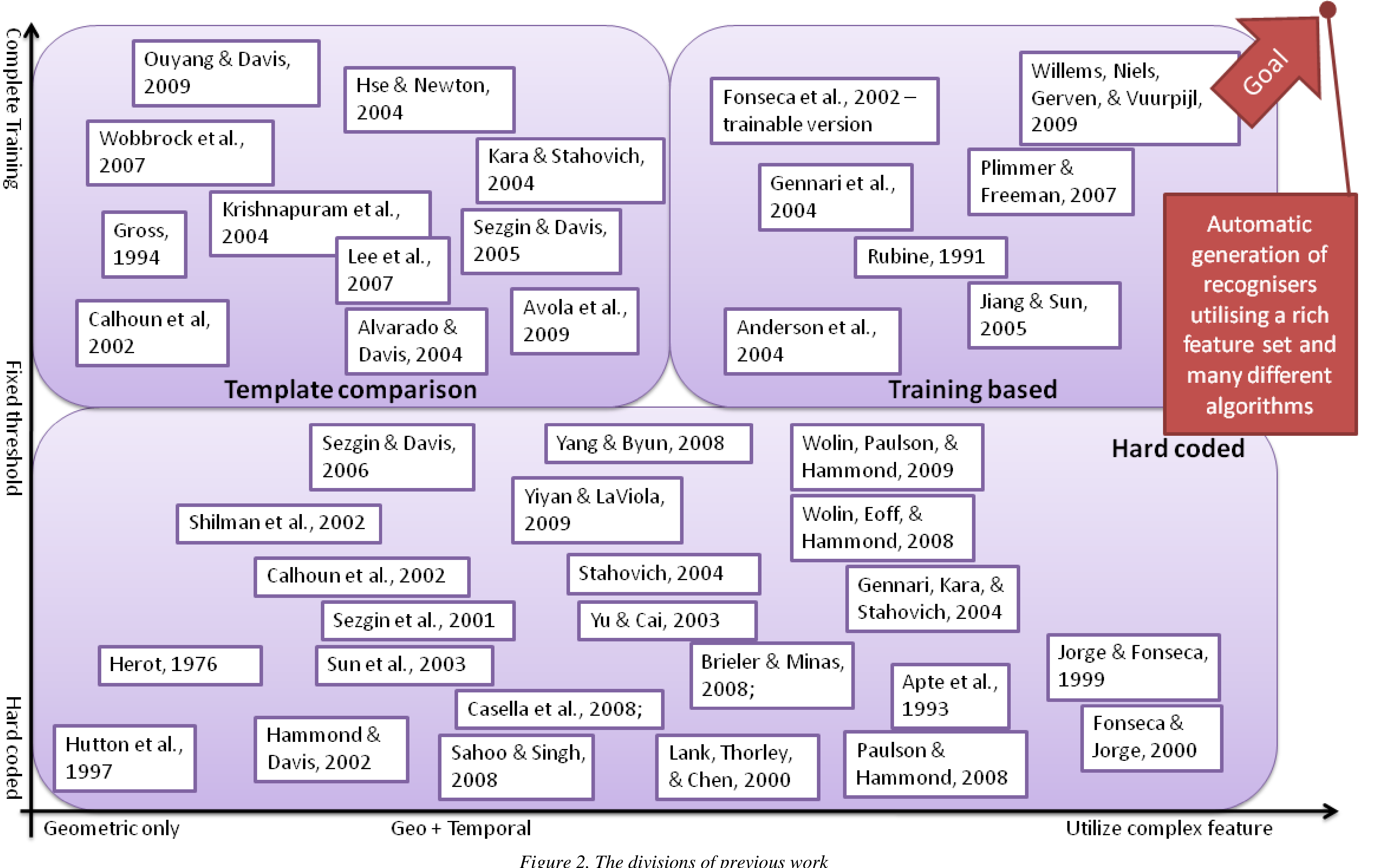


Figure 2. The divisions of previous work

2.1.1. Template Comparison

Regarding Figure 1, human cognition would instantly classify these shapes as rectangles with different rotations. One may notice “one shape is slightly longer than another, but they certainly have the same shape” while another may observe “they both have four right angles”. These two answers represent the two major approaches to template comparison: the first one is grid comparison, when the shapes are compared in grid space; the other is relationship comparison, when the geometric relationships of shapes are compared.

Grid Comparison

The observation that one shape is longer comes from aligning the two shapes and comparing their length, which requires rotation. The conclusion, “they are the same shape”, is decided by the similarity in their relative features, which is usually size independent. Recognisers with this approach normally divide a shape into a number of grids, reorient and resize the shape to match the grids, and compare whether or not the shape overlaps an existing template. As a simple example, Figure 3 shows how the shapes in Figure 1 would be represented in a basic grid comparison.

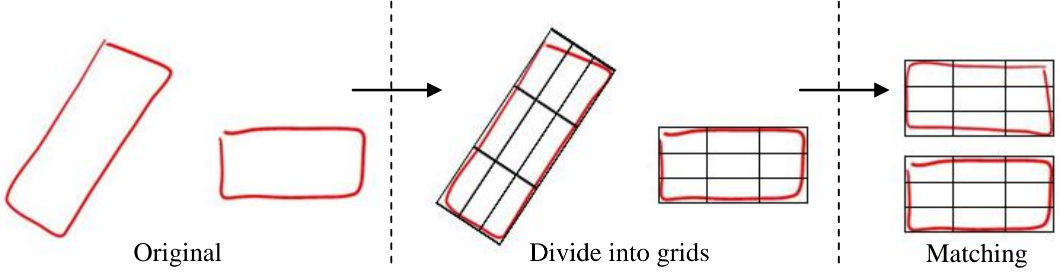


Figure 3. Example using Grid comparison

Wobbrock et al. (2007) take this approach with their $1 recogniser. A user specifies a set of templates where each will be compared with the input stroke. They calculate the distance between each part of the shape to judge how well it overlaps the templates. The best matching template will be the final classification. However, distance needs be calculated with points, hence certain mechanisms need to be applied to ensure the comparison is done on corresponding points. Additionally, directly overlapping the rectangles in Figure 1 would not make much sense until they are rotated and resized as shown in Figure 3. In the $1 recogniser, the process before recognition is separated into four steps: re-sampling, rotation, scaling and translation.

A digital stroke contains a series of coordinates with temporal data. Since the stroke sampling rate is constant, if a stroke is drawn twice as fast it will contain only half the points, as shown in Figure 4a. Different number of points which sits at different positions are not comparable. A re-sampling step addresses this problem by transferring the original points in the stroke to a constant number of equidistantly spaced points that still lie within the stroke.

Shapes are then rotated to ensure they can be compared. To speed the process, the rotation is done by making the line connecting the starting point and the centroid of the gesture horizontal, as shown in Figure 4b. Such implementation simplifies the process but introduces a constraint that shapes must be drawn with the same gesture. If a triangle template is drawn clockwise, a new triangle needs be drawn clockwise to be recognised. Shapes are then scaled non-uniformly to a reference square in order to remove the size effect. The translation step moves the shape to a reference point to facilitate comparison.

Both the template and the unknown shape undergo all four stages of the pre-process. After the pre-process, they have the same number of points and size, and lie in the same position with the same orientation. As the rotation step only approximates the best angular alignment, a more detailed correction is done to find the optimal angle. Finally the difference in distance between the unknown shape and each template will be calculated.

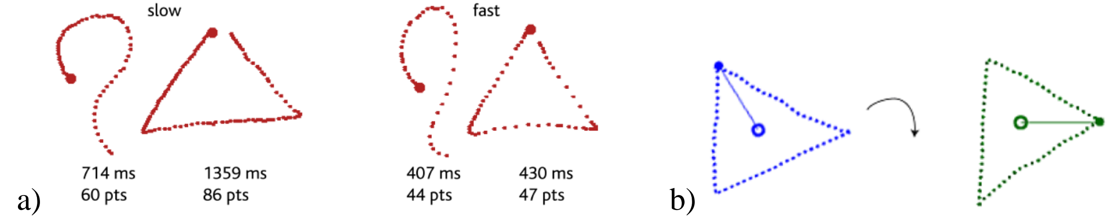


Figure 4. $1 processes(Wobbrock et al., 2007). (a) Difference in stroke times (b) Rotation of strokes

The $1 recogniser is fast and simple to implement, which demonstrates good results in experiments. However, the limitations make it suitable only for prototype usage. The recogniser cannot distinguish gestures which vary on specific orientation, aspect ratios or locations; for example, it cannot separate squares from rectangles. Also the lines can be recognised wrongly when it is drawn too short, as the recogniser may try to scale it uniformly which will cause it to lose the identity of being a line. Furthermore the input gesture needs be drawn in the same way as the example; otherwise the rotation will not complete its job.

Hse and Newton (2004) also apply scaling and translation to their data; however, instead of using points in the comparison, an input shape is scaled to form a 100x100 pixel model, which is then transferred into features with Zernike moments. Three different techniques are compared, including support vector machines (SVM), minimum mean distance and nearest neighbour. SVM is found to have the best performance.

Figure 5. Sketched symbol and their beautified version (Hse & Newton, 2005)

This approach not only recognises shapes, but also considers the direction and size, as shown in Figure 5, allowing beautification on top of sketched shapes. Although their work supports multi-stroke and is invariant to scaling, translation, rotation and reflection, it is a lazy system which recognition needs be triggered manually (Hse & Newton, 2005).

Krishnapuram, Bishop, and Szummer (2004) combine probabilistic models and affine transformation to match a shape with specified templates after the scaling and rotation. A Bayesian model is used to fragment a diagram into the most possible combination of subsets, where each subset representing a shape. As the system does not only return one classification result but returns the probabilities for each template, it can be combined with other frameworks.

There are also applications of off-line recognition, such as that done by Kara and Stahovich (2004). Each input symbol is firstly described as a 24x24 quantised bitmap as shown in Figure 6. They are compared spatially with the template symbols through four off-line classifiers each with different strengths. Each classifier returns a list of ranked results according to the similarity of the symbol to the templates, among which the average highest will be taken as the final classification. This voting system can reduce the chance of misclassification, because for most classifiers even if the top suggestion is incorrect, the correct result usually have higher rankings than the rest.

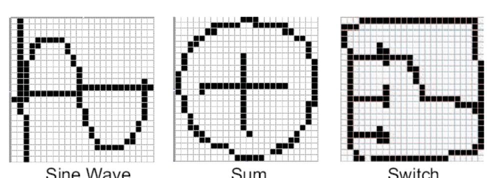


Figure 6. Examples of symbol templates(Kara & Stahovich, 2004)

Furthermore, Ouyang and Davis (2009) found evidence that current off-line recognition performs well, and thus utilised the ink information as feature images. The process is shown in Figure 7.

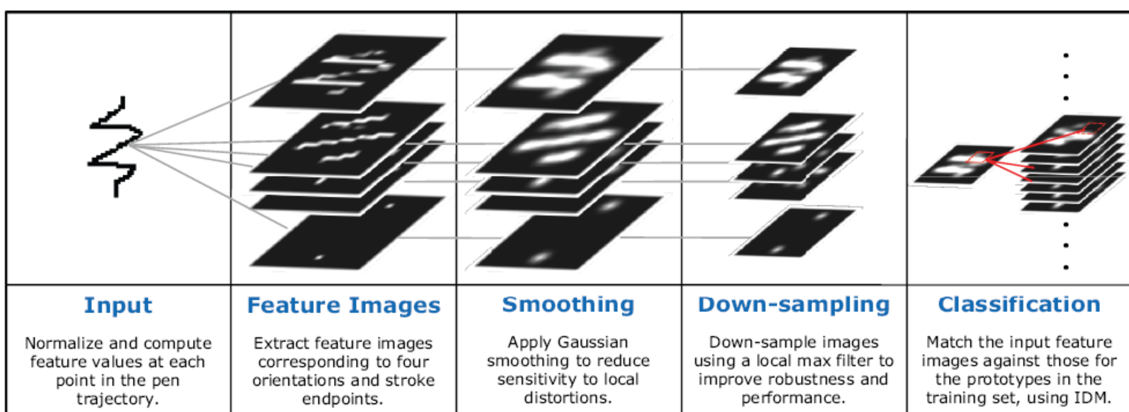


Figure 7. The system overview (Ouyang & Davis, 2009)

Their first step is to remove the difference caused by drawing speed, size and orientation, similar to Wobbrock et al. (2007). Five feature images are then generated, each considers a different feature. Four are orientation features that measure the orientations of each sample point, each corresponding to a different angle. One end point feature considers the beginning and end points. Each feature set is rendered into a 24x24 feature grid, smoothed and down-sampled to resist distortions. These feature images are then compared with the training samples with off-line recognition technique. The slow comparison process is optimised by having two comparison modes: the “coarse” mode which removes the bad candidates and the “exact” mode which does more detailed matching.

Instead of directly comparing the spatial distance, Gross (1994) use a raw glyph parser. A shape is sent into the parser which applies a 3x3 grid (which provides the optimal result) to the bounding box of the shape. Recognition is done by matching the sequence which the line crosses the grids, as shown in Figure 8. The number of shape corners is also recorded, to resolve the ambiguities of shapes with the same sequence but different corners, such as a circle (no corner) and a square (three corners) (Gross & Do, 1996).

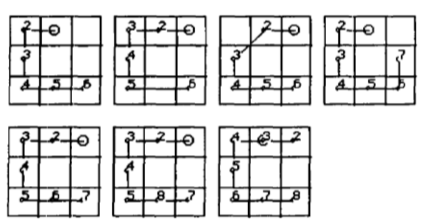


Figure 8. Training samples for letter C (Gross, 1994)

If multiple sequences match the unknown glyph, a second low level recognition is applied utilising features such as number of strokes, corners, sizes, aspect ratios and rotations. If no match is found, the program will relax the criteria. Users are allowed to resolve ambiguity and to name unknown shapes; as the program relies heavily on the sequences stored, the corrections will be added as a new example. Such user adaption can improve its performance.

Relationship Comparison

The recognised results of Gross and Do (1996) are put into a higher level recogniser named "configuration"; "A configuration is a set of elements (glyphs or configurations) of certain types arranged in certain spatial relations" (Gross & Do, 1996). The program is able to generate configuration settings automatically, and allows users to make modifications on those settings. It performs recognition by finding the template configuration which matches the input.

This "configuration" has a different nature from the grid based approaches. While it still compares the input with given templates, instead of comparing points or pixels, it compares the relationship of elements. We put this kind of approach into a sub category called relationship comparison approach, as mentioned at the beginning of this section. While these configurations seem to be similar to some expert systems we are to discuss in the next section, instead of being hardcoded, these relationships can be trained from examples.

Implementations with this approach commonly contain two levels of recognition. The lower level is the segmentation (usually hard coded) which divides the input graph into primitives such as lines and arcs; and the higher level is the actual comparison in relationship which makes the classification on shape types. For example, Calhoun, Stahovich, Kurtoglu, and Kara (2002) have used the same mechanism. After the lower level recognition, based on the training examples of a class, the symbol recogniser generates a semantic network, consisting of primitive as nodes and relationship as links. A node is labelled with its type (line/arc), length, relative length and its slope/radius. A link is labelled with the existence of intersection, relative location of intersection, angle between intersecting lines and existence of parallel lines. Absolute distances are described with pixels to cope with situations where size matters, while relative distances consider the proportion of all strokes belonging to the symbol.

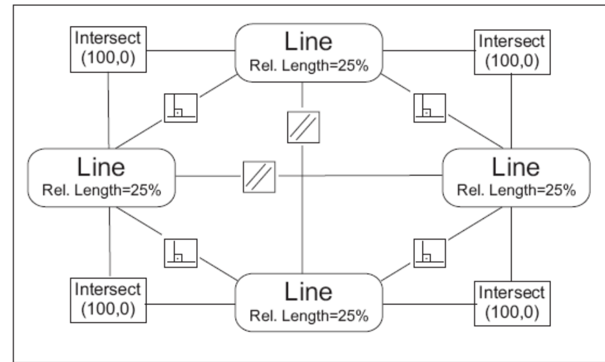


Figure 9. The semantic network of a square (Calhoun et al., 2002)

Training is done by examining symbols to identify the frequently occurring properties and relationships. Thresholds are placed to filter out noise. For example, intersections have a threshold of 70%, which means the attribute is only included if at least 70% of the training examples have primitives intersecting at the same position. Two recognising methods are implemented. One assumes shapes are always drawn with the same number and types of primitives in the same order, which is fast but unintuitive to use. The other allows more flexibility, which performs a best-first search with a speculative quality metric and pruning. This model is extended by Lee, Kara, and Stahovich (2007), in which the network description is refined as an attributed relational graph, with more information encapsulated. Similarities are measured with more detailed matrices and five search methods were included, each with different strengths.

Sezgin and Davis (2005) finds that different people draw with different orders, but although there can be many different orders in drawing one shape, only a few orders are preferred across people. Furthermore, drawing order is highly stylised; one tends to perform the same drawing order when sketching the same shape. They propose a hidden Markov model (HMM) approach based on geometric directions. By utilising a previous low level recogniser (Sezgin et al., 2001), the algorithm converts a stroke into 13 symbols, four for lines including positively/negatively sloped and horizontal/vertical, three for ovals including circle and horizontal/vertical ovals, four for polylines including 2, 3, 4, 5+ edges, one for complex approximations which indicates a mixture of curves and lines, and one to denote two consecutive intersecting strokes.

The intention is simple. Assume we have four types of line: horizontal (H), vertical (V), positive (P) and negative (N), and assume they can all be correctly detected. The stop

symbol in Figure 10 can be detected as a sequence of [V, H, V, H], while the skip-audio-track can be detected as [V, P, N, V].

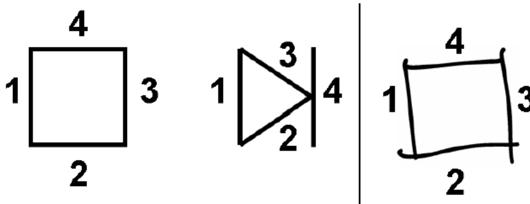


Figure 10. Demonstration of the mechanism (Sezgin & Davis, 2005)

Two approaches are used to encode the primitives into HMMs, the fixed input length HMMs and the variable length HMMs. The fixed input length training divides the examples into partitions where each partition contains only the example for the same shape with the same length. The HMMs are trained with the Baum-Welch method. Each class will have multiple HMMs each with varying length.

The fixed length makes it easy to build the HMM graph, as the destination can be easily computed. However two drawbacks appear. First the total number of training examples per model is reduced, because only the same shape class drawn with the same number of strokes are considered as the same. Second, even though two starting strokes are drawn similarly, if they are parts of different orders, they will not be presented together, thus reducing the recognition accuracy. These problems are avoided by applying the variable length training, which partitions the data only based on their class. Each class has its own HMM. Their evaluation displays accuracies over 95%, and the variable length input model performs slightly better.

Because these approaches perform segmentation before the actual relationship comparison, errors which occur in segmentation can be propagated to higher level recognition, thus increasing the error rate. Although the problem can be prevented by scanning through all possible recognition results, it is too expensive to do so. As a solution, Alvarado and Davis (2004) attacked the recognition from both bottom-up and top-down directions.

The process is separated into three steps. First, the bottom-up step is done by firstly applying a low level recogniser (Sezgin et al., 2001) to parse the input strokes into primitive objects, and then hypothesising the compound shapes even when the required elements are not drawn. Second, the top-down step attempts to find missing sub-shapes

from the partial interpretations generated from the bottom-up step by refining the wrongly interpreted shapes. Finally a pruning step is applied as a final stage to remove unlikely interpretations. The dynamic Bayesian network method is applied in button-up step to capture the interactive and incremental process (Sezgin & Davis, 2007). The template is built with descriptions and searched with Bayesian networks.

Instead of performing these checks, Avola, Buono, Gianforme, Paolozzi, and Wang (2009) add another layer of segmentation, apart from the traditional line and arcs, which finds the occurrence of closed region and poly-line. This approach provides them with an extra level of detail to prevent misclassification, which returns high recognition accuracy.

Overall, template matching methods are natural, and many supports can be taken from off-line recognition. A great advantage is the simplicity in extending the shapes to support, because they only rely on examples (Kara & Stahovich, 2005). However, while grid comparison methods require certain transformations which cause the loss of property, relationship comparison methods are hard to automate and their performance depends on the lower level recognition and segmentation. Furthermore, neither of them deeply explored the rich information provided by ink data. Compared with off-line recognition where only pixel data is available, this information may be the key to increasing the accuracy, since "no aspect of the sketched symbol may be safely ignored" (Johnson et al., 2009).

2.1.2. Hard Coded

Template comparison examines the input shapes with the pre-defined matrices or algorithms. However, there are situations where relationships which are easily understood by humans may be complex for computers to learn. For example, a diamond is simply a rotated square. To ensure these relationships are correctly learned, users need to provide datasets which allow the algorithm to detect the relationship. On the other hand, if a shape is recognised as a rectangle and we know it is rotated, heuristically we know it is a diamond. Such heuristics can be easily programmed into a recogniser with an IF statement, which is why many recognisers are implemented hard coded.

This section is separated into segmentation and expert systems. The mechanisms are similar, but segmentation aims to find low level details which provide the base for many other recognisers, while the expert systems are more complete systems.

Segmentation

Segmentation attempts to separate an input stroke into elements called primitives, each standing for the most basic element such as a line or an arc (Herot, 1976; Sun, Zhang, Qiu, & Zhang, 2003). Whether a stroke is drawn in single stroke or more, it would be divided into primitives. Most multi-stroke recognisers have applied segmentation as a base and used other techniques to further group and reason the primitives to the desired classifications. Its nature of filtering out the noises also makes it very suitable for beautification problems.

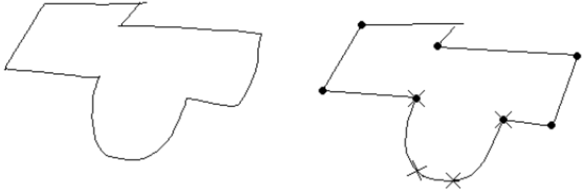


Figure 11. An example of segmentation (Sezgin et al., 2001)

Sezgin et al. (2001) detects feature points using stroke direction, curvature and speed data, by finding where the extreme numbers lie. The primitives are the paths between these feature points, as shown in Figure 11. To remove the false positive generated by noise, they firstly apply the average based filtering, which averages the values to generate a threshold and use it to filter out the unimportant values. The curvature data and speed data are then combined to form the hybrid fit, to detect vertices. Euclidean distance is calculated to decide if the section is a line or a curve. A curve will be approximated with Bezier curve. These beautification processes make the graph look more like what the user intended. The basic recognition is done in a manner similar to relationship comparison, but hard coded; for example, “polyline with 4 segments all of whose vertices are within a specified distance of the centre of the figure’s bounding box” (Sezgin et al., 2001) will be recognised as a rectangle.

A similar approach is applied by Calhoun et al. (2002). Speed and curvature data are utilised. Average based filtering is also applied, but users have the freedom to change the threshold. Curvature finding is done by putting a “window” shown in Figure 12 which covers several points. They connect the two outermost points in the window, and find the difference of each point in-between the line, with consideration on which side the points lay. If the absolute sum of the distance is less than a certain threshold, the curvature is considered to be zero. The most suitable size of window depends on the device and user.

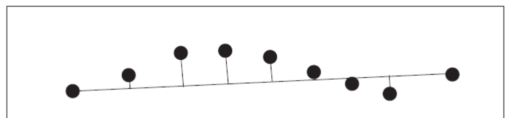


Figure 12. Calculating the curvature sign. The window includes 9 points(Calhoun et al., 2002)

Yu and Cai (2003) propose a domain independent approach to recognise smooth curves, hybrid shapes and polylines. They declare several requirements for an ink shape recognition system: it should allow natural interaction as with paper; it should cope with multi-stroke symbols; it should understand the hierarchical composition; it should try to predict the thinking of the user; it should be easy to use and it should be easily integrated into other systems.

The recognition is performed in two steps. The first step, imprecise stroke approximation, takes an input stroke and returns it as one or multiple primitive shapes. It is done by approximating the stroke with the pre-specified primitive shapes, and if it fails, they find the position where the maximum curvature change occurs, and perform the approximation again. The second step, post process, is triggered when the whole diagram is finished. The primitives recognised by imprecise stroke approximation will be further processed, to be presented as intended by the user.

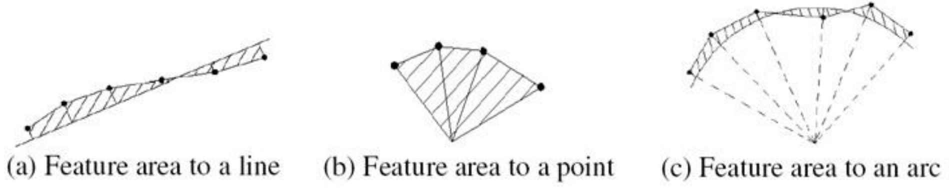


Figure 13. Feature area examples(Yu & Cai, 2003)

One important feature they use throughout this research is the “feature area” of a stroke, as shown in Figure 13. It is the area between the stroke and a reference object which can be a line, a curve or a point. The feature area to a line or an arc is calculated as the total area of all quadrangles formed by two consecutive stroke points and their foot points on the line. And the feature area to a point is calculated by the sum area of the triangles formed by two consecutive stroke points and the reference point.

Segmentation approaches involving feature point detection usually require a fixed threshold value to filter the noise (Calhoun et al., 2002; Sezgin et al., 2001; Stahovich, 2004), which may be too restrictive in real world applications. To improve the situation, Sezgin and Davis (2006) apply scale-space theory, implementing an algorithm which can find feature points without specifying a threshold. Furthermore Yang and Byun (2008)

propose a robust feature extraction algorithm which not only extracts the feature points but also reduces noise and eliminates hooks. As for the wrongly detected feature points, ShortStraw can examine their validity by checking the slope of neighbourhood points (Wolin, Eoff, & Hammond, 2008; Yiyan & LaViola, 2009), and use the multi pass approach to combine them with neighbours (Wolin, Paulson, & Hammond, 2009). Wolin et al. (2009) claim that ShortStraw is the most promising approach for corner finding.

Locating individual symbols in a multi-stroke situation is always challenging, even if segmentation is applied. Gennari, Kara, and Stahovich (2004) propose an approach with two steps: first by utilising the high ink density area and second by identifying points where characteristics of pen strokes change. The ink density is defined by:

$$density = \frac{ink_length^2}{oriented_bounding_box_area}.$$

Symbols usually have higher density, because the strokes are closer to each other. It is checked by a forward-backward algorithm. Looking at the previous and next strokes, if the addition of a stroke reduces the density below a certain threshold (which is determined empirically), the added stroke likely does not belong to the symbol. It does not matter if the stroke drawn after the finish of a symbol is a connecting line or the start of another symbol some distance apart, they both reduce the density. The process is demonstrated in Figure 14.

Ink	Start segment	End segment	Density	Density change (%)
	5	6	9.5	
	5	7	17.8	87.4
	5	8	16.9	−5.1
	5	9	10.9	−35.5
	5	10	12.0	10.1

Figure 14. the forward step when finding a possible end segment for a symbol(Gennari et al., 2004)

To reduce the complexity of parsing, users are restricted in finishing one symbol before drawing another. Once the candidates are enumerated using the two features, domain-

specific knowledge is used to remove candidates which are unlikely to be symbols. Finally a general symbol recogniser is applied which recognises common shape types, and the resulting recognitions are sent into a sketch interpreter with domain knowledge.

Four characteristics are compared to analyse the change of behaviour, which are: Type (line/arc), Length, Orientation (the angle between two segments) and Interaction type (end to end, end to midpoint or midpoint to midpoint). Each comparison is hardcoded.

Expert Systems

The techniques in this section are called expert systems because they are coded in such a way that each shape is handled by a different configuration. Apte, Vo, and Kimura (1993) presented a recogniser which is able to recognise six shape classes including rectangles, ellipses, circles, diamonds, triangles and lines. Multi-stroke symbols are supported, as long as the strokes of the symbol are drawn without a pause, because the recognition is triggered by a time-out event. This assumes that users can only (and must) pause when one symbol is finished and another is to be started.

The algorithm contains three different filters, which are applied to different shapes in different combinations, as shown in Table 1. Once a series of strokes are drawn and the drawing process is paused over a certain time threshold, these strokes are sent into the recogniser, which applies the three filters as a tree structure. The recogniser reports 97.5% correctness. However, it would not cope with rotation of shapes, and a shape must be drawn without pause.

Table 1. Filters used by Apte et al. (1993)

Name	Method	Where to apply
Area-Ratio Filter	Area of convex hull / area of bounding rectangle. Triangle and diamond will have about 50% while rectangle close to 100% and Ellipse about 80%.	Triangle, Diamond, Rectangle, Ellipse
Triangle-Diamond Filter	The relation on the left and right corners are checked, as corners of triangle are near the end while corners of diamond are near the middle.	Triangle, Diamond
P^2/A Ratio Filter	Perimeter^2/Area, the equation for each shape is calculated to cope with size variation	Rectangle, Ellipse, Circle, Line

Fonseca and Jorge (2000) extend the system to cope with shapes of different size and rotation, even shapes drawn with dashed strokes or overlapping lines. The tool calculates a number of features based on three special polygons, which are: the largest area triangle within the convex hull, the largest area quadrilateral within the convex hull and the smallest area enclosing rectangle of the convex hull. They are shown in Figure 15.

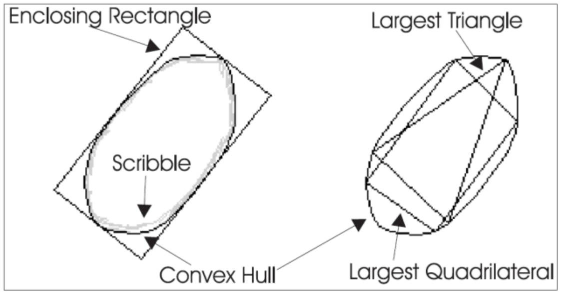


Figure 15. CALI polygons used to estimate feature (Fonseca & Jorge, 2000)

After the calculation, fuzzy logic is applied to handle noises. The fuzzy sets are deduced from training data (Jorge & Fonseca, 1999), which makes the classifier more adaptive to badly formed shapes, but still does not allow new shape classes to be added. Recognition is done in a rule based manner, features are extracted from strokes and a series of IF-ELSE statements are applied to obtain the result. The features used can be found in (Patel, 2007).

Lank, Thorley, and Chen (2000) present a UML sketch system. When a stroke is fed into the system, it is passed to a retargetable segmenter, which groups multiple strokes into a glyph if overlapping occurs, because it is an indication they belong to the same shape. A filter then transforms the group into the required data format, and sends it to the appropriate recogniser. The recogniser will be domain specific, and in the particular implementation in the paper, the authors build a UML recogniser.

The UML recogniser firstly makes a histogram out of the size of glyph, and then uses the size as information to perform intelligent pruning. For example, characters are normally smaller than class box in a UML diagram. Almost all the recognitions are integrated with experience or heuristic, applying information such as the number of strokes contained in the glyph, distance metrics, total stroke length and bounding box. Because the non-arrow shapes are easier to recognise, a special test is applied to distinguish the different arrow types, including open arrow head, closed arrow head and diamond. After the symbols are identified, refinement based on domain knowledge will be applied to correct the mistakes made by the retargetable segmenter. Overall intensive heuristic knowledge is applied, which is very specific to the UML domain.

With a Rubine (1991) based classifier (which will be discussed in the next section) for low level primitives, Shilman et al. (2002) apply visual constraints (including distance, angle and ratio) for higher level recognition; the threshold for these constraints are hard coded, expressed as Gaussian distributions. This approach is similar to the relationship

comparison method described in section 2.1.1. In fact, many hard coded approaches try to recognise shapes in a bottom-up manner: input graphs are segmented first, and domain knowledge is applied to resolve the ambiguity and perform higher level recognition. (Brieler & Minas, 2008; Casella, Deufemia, Mascardi, Costagliola, & Martelli, 2008; Sahoo & Singh, 2008). Paulson and Hammond (Paulson & Hammond, 2008) attempt to improve this method by, instead of only recognising arcs and lines, building a low level recogniser which can recognise eight different shapes. The selected shapes include line, polyline, circle, ellipse, arc, curve, spiral and helix. Input stroke undergoes the eight primitive tests, each returning a likeliness value. These values are then combined by the hierarchy step which unites the result and makes the final judgement.

A multilayer approach is presented by Hamond and Davis (2002). It uses attributes such as angles, slopes and other properties. A pre-processing step is applied by utilising the previous recogniser (Sezgin et al., 2001), which fits each stroke into either an ellipse, a line, a polyline (consisting of multiple lines) or a complex shape (consists of lines and curves). It then tries to form a stroke collection that is considered as a recognisable object. The division mainly applies the geometric property, by calculating the distance between each object. The process then enters the recognition step, in which stroke collections are examined by hardcoded recognisers. Eight object recognisers are presented for UML shapes, each acting as an expert analysing different aspects, which is similar to the approach taken by Hutton, Cripps, Elliman, and Higgins (1997). If more than one possible candidate is found, the situation will be resolved in the next stage, identification, which makes decision based on a hard coded priority list.

The hard coded approaches are developed so rich features provided by digital ink stroke can be used with heuristic applied. However because of their hardcoded nature, they are more complex to maintain or extend (Johnson et al., 2009). This makes them very cost ineffective. Although attempts have been made to address the problem by trying to support a large number of basic shapes (Fonseca, Pimentel, & Jorge, 2002), as described in section 1.1, such an approach is limited. This is because the more shapes included, the more ambiguity occurs (Schmieder, 2009). For example, if a recogniser recognises square, rectangle and diamond, in a diagram containing only rectangle and diamond, some rectangles and diamonds would probably be classified as square, a situation in which loss of accuracy would occur despite how the user interpreted the square result. Furthermore, although the heuristic based approach is easy to reason and program, it cannot deal with

the hidden relationships between shapes. A method must be selected which enables the use of different features and allows the automatic training.

2.1.3. Training Based

Many types of information can be retrieved from a digital ink stroke (at least enough for humans to distinguish between them), but unlike the number of corners which humans can easily decide between, most features are either unobvious or hidden. For example, the feature "vertical movement / height of the bounding box" (Fonseca et al., 2002) can be considered unobvious since they cannot be readily described by human cognition. Alternatively, the time taken to draw a stroke is completely hidden because it is not visually recognisable. These are the features that distinguish on-line and off-line recognition.

Certainly we can export those types of information as numeric data and plot graphs to help us understand and deduce rules, such as is done in CALI (Fonseca et al., 2002). However, such methods cannot detect cross relationship; to take the simplest case, a triangle rotated 45 degrees is still a triangle, but a square rotated 45 degrees would be a diamond. More complex versions of such cross relationship, if not heuristic, would take much effort to be distinguished from numerous graphs, even if we only considered ten different features and consider relationships between two features each time. And there is certainly a possibility that cross relationship can occur between three or more features.

Furthermore, it is impossible to plot graphs for all the information within a stroke. A stroke basically contains spatial, temporal and pressure information. This information can form an infinite number of features. For example, we may deduce the average velocity as a new feature; however it is also possible that the square root of the attribute will have a more direct relationship to the accuracy. A more concrete example can be found in (Willems, Niels, Gerven, & Vuurpijl, 2009) where they discovered that the standard deviation and mean of features contribute to the recognition. With this enormous amount of information, processing manually lacks efficiency and accuracy. One promising approach is machine learning (Hammond et al., 2008). Many machine learning algorithms are developed and successfully applied in different areas, which utilises computers' ability to perform routine works to find the relationships between features and targeting classification. In this section we will discuss several applications in stroke recognition.

Rubine has implemented an algorithm based on the classic linear discriminator (Rubine, 1991). It extracts 13 numeric features from each stroke, and uses them to train a statistical model. A trained model contains a number of gesture classes, and each class has a linear evaluation function over input features. When a stroke is given, their features are calculated and passed to each class within the model. Different classes have different evaluation functions, which will result in different calculated values. The class which returns the maximum value would be the target class.

Many researchers such as Shilman (2002), Landay (1995) and Plimmer and Freeman (2007) apply Rubine with modifications. Extending the supported shape classes is easy. All that needs to be done is to provide the training examples for the new class. The algorithm can also be upgraded with new features, since there is no guarantee the original Rubine features are the best performing ones. Evidence has shown that the features Rubine applies are not the optimal combination (Choi, Paulson, & Hammond, 2008). Rubine also attempt to analyse the relationships between accuracy, training sample and number of shape classes, as shown in Figure 16.

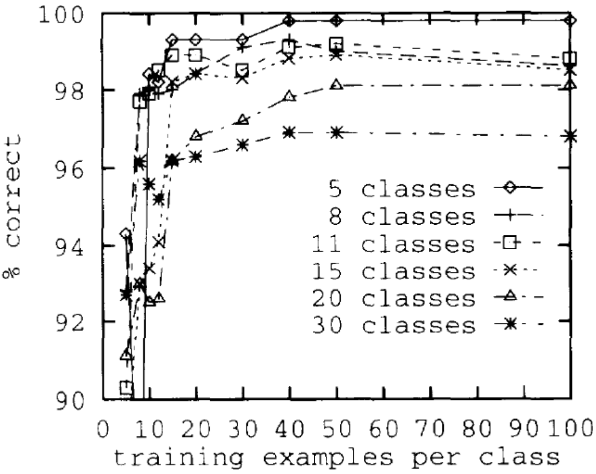


Figure 16. Recognition rate vs. training set size (Rubine, 1991)

This figure illustrates the fact that increasing the number of shape classes reduces the accuracy, whereas increasing the number of training examples increases the accuracy. However, only a small amount of training examples per class is required to achieve optimal accuracy, where further increment in training examples will have minimal effect. The same observation is reported by Fonseca et al. (2002).

CALI contains many features but cannot support new shapes easily due to its hard coded nature. To allow more flexibility, Fonseca et al. (2002) apply machine learning techniques

to build a trainable version of it, where they test three machine learning algorithms including KNN, ID3 and Naïve Bayes. Results show that Naïve Bayes is fastest in both training and recognising. The Naïve Bayes algorithm computes the probability of each feature value to each class, and combines the probabilities to identify the most likely class. It assumes there are no statistical dependencies between each feature; although the assumption is generally not true, the algorithm can still provide acceptable accuracy with 50 examples per class and stabilise to a good performance at around 100 examples per class. Naïve Bayes algorithm is also used by Gennari et al. (2004) with nine features and the same assumption, which also returned with good results. Once a symbol is recognised, domain knowledge is applied to decide if the interpretation of the symbol is consistent with the rest of the diagram; the recognition will be revised if it is inconsistent.

HMM is widely applied in speech recognition, which also found its place in sketch recognition. Jiang and Sun (2005) and Anderson, Bailey, and Skubic (2004) all apply this technique, with different focuses.

Instead of using the primitives to build HMMs such as done by Sezgin and Davis (2005), Jiang and Sun (2005) apply feature based HMM training. They use first-order left to right HMM, believing "the position and structure of current stroke is usually dependent on the previous stroke" (Jiang & Sun, 2005). A single HMM state is used to represent one stroke, and the HMM models are trained from features with Viterbi decoding. Recognition will utilise the features to get probabilities by using the trained HMM and return the rankings of each class to users. The features used contain both spatial (geographic) and temporal (dynamic) data. Their recognition delivers over 95% recognition accuracy.

Training based approaches can be combined with other methods. Anderson et al. (2004) suggest a system based on HMM. The system can be trained automatically, but the classification has high false alarm rates which tend to return the correct answer as the second most likely. Thus they are forced to implement a post process which use heuristics to reject misclassified symbols. Although this implementation becomes a hybrid system, the training based HMM still reduces the amount of work required in comparison to hard coded approaches.

For training based algorithms, the quality of features is as important as the algorithm which utilises them. Research in sketch recognition forms a good selection of features, which can be united for further use (Blagojevic, Schiemder, & Plimmer, 2009). The

previous research do not utilise a large number of features. However, more and more researchers have noticed the opportunities (Blagojevic, Plimmer, Grundy, & Wang, 2008b; Choi et al., 2008). Willems et al. (2009) conduct a study to evaluate the effect of different feature sets. Three groups of feature are used: one has 48 features from the previous research which contains geometric features; another extends those 48 features by finding the mean and standard deviation to generate a total of 758 features; the last contains features used extensively in character recognition. They report the extended feature set improves the accuracy between 10% and 50% compared with only geometric features; and between 25% and 39% compared with two previous recognition techniques. This demonstrates that the good application of features can produce improvements in the recognition accuracy.

Although many training approaches exist, very little applied a large number of features, and there is no guarantee any of the machine learning algorithms used are the best because most research focuses only on one algorithm. This approach is promising, yet not fully explored.

2.1.4. Discussion

Accuracy is the most important attribute of a sketch recogniser. However, it normally comes with domain knowledge. Evidence has shown that a recogniser may not perform as well if it is not used for the exact problem it is designed for (Schmieder et al., 2009). Building a recogniser which supports every diagram is impractical, however as there are many diagrams, making the recogniser extensible is very important. The hard coded recognisers fail with this aspect. Although template comparison based recognisers can be trained, most of their implementations do not utilise much information from the input strokes.

The training based recognisers take advantages from both areas. They can be trained with many features. However little research explores the application of large feature sets and no comparison exists to judge the performance of the algorithms applied. Applying rich feature sets and exploring different algorithms may improve the accuracy of existing research.

2.2. Data Mining in Related Domains

Data mining is the technique used in finding patterns in huge volumes of data. There are several applications in digital ink shape recognition, as explained in Section 2.1.3; but as described, there are still limitations. This section reviews the application of data mining in two related areas – the ink character recognition and the speech recognition. They are similar to digital ink shape recognition in various aspects, and their success should aid this research.

2.2.1. Ink Character Recognition

Eager character recognition is considered, it can be viewed as the subset of sketch interpretation (Johnson et al., 2009). It is similar to ink shape recognition because both take ink strokes as input.

Smithies, Novins, and Arvo (1999) utilise nearest-neighbour classification, and describe their classifier as being similar to Rubine, but with a feature space of approximately 50 dimensions. Each user needs to input 10 ~ 20 training examples; although it seems to be user dependent, they find the trained classifier is user independent. Bahlmann, Haasdonk, and Burkhardt (2002) apply dynamic time warping technique as a kernel to SVM, and the result is competitive with an HMM-based classifier. LaViola and Zeleznik (2004) use three classifiers to do different tasks including pre-processing, tuning and fine classification; which demonstrates a good way of combining different classifiers. Tay (2008) compares three data mining techniques: SVM, C4.5 and Naïve Bayes to interpret ink characters by extracting 17 features from them and report that SVM demonstrates the best accuracy. The same recommendation is given by Do and Artières (2009). WEKA is applied in research on text-shape dividing (Waranusast, Haddawy, & Dailey, 2009), where SVM shows promising results; however, while their approach depends on temporal data, their data was not collected directly from different people but copied by the same people, which reduces the validity of their evaluation.

Tappert (1982) applies template matching to match the similarity between the training characters and the written ones. A set of pre-processes are performed to remove the noises, including character separation, hook removal and to establish data points in roughly equal distances. Two attributes, the slope angle and the height of each point, are then calculated and are used for the comparison. Connell and Jain (2001) post a similar approach;

however they not only use difference in distance but also apply nearest neighbour and decision tree to learn the relationships. This method is extended for signature detection and obtains promising results (Jain, Griess, & Connell, 2002).

Connell, Sinha, and Jain (2000) use five classifiers together to classify the input strokes. Two HMMs are applied for local on-line features while three nearest neighbour algorithms are applied to off line features. Contributions of each are decided through their error characteristics. The final accuracy (86.5% for correct answer being the top choice, or 96.8% for correct answer falling in the top five) is much higher than the accuracy of individual classifiers (ranging from 43.1% to 77.4%), which proves the advantage of combining different classifiers. A similar combination process is applied by Alimoglu and Alpaydin (2001) who find combining multiple instances of multilayer perceptron classifier can improve the accuracy.

Approaches in character recognition can be applied in digital ink shape recognition, but normally they can be further optimised (Willems et al., 2009). This is because although great similarities exist between character recognition and shape recognition, they are still different. First, people usually finish one word before starting another (Smithies, Novins, & Arvo, 2001), where it is more flexible when plotting shapes. Second, characters normally flow either horizontally or vertically, where shapes can occur anywhere without considering the shape drawn previously. Third, a character is usually written with similar sequences, where shapes have more variation. These differences make character recognition a different problem from shape recognition.

2.2.2. Speech Recognition

Generally speech recognition involves transferring a sequence of waveforms into text they represent. In MPEG-7 Standard (Manjunath, Salembier, & Sikora, 2002), both speech and ink recognition are considered as multi-media input. People speak with different speed and pause, which is the same as variations in the drawing of diagrams – a stroke can come directly after another or with a huge time gap. The large variation between the speeches of individuals is similar to the variation between their drawing styles. Hence we believe the discoveries in the field can aid our research.

HMM is one of the most widely applied methods in speech recognition, due to its statistical framework which is able to model the sequential behaviour of speech

waveforms (Ganapathiraju, 1998; Juang & Rabiner, 1991; Rabiner, 1990; Young, 1996). There are also hybrid systems which further improve HMMs such as the combination with Neuron network (Rigoll & Neukirchen, 1996; Schwenk, 1999; Trentin & Gori, 2001) or SVM (Ganapathiraju, Hamaker, & Picone, 2004), which demonstrates the combination of different classifiers can improve the accuracy. SVM alone can be trained as the most generalised classifier in this domain (Chin, 1999), with the application of different features (Clarkson & Moreno, 1999; Klautau, Jevtic, & Orlitsky, 2002).

As special subtypes of sound recognition, the detection of emotion from speech or genre from music can be even closer to digital ink shape recognition. They have fewer class types compared to the number of words in speech, but each covers a wider range of varieties.

Emotion recognition attempts to extract emotion features from speech signals. Many data mining techniques are applied such as Neuron network, C4.5 tree, n-nearest neighbour (Kosina, 2002), Naïve Bayes and SVM, with feature dimensions from 37 (Yacoub, Simske, Lin, & Burns, 2003), 200 (Oudeyer, 2003) to 1280 (Vogt & Andre, 2005). Evidence shows that Neuron network and SVM outperformed other techniques (Yacoub et al., 2003).

Basili, Serafini, and Stellato (2004) apply six algorithms including Nearest neighbour like algorithms, with only five features to find the impact of the small feature set. Naïve Bayes is outperforming the rest, although all classifiers have such low performance that they consider repeating the experiment with more features. Although many classifiers are proven to be effective in the stated experiments, even with the same classifier, many factors can affect the performance, and should be taken into account during optimisation (Norowi, Doraisamy, & Wirza, 2005).

Research in speech recognition usually utilise large amounts of features. This contributes directly to their accuracy, as compared with experiments with only small numbers of features. On the other hand, many algorithms are applied, and the evidence supporting Bayesian Network and SVM is valuable to our study.

2.3. Software Used

This research is based on two software components. One is DataManager which is used for generating features, and the other is WEKA which is used for data mining. This section gives a brief review of them.

2.3.1. DataManager

DataManager (Blagojevic, Plimmer, Grundy, & Wang, 2008a) is a tool which can collect and label digital ink sketches. These labelled sketches can be converted into groups of features, where the current version of DataManager is capable for generating 119 features (Blagojevic, 2009). It also consists of an evaluator which can be used to compare the performance of different classifiers under the same settings (Schmieder et al., 2009).

At the start of this project, in comparison to other research, DataManager implements the largest number of features, including but not restricted to the ones from previous research. It is also extendable so new features can be easily added. Although at the end of this study the work of Willems et al. (2009) was published, we have not had an opportunity to utilise their features due to the time constraint. DataManager calculates features for each stroke, which includes features that consider only the stroke itself, as well as the temporal spatial context features (TSC-features), which are features that consider the temporal (the previous/next stroke) and spatial (the other strokes geographically around it) relationships. Although it is capable of generating such a rich collection of features, it cannot generate a recognisor based on these features, which limits the application of the features.

The evaluator (Schmieder et al., 2009) is also important to this study. To test the result of our implementation it should be compared with other classifiers. However, comparing the reported percentage accuracies is not meaningful if they are obtained from different data. The experiments need to be controlled, but classifiers are varied in the language used and the input/output interfaces, which increases the complexity to make comparison between them. The evaluator has integrated many existing classifiers and provides an interface which allows easy access and comparison between them.

2.3.2. WEKA

WEKA (Hall et al., 2009) is a data mining tool which is open source and has many data mining techniques implemented. Users can use these data mining techniques either

through a GUI, a CLI or directly calling the methods through a JAR provided. The tool is successfully applied in many data mining research projects mentioned in previous sections (Basili et al., 2004; Kosina, 2002; Norowi et al., 2005; Oudeyer, 2003; Vogt & Andre, 2005; Yacoub et al., 2003).

Each implemented algorithm can be modified by changing its settings, which offers certain flexibility. The tool is growing with new algorithms written by data mining professionals, with the most advanced techniques. Hence an obvious advantage of using WEKA is that the result of this research can be easily extended whenever WEKA is updated, and can utilise the knowledge from professionals directly, without the need of re-implementing their work. On the other hand, WEKA is developed to support different data mining problems. It is not specifically configured for ink shape recognition, hence tuning is required; while the tuning can be done via its interface, the flexibility offered increases its complexity, which requires time to learn. This is why we aim to find a way to merge it with DataManager.

2.4. Summary

We have surveyed the literature on digital ink recognition. It can be separated into three groups: template comparison, hard coded and training based. Among these approaches, the training based recognisers demonstrate both extensibility and good use of ink information; considering both accuracy and developing time, it is the most promising approach. However, it still has limitations. First, only a few algorithms have been applied which are not necessarily the optimal choice. Second, the features used are limited in both number and variety. The works in related domains are also examined, including character recognition and speech recognition. The successful application of data mining in these similar domains introduces many potential algorithms, and shows the possibility in applying data mining in ink shape recognition.

The tools this project is based on are also introduced. DataManager is capable of generating a rich feature set from a single stroke, and provides much functionality for collecting stroke data. However it cannot be used directly to produce recognisers. WEKA is an open-source data mining tool which supports many data mining algorithms, but can be too complex to use.

In conclusion, we believe the application of data mining with a rich feature set can resolve the limitations in training based approaches, therefore our goals include the application of different data mining techniques for WEKA with a qualitative large feature set generated by DataManager, the identification of the strengths of individual algorithm, and based on the observations to implement a recogniser generator which dynamically creates recogniser for input data. The following chapter outlines the methodology used to achieve these goals.

Chapter 3
Overview

In the previous chapter we found training based approaches is the most promising method for digital ink shape recognition, and decided to combine data mining techniques provided by WEKA (Hall et al., 2009) and the features from DataManager (Blagojevic, 2009) to allow the dynamic generation of recognisers. Such a combination is beneficial in several ways: rich features can be used; recognition can focus on the targeting diagram domain, and different algorithms can be selected, where each may have different performance toward different problems. Successful implementation of this can minimise the effort required to customise a good performing recogniser.

This chapter describes the methodology adopted to apply data mining to digital ink shape recognition and to utilise the knowledge gained to design and implement a recogniser generator. There are four steps to this process:

1. Data is needed for the data mining process; Section 3.1 describes the targeting data and the reasons they are chosen.
2. The WEKA Data mining tool that we are using in this project includes many algorithms, most of which can be tuned. Section 3.2 provides an overview on how algorithms are selected from WEKA and the approaches to optimise their performance. Furthermore we explore how data mining can improve performance with attribute selection and cooperation of algorithms.
3. Once the best algorithms have been tuned they can be used to generate recognisers. Section 3.3 describes the implementation of the recogniser generator.
4. Finally Section 3.4 explains how the generated recognisers will be evaluated against other recognisers.

3.1. Data Collection and Process

The first step of this study is to collect data to train algorithms. Instead of applying the most general form of data mining algorithms, we want to optimise them for the problem domain which is diagram recognition. Because the nature of sketched diagram is different

from speech or handwriting their success cannot be directly copied. Sketched diagrams are required for the optimisation. However, if the algorithms are optimised toward a single sub-domain such as UML class diagram, it can be biased to the sub-domain, which can reduce the effectiveness toward other sub-domains. Hence, instead of optimising the algorithms for a specific sub-domain such as ER or UML diagram, we have to optimise these algorithms for the whole diagram domain.

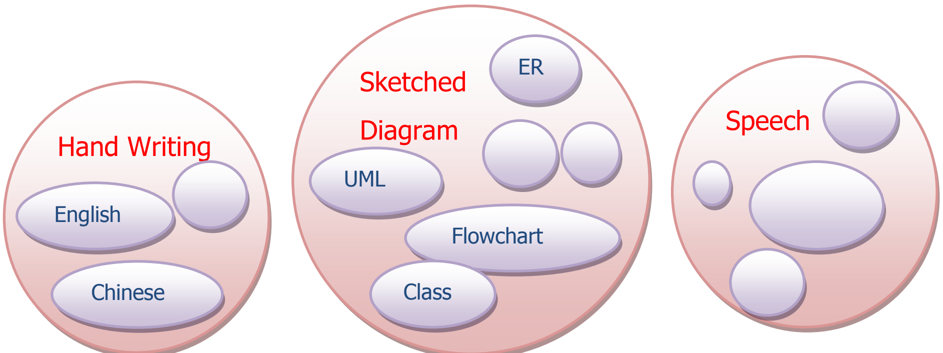


Figure 17. Domains and their sub-domains

Yet, there is no way to collect data which represents the complete hypothesis space for diagram domain, because it includes an infinite number of different diagram types and shapes. Instead we decided to carefully select multiple types of diagram to capture different perspectives of the diagram domain. The collected diagrams must be real diagrams which are used in the real world, and have different characteristics and complexity so they can reveal the impact these differences may bring to the data mining algorithms. Additionally they must fit our requirements, in particular that all shapes that are contained in the diagram can be drawn with a single stroke. We believe if the different datasets show similar behaviour toward the same optimisation, such optimisation can be applied to other domains too.

Once the targeted diagram types are determined, they need to be collected. Data collection involves not only finding participants to draw shapes but also the time required to label them for training. Two methods exist for collecting data; they can either be collected as complete diagrams (in-situ-collection) or as isolated shapes (isolated-collection). In-situ-collection requires the user to plot a whole diagram of the targeting diagram domain. Isolated-collection considers only the individual shapes. For example, if one aims to build a recogniser for flowcharts, in-situ-collection would have the participants actually plotting flowcharts, while in isolated-collection they only have to draw rectangles, circles, lines, arrowheads and diamonds separately – which are the shapes appeared in flowcharts. In

comparison, isolated-collection is easier: participants only need to know how to plot these shapes. Furthermore in-situ-collection takes more processing time because the collected data need to be labelled manually, while in the case of isolated-collection, because the shapes are already separated, this process is not required.

Field et al. (Field et al., 2009) conducted a study and revealed that there is no significant difference between the two data collection methods. On the other hand, Schmieder et al. (Schmieder et al., 2009) report that the classifiers trained with in-situ-collection have better performance in most cases.

The decision was made to apply in-situ-collection. We believe people draw differently when they are drawing a shape within a diagram, as part of their thinking is used to consider the structure. Also, the spatial and temporal contexts captured during in-situ collection, such as "arrow heads are normally drawn after a line is drawn", are valuable. Furthermore, the percentage of shape occurrence may also be important, for example "flowcharts only contain two ellipses" or "the number of lines would be similar to the number of arrowheads in directed graph" may help resolve many ambiguities. Although it is more difficult to find participants since they need to have knowledge in plotting these diagrams, the resulting classifiers should be more reliable because the targeting users are those who know how to plot these diagrams. We also decided to evaluate this assumption by collecting a diagram set in isolated-collection and evaluate its performance against the other datasets.

Initially none of the collected strokes are classified. We have to manually label the shape class for each stroke, so data mining algorithms can train classifiers based on these classifications. Features will be calculated from these labelled strokes which allow data mining algorithms to use and generate classifiers. Processes may be required on these calculated features to ensure they can work with the algorithms.

3.2. Data Mining

Once all features are assembled they are ready for data mining. WEKA is selected as the data mining tool, because it provides many data mining algorithms and is open source software which allows the actual implementation of each algorithm be analysed. Suitable algorithms implemented in WEKA will be applied to train the data, and those algorithms with better performance will be selected for optimisation. These optimised algorithms will

then be compared. We believe by analysing the results of our experiment the strength of algorithms can be revealed.

Two assumptions are made in this study. First, the diagram domain can be generalised. As described in section 3.1, we are more interested in optimising the complete diagram domain instead of individual sub-domains. Second, our data can generalise the diagram domain. Even if the diagram domain can be generalised, there is no guarantee that the data we used in this experiment can represent the hypothesis space. But even if they do not represent every aspect of it, we believe certain common properties will exist between them.

To test if these two assumptions stand, not all collected datasets are applied in this step. One dataset will be reserved for evaluation; if it demonstrates the same behaviour as the ones used in the experiment, we will have more confidence in these assumptions.

3.2.1. Algorithm Selection

An algorithm is a set of rules the computer can follow, and the algorithms provided by WEKA can generate classifiers based on the training data. Although WEKA provides many algorithms, not all of them are suitable to our dataset. This step aims to remove the ones which are inappropriate to the diagram domain represented by our dataset, allowing us to concentrate on the promising algorithms. Many reasons can make an algorithm unsuitable, for example, it may lack the accuracy to diagram recognition problems due to its simplicity, or it can be designed to only support nominal attributes when most of our calculated features are numeric. Different algorithms perform differently in different problem domains, and we aim to find the most suitable algorithms for diagram recognition.

3.2.2. Algorithm Optimisation

The goal of this step is to analyse the strength and performance of the selected algorithms, and to optimise their performance by altering their settings. All the experiments are done through the experimenter interface WEKA provides. Because the training and testing time is considered, the same computer will be used for the same set of experiments.

Performance of a classifier can be judged by three aspects: the accuracy, the time required to test input data and the time required to train it. The accuracy is the most important

attribute, since a classifier which cannot classify accurately is of little use. Since we aim to build recognisers for eager recognition, there is little tolerance for the time to test input data; it needs to be fast enough so users will not have to wait before drawing the next stroke. However, as long as it is fast enough, there is no need to optimise it. Time to train a classifier is relatively unimportant, since it happens only once in the life cycle of a classifier; however, if a classifier takes much longer to train without producing significantly greater accuracy, the faster versions would be considered better in this study.

The first two experiments are conducted with 10 fold cross validation with ten rounds. For each round the data is split into ten equal-sized groups randomly; each group will take turns being the testing sample while the remainder constitute the training sample, as shown in Figure 18a. In total a hundred runs will be performed in one experiment and the resulting accuracy will be the average of this hundred runs.

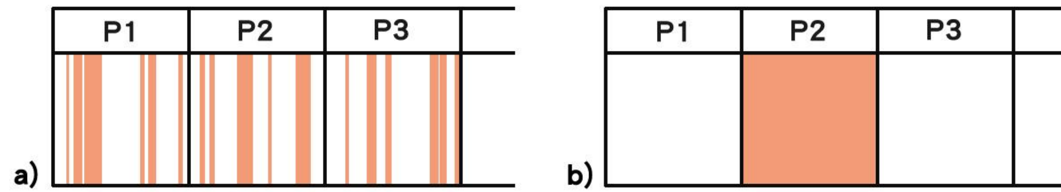


Figure 18. Comparison between cross validations: (a) based on sample, (b) based on participant

While for most studies 10 fold cross validation can reduce the effect of overfitting, it is not perfect for diagram recognition. Certainly it can remove the overfitting of training samples; however, it cannot remove the overfitting of drawing styles. Consider in Figure 18a (the orange means testing samples) the traditional cross validation performs the random selection based on samples, hence data from each participant will appear in both training and testing samples. If drawing style does matter, because the users of a recogniser are likely be different from the ones providing the training data, the result of 10 fold cross validation becomes optimistic. A better evaluation method would be the case of Figure 18b, where the random selection is based on participants. However this option is not provided by WEKA.

Although drawing styles are important (Alvarado & Davis, 2004), it may well be the case that the data collected are enough for the algorithms to filter the minor differences and represent the important features. The splitting experiments are conducted to verify the number of examples required to have a usable recogniser, as well as trying to reveal the performance the recognisers can achieve if style information is excluded.

Each algorithm will undergo four steps of optimisation: basic experiment, optimise experiment, splitting experiment and attribute selection.

Basic Experiment

Each algorithm in WEKA contains several settings which can be adjusted to alter the nature of that algorithm. However, we are uncertain about the effects of each setting. This experiment tries to find the best value for each setting of each algorithm. The aim is to maximise the accuracy. WEKA provides an algorithm called "GridSearch" which can find the best settings; however it is computationally expensive. In addition, while it returns the best value of settings, the value is decided in internal cross validations, which has the probability of still not being optimal. Furthermore the relationship between the settings and the accuracy is not shown, which reduces its validity because there may be associations between a particular data type and others, which we may find from this information. Hence instead of using GridSearch, we decided to explore the settings by manually changing them.

To detect the effect of each setting, they are considered individually. If one setting is modified to examine the difference it makes, all other settings stay in their default values. Both true and false cases will be applied to binary settings, and numerical settings will be tested with a series of different values. For each setting, the accuracy generated by changing its value will be compared, and the value which returns the highest accuracy will be taken as the optimised value for the setting. If a modification has different effects on different datasets, in this study the average of all datasets is considered.

We assume there is no inter-relationship between the settings. Inter-relationship means that the combination of two settings causes different effects when compared with the effects of changing only one of them. Consider a simple example: individually, "drunk" and "driving a car" will not necessarily lead to the result of "illegal driving"; the result only changes if the two occur at the same time. This assumption is likely to be invalid; however, there are many settings for each algorithm, and most of them are numerical settings. It is not possible to have a test capturing every single combination of features, and there is no good reason to select some combination over the others. This assumption was made to simplify the experiment.

Optimise Experiment

The best settings are found in the basic experiments; in this experiment these results are combined to form the “optimised algorithm”. Based on the “no inter-relationship” assumption, the resulting accuracy should be improved. The optimised setting is tested against the default setting.

All three aspects, the accuracy, training time and testing time are considered. First, the testing time of an algorithm must be under a certain threshold, to ensure it is suitable in eager recognition. Second, if the optimised settings make the generated classifiers significantly more accurate, the algorithm is considered as being truly optimised. Lastly, time to train is considered if the difference in accuracy is marginal; if it takes much longer to train for only a minor improvement, the default setting is suggested to be a better option.

Splitting Experiments

The previous experiment evaluated the optimisation settings. However the effect of drawing styles may reduce the validity of the experiment. Additionally, we are interested in the association between the numbers of training examples versus the resulting accuracy of a classifier. Based on these interests, two sets of data splitting experiments are designed.

For each experiment, the data is split into training and testing. A 10% splitting indicates 10% of the data is selected for training and the remaining 90% are used for testing. Nine different splits are chosen, from 10% to 90% with 10% intervals.

The first set of experiments is RandomSplitting. In this experiment, the training examples are selected randomly, similarly to Figure 18a. To reduce the effect of noises, ten rounds are performed in one experiment, while during each round the data is selected with different random seeds. The resulting accuracy for one experiment is the average of these ten rounds. This experiment can discover the numbers of data required to return a reliable classifier. However, since the splitting is done randomly, similarly to 10 fold cross validation, the effect of drawing styles is still presented.

If a classifier is to be trained by the people who will be using the classifier, RandomSplitting and 10 fold cross validation will show the performance, because we can assume several strokes from each person will be taken and tested on the rest of strokes

which are not seen. This option also simulates the case when the collected data can essentially represent the whole population.

The second set of experiments is OrderedSplitting. In this experiment, the training examples are selected from the start of the dataset. For example, with a dataset of 500 strokes, 10% splitting will take the first 50 strokes to be training examples, while the rest become testing examples. Because our datasets are organised in the order of participants, and the numbers of strokes drawn by each of the 20 participants are similar, we can assume that each 10% is equivalent to two participants. Only one round of experiments will be performed for each experiment, which decides the resulting accuracy.

As the testing data is drawn by different participants from those who contributed to the training data, this experiment better simulates the situation when the user is different from the contributors of the training data. Also this experiment can demonstrate the difference in drawing styles – if the resulting accuracy is similar to the RandomSplitting result, the style difference can be ignored, otherwise the style difference needs to be taken into account (by for example always having the user training the recogniser).

OrderedSplitting can be further improved by performing more experiments for each percentage, each containing different training participants, and averaging the results, as shown in Figure 18b. This mechanism is not implemented because the major focus of the splitting experiments is not to compare individual algorithms, but to compare the difference between the optimised setting and the default setting of each algorithm. With the same algorithm and the same data, we believe if optimisation is successful it should show improved accuracy.

Attribute Selection Experiment

An algorithm may consider all the given features. However, there may be misleading or duplicated information in these features which can potentially affect the performance of the generated classifier. We believe selecting better features has the potential to improve the algorithms. Instead of finding better features and removing them heuristically, WEKA provides a special meta classifier called “attribute selected classifier”, which allows users to specify an attribute evaluator and an algorithm to use, and it will apply the evaluator to find the better features first, and use them in the algorithm.

Many attribute evaluators are provided. Since we already know which algorithms to use, instead of exploring all these evaluators, Dr. Frank suggested the application of wrapper subset evaluator. This evaluator can search the better features with the knowledge on how the training sets interact with the applied algorithm (Kohavia & John, 1997). This technique is also applied by (Paulson, Rajan, Davalos, Gutierrez-Osuna, & Hammond, 2008) in diagram recognition which reported successful results.

Originally this step was to be performed with optimised settings, to see if it can further optimise the best configured algorithms. However the experiment requires too much time. For example, one round of the default configured LMT can take two and half hour to complete on average. A significantly longer time is required for optimised versions, which is beyond the computational time available within the project time span.

Although the desired setting is to apply 10 fold cross validation which in default take a hundred rounds, it would take 10 days to run on the default configured LMT with one dataset. The time span is more than we can afford, so we decided to find other ways to make the comparison. Instead of using 10 fold cross validation, RandomSplitting is selected since it is ten times faster and has similar effects to 10 fold cross validation. The percentage is selected to be 20% since most algorithms can achieve 90% accuracy with this many samples. Also, instead of using the optimised algorithms which may use more time to generate a classifier, the default ones are used.

3.2.3. Algorithm Cooperation

After the four stage optimisation, algorithms are optimised with their best settings. According to previous studies (Alimoglu & Alpaydin, 2001; Connell et al., 2000; Kara & Stahovich, 2004), the resulting accuracy may be further improved by combining these algorithms.

The first method of combining the classifiers is through Voting. An algorithm alone may give incorrect result, however the combination of them has a better chance to report the correct classification. The second method is through Stacking, where each classifier has a different weighting toward the problem, based on their performance toward different areas. Different combinations were considered during the experiment, because we believe different algorithms perform differently, and even a worse performing algorithm can aid a better algorithm if its strength covers a different aspect.

3.2.4. Revised Rubine

Rubine (Rubine, 1991) is a simple classifier introduced in the literature review. It is widely applied in recognition tools, and is found to have good performance compared with more recent recognisers (Schmieder et al., 2009). Because evidence shows that different features can improve its classification accuracy (Plimmer & Freeman, 2007), we believe with a systematic mean of selecting features from our rich feature set, the algorithm can be further improved. Data mining is applied for this feature selection. We wanted to select features based on the structures of the WEKA classifiers; however, because the algorithms selected in this study are complex, their structures are difficult to interpret (Witten & Frank, 2005). The decision was made to use the attribute selection feature provided by WEKA to automatically select features.

3.3. Implementation

DataManager (Blagojevic et al., 2009) can calculate 119 features for each stroke; but it cannot utilise these features to generate a recogniser. Although classifiers can be generated by feeding these features into WEKA, using the WEKA interface directly with DataManager has three disadvantages. First, the interface provided by WEKA to train a recogniser is complex. Users have to spend much time to learn how to control and configure the tool. Second, professional knowledge is required; WEKA provides many algorithms, however it is unlikely that non-expert users can decide the most suitable algorithm for their data. Third, even if users successfully generated recognisers, they still have to blend WEKA with their project; for DataManager or any C# based tools, the blending of Java based WEKA still requires effort. Furthermore, this project involves the optimisation of Rubine, which needs to be implemented with the WEKA algorithms.

Hence, we decided to make a bridge connecting WEKA and DataManager. Rata.SSR, the intermediate program will address all the problems mentioned above and have the following attributes: it should first be simple to use; the choice of algorithms should be restricted to the most suitable ones for diagram recognition, in order to simplify the task for users; it should also be extendable such as being capable of working with classifiers generated directly from WEKA, so the advanced user can still optimise the algorithm themselves; lastly, it must support C# interface.

3.4. Evaluation

To test the performance, the recogniser generator, Rata.SSR, will be used to generate a number of selected classifiers, and will be evaluated against existing recognisers supported in Evaluator (Schmieder et al., 2009). The evaluation will include the dataset which is not used in the optimisation process, to ensure the optimisation result can be applied not only to the trained domains but also to the whole diagram domain. If our recognisers produce performance superior to others then data mining is proven to be a good approach in sketched diagram recognition, and even if there is no significant difference between the levels of accuracy, our artefact is still beneficial because it is flexible and requires no expert knowledge when being used.

3.5. Summary

In this chapter, we have described the process and decisions required to implement a generic recogniser generator. Decisions on sample data are specified, and these data are to be collected via in-situ-collection. Algorithms are optimised not only to specific datasets, but toward the whole diagram domain. The details on how the optimisation would be conducted are explained; three levels of optimisation will be conducted to validate the best configuration for settings for individual algorithms, while attribute selection and cooperation algorithms will be applied to further boost accuracy. The optimised algorithms are to be implemented into Rata.SSR, a recogniser generator, where it should be simple to use yet extensible. The implementation is used to generate a set of recognisers and evaluated against existing recognisers. Chapters 4 to 6 describe the details of this process.

Chapter 4
Data Mining

The previous chapter outlined the methodology for analysing the algorithms and optimising them. These optimised algorithms will be implemented in a recogniser generator, which will be evaluated against existing classifiers.

This chapter focuses on the data mining to identify the best algorithms to be used in sketched diagram recognition. Four different datasets are collected to represent the diagram domain. Algorithms implemented in WEKA(Hall et al., 2009) are tested, and together with professional suggestions the most promising ones among them are selected for detailed analysis. Four stages of optimisation are performed on these algorithms, as described in section 3.2.2, and they are also combined to form stronger classifiers. In addition, data mining is also applied to find the best configuration of Rubine features.

4.1. Data Collection and Preparation

Data mining is a technique to find underlying relationships among a collection of data. If the relationships between the features and the target classification are found, the target can be predicted as long as those features are obtained. Hence the most important element to a good classifier is the training data. These data need to be collected from real people under real usage, to maximise the chance of capturing the relationships in a real world use case, as described in section 3.1.

4.1.1. Diagram Sets

We want to collect diagrams with different characteristics. Many types of diagrams exist, however we are restricted by two requirements: first, each shape presented in these diagrams needs be formable with only one stroke; second, participants need to have knowledge in drawing these diagrams, hence the diagrams need to be of a standard notation. With consideration of different diagrams appeared in engineering and science, the decision is made to collect the following datasets: shapes, directed graph, flowchart and class diagram.

Table 2. The different datasets

Dataset name	Rectangle	Ellipse	Triangle	Line	Arrow	Diamond	Total	Collection
Shapes	80	80	80	80	80	80	480	Isolated
Directed graph	0	146	0	172	173	0	491	In-situ
Flowchart	99	42	0	242	239	61	683	In-situ
Class diagram	170	0	57	215	96	60	598	In-situ

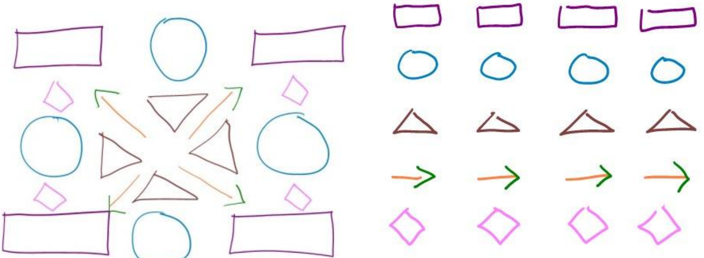


Figure 19. ShapeData from participant 14 and 19

Shapes dataset (ShapeData) includes six shapes, each drawn in isolation, and has no relationship with other shapes. This simulates isolated-collection as described in section 3.1. All shape classes used in other datasets can be found within ShapeData.

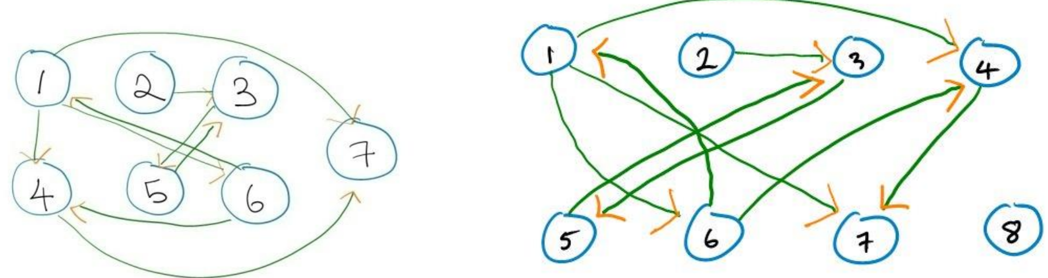


Figure 20. GraphData from participant 14 and 19

Directed graph dataset (GraphData) includes three shape classes, in which a certain relationship is presented between them. Although the diagram is relatively simple both in the number of shape classes and their relationships, already it introduces challenges including the need to distinguish between arrows and lines, the unorganised sequences each shape is drawn in and the overlapping between each strokes.

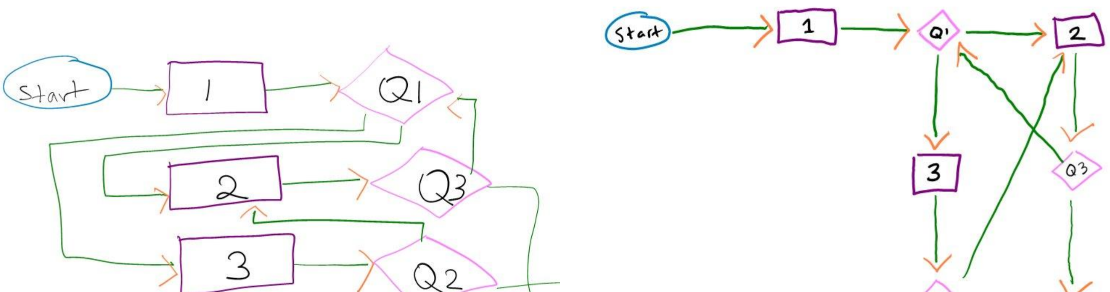


Figure 21. Flowchart diagram from participant 14 and 19

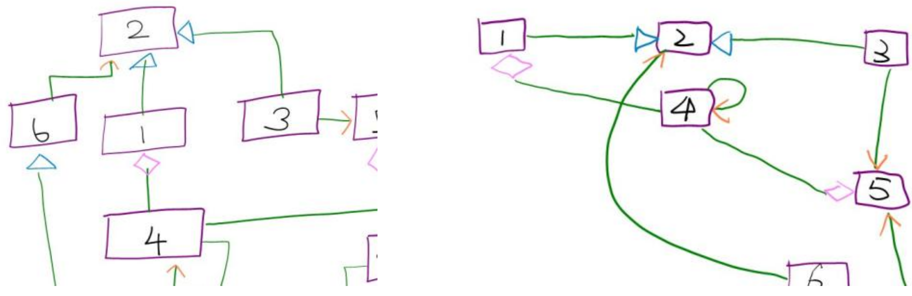


Figure 22. Class diagram from participant 14 and 19

Flowchart dataset (FlowChart) and class diagram dataset (ClassData) both include five shapes with relationships between them. They represent difficult diagrams – more shape classes and more complex relationships.

The four datasets can be categorised by two characteristics: the method by which they are collected, and the complexity represented by the number of shapes. ShapeData is the only dataset utilising isolated-collection; it has the highest number of shape classes, which indicates the highest complexity measure over all datasets. Hence, if different collection methods do not affect the accuracy, the average recognition rate for ShapeData should be lower than all other datasets. On the other hand, GraphData, FlowChart and ClassData are all collected by in-situ-collection. Among them, GraphData is certainly easier to recognise because it involves only three shape classes. In comparison, FlowChart and ClassData both have five shape classes and similar distribution, which we believe are more complex than GraphData.

4.1.2. Participants and Collection

The participants should be experienced in drawing these diagrams. We randomly picked university students who were working in the computer lab, ensuring they are majoring in either computer science or software engineering, because both departments teach all diagrams we aim to collect. The age of participants varied from 18-26 and they were of mixed gender.

Data collection was done with a Toshiba R400 tablet laptop with Windows Vista installed. It utilises *Intel® Core™ Duo Processor U2500* and has 2GB RAM installed. DataCollector, which is implemented within DataManager(Blagojevic, 2009), was used. Before the collection, participants were asked, through the form shown in Appendix A, to self rank their familiarity to each diagram. No question was asked regarding ShapeData because we assume people know how to draw the required shapes.

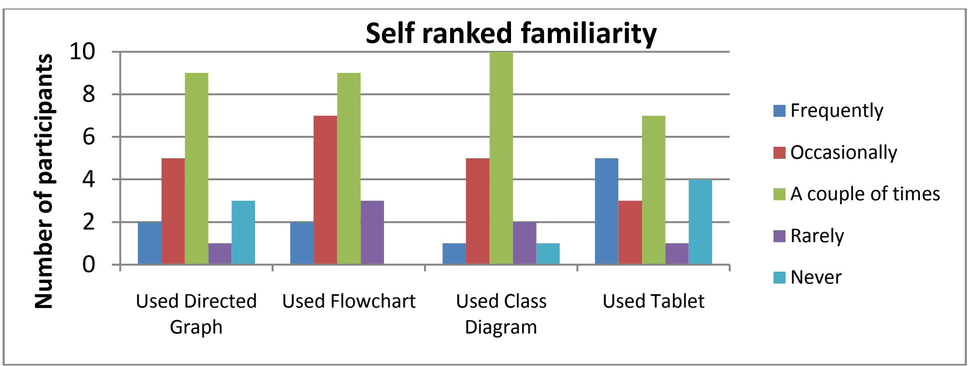


Figure 23. Self ranked familiarity

Any scale apart from Never indicates the participant has used the diagram before. Directed graph has several Never scales, according to which those participants said that they have used the diagram without knowing it is called directed graph. The participant who answered Never in the class diagram has seen but never drawn a class diagram before.

Participants were asked to play with the tablet and the DataCollector in DataManager until they are confident in using the digital pen to create ink on it. They are then asked to complete the ShapeData. Each participant was requested to draw four shapes for each shape class.

Upon the completion of ShapeData, two information sheets are provided: the sample graph sheet and the dictionary sheet. They contain information on how to draw GraphData, FlowChart and ClassData, as shown in Appendix B. Participants will continue to plot these diagrams with the aid of these two information sheets.

The sample graph sheet contains one simple example for each diagram type. They are in their simplest form, presented only to remind participants of how the elements of the diagram may be connected. Since the structure is far simpler than what participants would be requested to plot, and it is generated digitally with no hint of how a shape may be plotted, we believe they should not have effect of direct copying (Field et al., 2009).

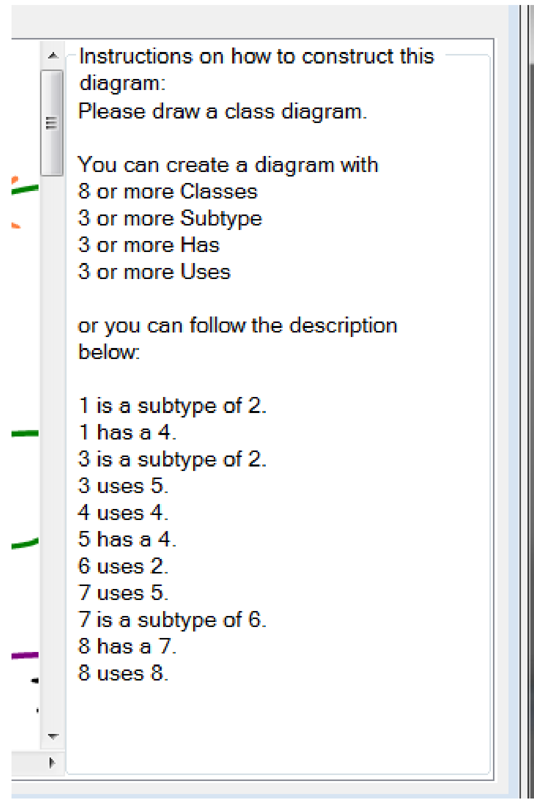


Figure 24. Instruction for drawing ClassData

The number of shapes for each dataset needs to be fixed to ensure each participant contributes equally to the data mining process. However, displaying the exact diagram may lead to participants copying and lose the thinking process. Hence, instructions are shown as text in DataCollector as shown in Figure 24, to avoid participants directly copying the structure. The instruction for each dataset can be found in Appendix C. These instructions are further explained by the dictionary sheet shown in Appendix B. Participants are told that they can follow the description text, or they may create their own customised diagrams as long as these diagrams have sufficient number of shapes as requested. Most participants decided to follow the preset instructions.

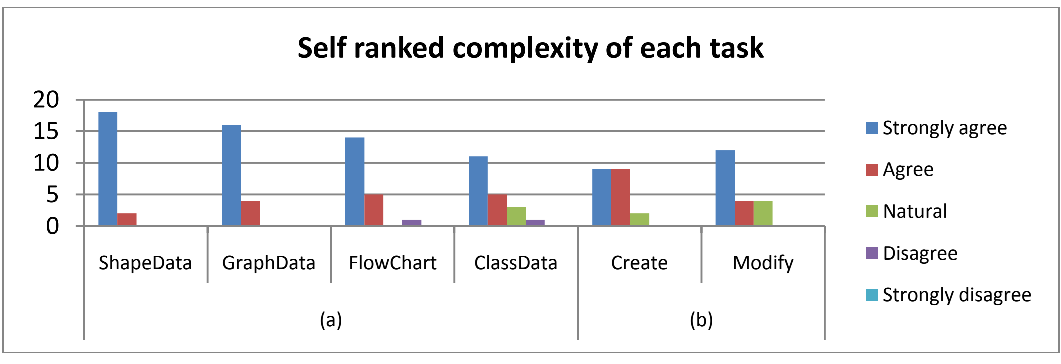


Figure 25. Self ranked complexity of each task

At the end of data collection, participants were asked to self rank how easy it is to understand and perform each task, through the questionnaire shown in Appendix A. According to Figure 25a, the reported complexity of each diagram is similar to the complexity when they are designed in section 4.1.1. The design of individual tasks may also contribute to this result, as one participant reported that the ClassData task can be made simpler. On the other hand, Figure 25b shows that most participants agreed DataCollector is simple to use, even if they had never used a tablet computer before. This indicates that the complexity would not come from the user interface of the DataCollector. Seven participants reported that they need to remember to draw single strokes, which indicating that although they can be achieved, single stroke shapes are not very intuitive.

An interesting observation is that all participants can draw all shapes with single stroke fluently in the practice session and the ShapeData task. Mistakes in multi-stroke only occur in the in-situ-collection tasks. This shows the thinking processes are affected when the structure of a diagram needs to be considered.

4.1.3. Labelling

After datasets are collected, they need to be manually labelled before they can be used. Labelling is referred to as classifying each shape within a diagram. The FeatureCalculator provided in DataManager can generate all kinds of information from a given stroke, however it is the user's responsibility to classify these strokes, so the computer can know which shapes are the same and try to find their similarity. The labelling can be done via DataManager's Labeller.

Shapes accidentally drawn with multiple strokes or multiple shapes drawn with one stroke are ignored, because we are only interested in single stroke information. These situations exist because we avoided correcting the participants during the data collection, since such action interrupts the plotting process and may lead to unnatural behaviours. However among more than two thousand collected shapes, there are only 13 such cases, which we believe will not impact on the validity of our data. Strokes generated when participants accidentally touched the tablet with the pen are also ignored. These unlabelled strokes will not participate in the data mining process. Furthermore because the aim of this research is diagram recognition, all written characters are ignored.

4.1.4. Feature Calculation

The version of DataManager (Blagojevic, 2009) used in this study is capable of generating a total of 119 features, or any subset of them. These include both nominal and numerical features. Four features are excluded from the feature calculation in this study, including the Microsoft Divider Result, the Microsoft Divider Closest Stroke, the Microsoft Divider Previous Stroke and the Microsoft Divider Next Stroke. These features are slow to calculate; if they are included in the feature calculation, 1.42 seconds will be required to calculate the features for each stroke. This is too slow for eager recognition. Furthermore, we do not have knowledge on how Microsoft Divider is implemented, or when it may be altered, and this black box type feature cannot be understood if we want to discuss the important features.

Thus a total of 115 features are used in this experiment. The time taken to calculate these features for a single stroke is 0.087 seconds. While this number is acceptable, in data mining the time taken for a classifier to recognise a stroke must also be considered.

4.1.5. Feature Cleanup

Several values of features are undesired. One of them is empty value. If certain components are missing, some features can return empty value indicating the calculation cannot be performed. It is not a big issue for WEKA algorithms since most of them can deal with missing values, but it is critical to Rubine. Other types of undesirable values include infinity, negative infinity and NaN. Not all algorithms in WEKA can deal with these values, and they can prevent some algorithms from running, as well as causing others to return bad results. These values are replaced with nothing (to make WEKA treat them as missing values).

Unlabelled strokes will still present in the calculated feature list, with empty class value. They are also removed manually, because they are not of interest in this study. Another process which needs to be done is to move the class feature to the end of the feature list, since the experiments will be conducted in WEKA’s experimenter, which assumes the class variable is the last feature.

4.2. Scope of Algorithms

The aim is to select the algorithms which are performing well and to optimise them further. A simple dataset of undesirable graphs which was collected in a previous study was used; it contains circles, lines and written characters. Initially the experiment removed all characters; however most classifiers achieved high accuracy which cannot help decision making when choosing algorithms. Hence another experiment was conducted which considers the characters. We believe good algorithms should perform well in this relatively simple dataset even if simple characters are included which adds a layer of complexity.

Table 3. Algorithms with accuracy over 90%. Final candidates are shaded in red

Algorithm	Accuracy	Algorithm	Accuracy	Algorithm	Accuracy
SMO	96.18	Dagging	93.61	J48	92.11
Simple Logistic	96.13	Random Sub Space	93.41	RBF Network	92.10
LMT	96.08	Bagging	93.38	Attribute Selected Classifier	91.86
Multilayer Perceptron	96.01	NB Tree	93.32	Decision Table	91.84
Random Committee	95.58	Multi Class Classifier	93.27	Logistic	91.77
Classification Via Regression	95.53	Filtered Classifier	93.00	J48graft	91.75
Rotation Forest	95.48	Class Balanced ND	92.87	Ridor	91.73
FT	95.12	DTNB	92.81	NNge	91.18
Random Forest	95.09	Ordinal Class Classifier	92.79	Random Tree	90.84
IB1	94.75	Data Near Balanced ND	92.77	REP Tree	90.76
Ibk	94.75	Bayes Net	92.74	Naïve Bayes	90.09
Logit Boost	94.68	JRip	92.70	Naïve Bayes Updateable	90.09
END	94.15	ND	92.66		
PART	93.72	LAD Tree	92.53		

The classifiers which achieved 90% accuracy during the experiment are shown in Table 3. Algorithms which failed to classify the dataset or achieved poor accuracy are not considered. Although many classifiers are filtered off, there are still 40 algorithms remain which are more than our project scope can handle. Picking the top ones is possible, however the fact that a recogniser performs well on this dataset does not guarantee its performance under other datasets; furthermore, because we considered combining

recognisers for further accuracy improvement, algorithms with strength in different aspects are desired, but such information cannot be deduced from the accuracy alone.

In the end, we decided to seek suggestions from a data mining expert. Associate Professor Eibe Frank, a researcher in the WEKA project and the author of a data mining book, *Data Mining: Practical Machine Learning Tools and Techniques* (Witten & Frank, 2005) shared his knowledge and experience with us. Eight algorithms were suggested, including Bagging, END, LADTree, LogitBoost, LMT, MultilayerPerceptron, RandomForest and SMO. They are marked red in Table 3. These algorithms represent a range of different approaches, including tree generating algorithms, SVM, neuron network and meta-algorithms which boost the performances.

BayesianNetwork was also selected. Although it was not suggested by Associate Professor Frank, because it was applied in many of the previous studies discussed in Chapter 2, we wanted to compare it with other algorithms. Although there are classifiers which appear to have better performance, for the scope of this project they are not explored further.

4.3. WEKA Algorithms

In this section each selected algorithm is analysed. Each subsection starts by introducing the algorithm, followed by the effect of changing its settings. These alternations are plotted onto graphs for ease of comparison, and the information can be found in the accompanying CD. The four experiments to optimise the algorithm, which were introduced in section 3.2.2, are conducted. After all algorithms are optimised, they are ranked by comparing the average accuracies.

These experiments require a long time to run, and for most of the study the available computation power was very limited. Initially only one machine was used for data mining which uses *Intel® Core™2 Duo Processor E8400* and 4GB RAM, and close to the end of this study three machines with similar specification, and two virtual machines based on *Intel® Xeon® Processor E7330* and 2GB memory were available for use. However, on any of these machines there are experiments which can take weeks to complete, for example, the attribute selection for MultilayerPerceptron.

All experiments in this section were done with the Experimenter interface provided by WEKA, which allows batch data mining. Several rounds of experiments were conducted.

At the start of this project WEKA 3.6.2 was used; however because some required functionality is unsupported in the version, we changed to version 3.7.1. All experiments were repeated because for some experiments the two versions returned different results. Furthermore when mislabelled data are found we repeated the experiments to ensure the correctness of the results. One exception is the ShapeData. It was collected to simulate the situation of isolated-collection; however at the end of evaluation we found the Temporal Spatial Context features (TSC-features) were not removed. Because each shape should be collected individually, there should not be any TSC-features. When analysing the impact by repeating all basic and optimise experiments, we discovered the difference is marginal, and decided not to repeat these for splitting experiments and attribute-selection; however, they are corrected for the basic and optimise experiments, and also all the evaluations.

Settings that show no effect toward the accuracy will not be discussed explicitly. Debug and Seed, which appear in almost all algorithms, are not discussed. Debug is not discussed because it only prints additional information, which is not directly related to optimisation. Seed changes the value generated whenever a pseudo-random number is required; while it does affect the accuracy, this is only due to the difference in random numbers, which is not truly making optimisation.

The graphs plotted in this section may not have the same scale, because it does not make sense to compare across different settings. They are adjusted to allow the best presentation of the data. Among each algorithm, the splitting experiments are changed to the same scale, which allows comparison between OrderedSplitting and RandomSplitting.

4.3.1. Bagging

Bagging, the shorthand for "**b**ootstrap **agg**regat**ing**", is a method which generates multiple classifiers of the same kind and uses them with a voting system. All generated classifiers have the same weight during the voting process. It has the ability to turn a weak classifier into a stronger one, especially if the classifier is unstable (Witten & Frank, 2005). In contrast, it should not be applied with a stable classifier, which may degrade its performance (Breiman, 1996a).

Different training sets are required to train different classifiers. Such requests can be impractical because normally there exists only one training set, with a limited number of samples. Dividing it may reduce the information presented and reduce the effectiveness of

learning. To deal with this problem, given a set of training instances, Bagging generates different datasets by randomly replicating some of the content and removing some of the others. The resulting dataset will have the same size as the original, but with different content. Such an operation does not alter the data, because there is no additional information which was not presented in the original data. The results indicate the success of the process: using the Bagging system is usually better than using the single classifier only, and is never substantially worse (Witten & Frank, 2005).

Basic Experiment

Table 4. Bagging options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Bag Size Percent	Size of each bag, as a percentage of the training set size.	Int
Calc Out Of Bag	Whether the out-of-bag error is calculated.	Bool
Classifier	The base classifier to be used.	Other
Debug	If set to true, classifier may output additional info to the console.	Bool
Num Iterations	The number of iterations to be performed.	Int
Seed	The random number seed to be used.	int

The options available in WEKA for Bagging are shown in Table 4. And we will discuss each in this section.

- Bag Size Percent (default: 100)

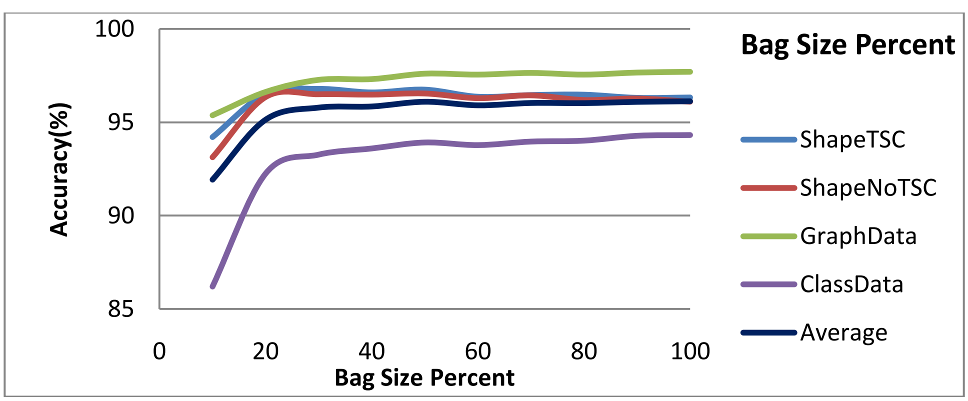


Figure 26. Bagging: Bag Size Percent

This decides the number of training samples within each bag. Smaller bag size accelerates the process; however, on average larger bag size brings higher accuracy. This behaviour is expected because with a small bag size, less data can be used in each training process, which according to Rubine (1991) will cause reduction in performance. On the other hand, it is also very interesting that only a small amount of data, around 20%, is required for all datasets to achieve good performance.

While GraphData and ClassData continue to increase after the turning point, ShapeData showed different behaviour. Its performance starts to decrease after the maximum accuracy is reached. We believe this showed the simplicity of ShapeData in that the small bag size is enough to train it well, and a larger bag size starts to overfit the data.

- Classifier (default: REPTree)

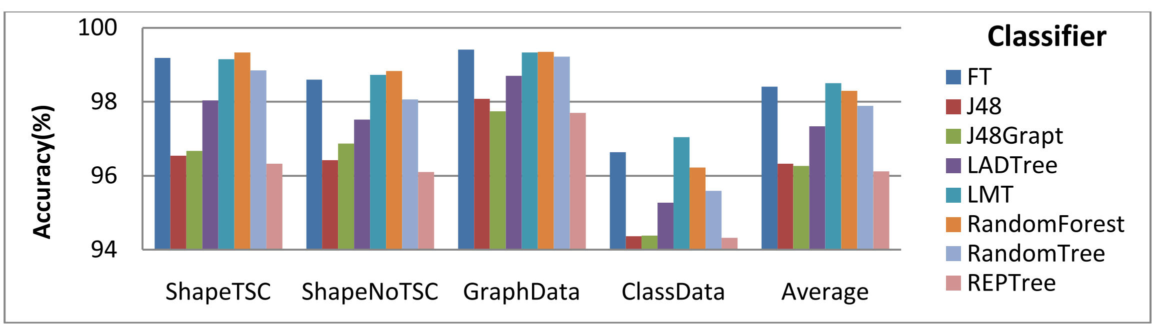


Figure 27. Bagging: Accuracy vs. Graph types

Better trees have better performance. However because these trees can be tuned by changing their settings, we decided to only apply the default tree in our study, and assume the optimised configurations can also be applied to the other trees.

- Num Iterations (default: 10)

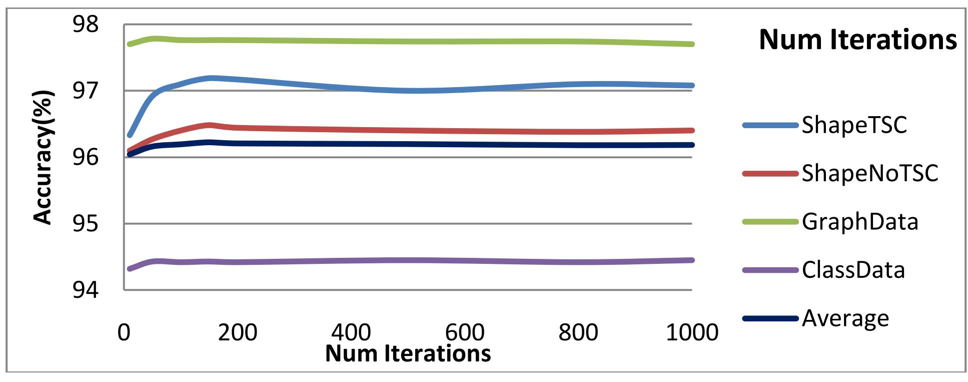


Figure 28. Bagging: Accuracy vs. Num Iterations

This defines the number of iterations. One tree is generated during each iteration; in other words the more iterations Bagging runs the larger voting committee it will have. However, once enough bootstraps are presented, since they are still based on the same information, it will not continue to improve. A stable status will be reached, where the change of settings brings little effect. According to the data, 50 – 100 iterations will guarantee the reaching of steady state.

ShapeData showed a different trend from other datasets. It is improved the most, but is also the most unstable when the stable state is reached. Considering BagSizePercent, ShapeData showed indications of overfitting while the others reached good performance at 100% bag size. We believe its large improvement in Figure 28 is because more iteration can help remove this overfitting. Additionally, the inclusion of TSC-features introduced even more improvements in accuracy, while causing the stable state to become more unstable. The unstable behaviour should come from the proportional difference; because the increment is large, the variation is also very significant.

- Ineffective settings

CalcOutOfBag: The out of bag error can be referred to as the generalisation error. Since about one third of the training samples are left out from each bootstrap, they can be used to estimate the error rate (Breiman, 1996b), and return a more reliable estimate. According to our data no impact on accuracy can be found.

Optimise Experiment

Combining the observations from the basic experiments, the algorithm is optimised with the following configuration:

WEKA Settings	Bag Size %	Calc Out Of Bag	Classifier	Debug	Num Iterations	Seed
Default	100	False	REPTree	False	10	1
Optimised	100	False	REPTree	False	70	1

Only the number of iterations is changed. Although changing the base classifier can further boost the accuracy, it was not explored due to the reasons explained in the last section.

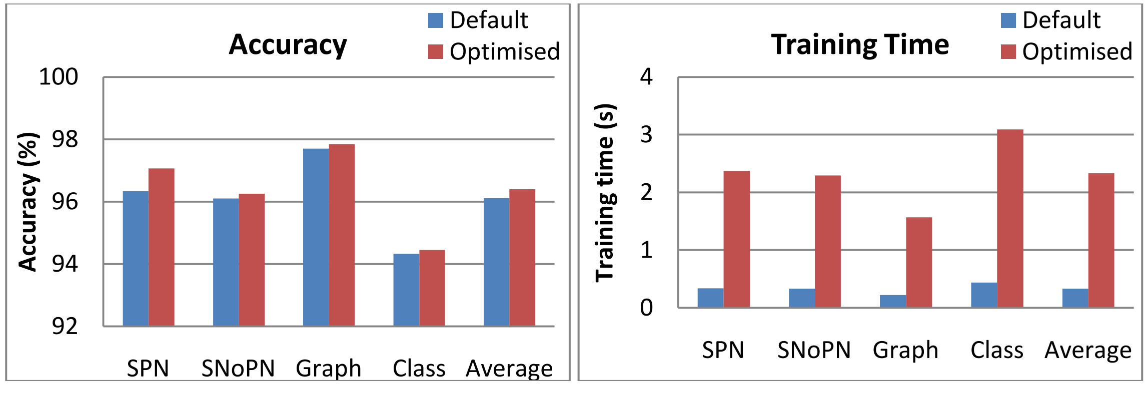


Figure 29. Bagging: Optimise experiment

The optimisation can improve the accuracy by 0.4% in average, with training time eight times longer. Since in both settings the testing time is less than 0.01 seconds and the optimisation improved all datasets, we think optimisation can be considered, but is not necessary as it is not significant.

According to Figure 28 the accuracy will reach a stable state after the maximum accuracy is reached, hence, if the training time is affordable, the number of iterations can always be extended. Furthermore if data is collected by isolated-collection, for example in the situation of gesture recognition, our study shows smaller bag size can be used to improve the accuracy.

Splitting Experiment

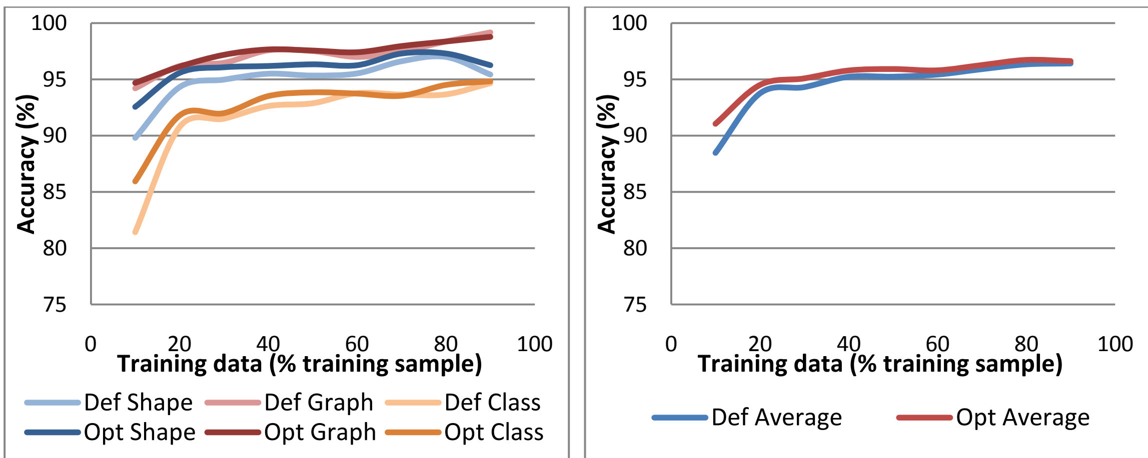


Figure 30. Bagging: RandomSplitting

The optimisation has larger significance with less training data. It can be explained by the extra iterations performed, since within each iteration there is duplication process to introduce variations. These duplications can simulate the data which are not selected. The optimised results are generally better, although at high percentage the differences become less significant.

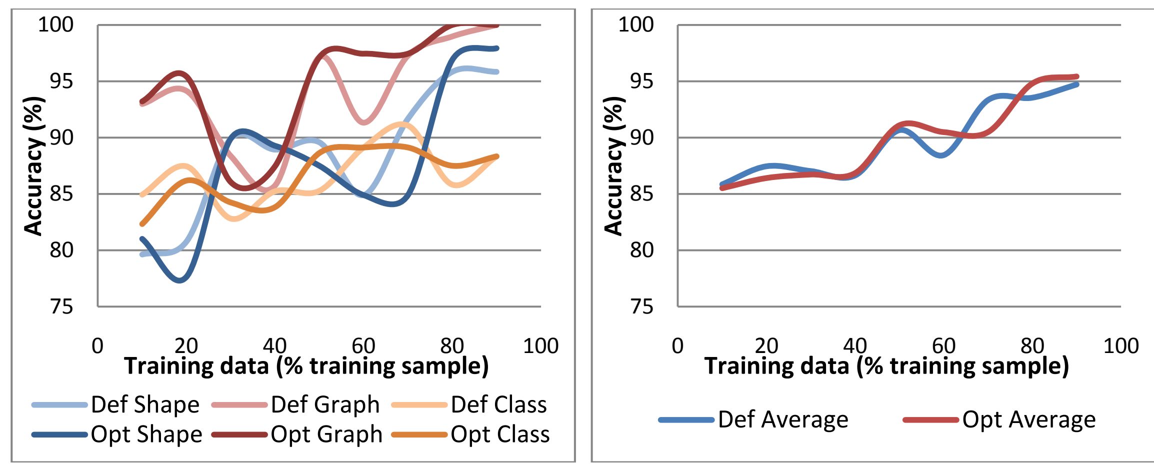


Figure 31. Bagging:OrderedSplitting

While in RandomSplitting the optimisation is always improving, the OrderedSplitting does not consistently demonstrate the same behaviour. We believe it is caused by overfitting. In RandomSplitting every drawing style has its representative, which is magnified by the process of Bagging; however, in OrderedSplitting the data only represent the selected participants, with which the generated committee are overfitted with the style of the training participants. However as shown with the graph, the average accuracy tends to increase with the training sample, which indicates that with more training samples the drawing behaviours can be generalised. Overall the OrderedSplitting results are lower than RandomSplitting.

4.3.2. Random Forest (RF)

RandomForest is a mechanism which generates many trees and uses them to make classifications. Each tree is generated depending on a random vector, and they vote for the most popular class (Breiman, 2001). In WEKA, RandomForest is implemented as Bagging with random trees. We have validated this by performing the same test with RandomForest and Bagging with random trees, and found exactly the same result.

A random tree works similarly to normal trees; however, it does not split on the best feature, but randomly choose from a number of best features. This behaviour makes the generation less stable, allowing more variation in the generated trees, which is a desired property of the Bagging algorithm (Witten & Frank, 2005).

Basic Experiment

Table 5. RandomForest options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Debug	If set to true, classifier may output additional info to the console.	Bool
Max depth	The maximum depth of the trees, 0 for unlimited.	Int
Num features	The number of attributes to be used in random selection (see RandomTree).	Int
Num trees	The number of trees to be generated.	Int
seed	The random number seed to be used.	int

- Max depth (default: 0)

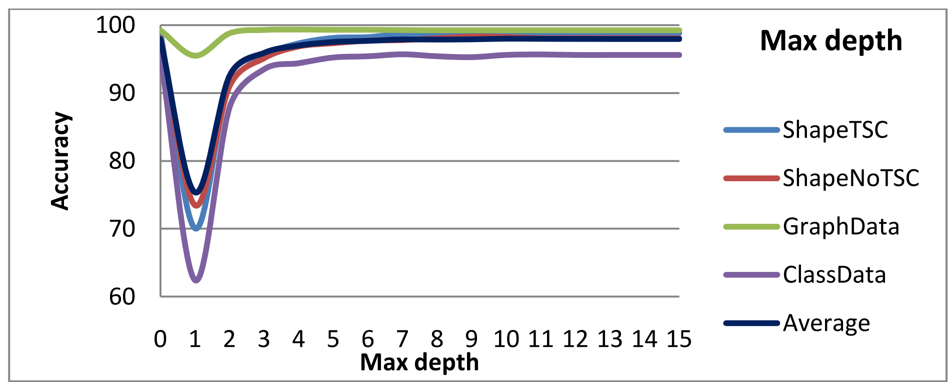


Figure 32. RandomForest: Max depth

The maximum depth indicates how deep each tree can grow. In the original implementation it is set to be unlimited, meaning a tree can have as many nodes as it needs, and no pruning is applied. Figure 32 shows the worst accuracy occurs when the depth is set to one, which means each tree has only one node. This essentially means we are randomly picking one attribute and use them to decide the accuracy, which does not give enough variation, and is unlikely to have correct recognition.

While the deeper each tree is the more detailed classification can be applied, it will also increase the chance of overfitting. However, in a voting mechanism, overfitting of individual cases can bring more variation, which may bring positive effect. This explains why the accuracy does not decrease after the max depth is increased. Interestingly, the depth of each tree need not be high; simple datasets such as GraphData can have high accuracy with only two levels, while around six can guarantee the performance of ShapeData and ClassData. The required depth certainly depends on the number of classes. However if the training performance is not an important issue, the default value, zero, which means no restriction in depth, works well.

- Num features (default: 0)

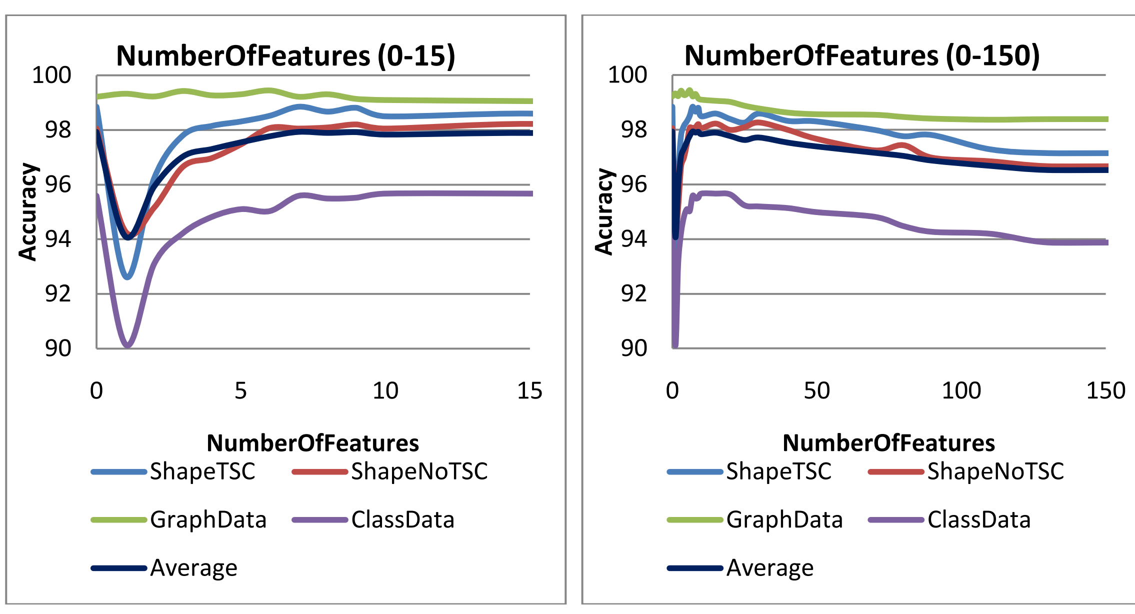


Figure 33. RandomForest: Number of Features

This setting decides the number of the best features to consider upon splitting a node. The lowest accuracy occurs at one, which means whenever splitting is required one feature will be selected randomly, in such case all generated trees will be the same, because they are all based on the same selection (the best feature) and no variation is introduced. Using the same tree to vote for an answer is not effective. Figure 33 demonstrates that for the early period, from 1-15, more features leads to better accuracy; this is because variations are introduced which captures different aspects of the better features. GraphData is a special case which has no significant drop in accuracy with only one feature selected. Such behaviour indicates the dataset is rather simple and one tree alone is capable to make good classifications.

The accuracy starts to decrease when ten or more features are used. A clearer trend can be observed when consider NumberOfFeatures(0-150) in Figure 33. This is because with more features, many of them can be unimportant or even bad features. While the variation generated by good features can benefit the voting process, misleading features introduce only noises. Thus, a balance needs to be found. The default value zero set the setting to base-2 logarithms of the number of attributes plus 1, which in our case equals seven. This dynamic decision returns the optimal performance.

- Num trees

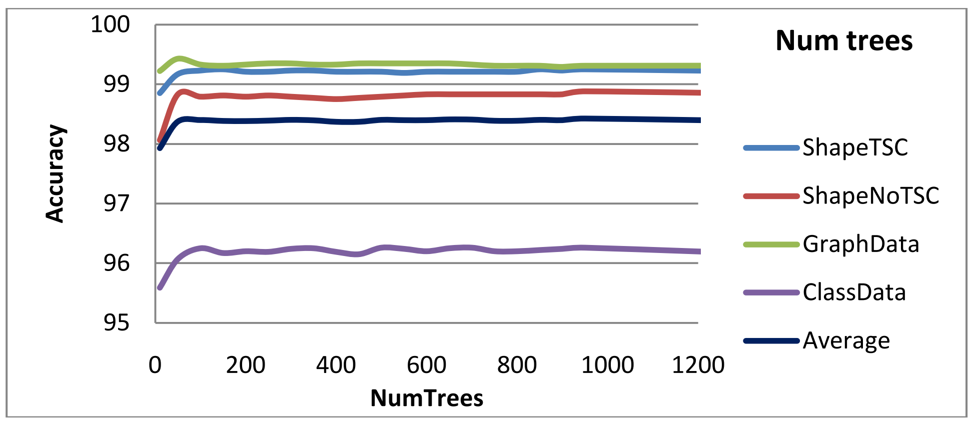


Figure 34. RandomForest: Number of Trees

To utilise the voting mechanism, more than one tree is required. Although more trees can increase the power of voting, the accuracy stabilises at a threshold where keep increasing the number of trees stops contributing to the accuracy. According to Figure 34 the accuracy of each dataset stabilises after 100 trees.

The lowest number for this experiment is ten trees. If ShapeData with TSC-features can be ignored, the level of improvement can almost be associated with the number of shape classes; the more classes the larger the improvement. This is because more relationships can be captured with more trees, and more complex problems contain more hidden relationships. TSC-features included ShapeData has a relatively smaller rise of accuracy, which shows these features can contribute to the accuracy with only a small number of trees required.

Optimise Experiment

WEKA Settings	Debug	Max depth	Num features	Num trees	Seed
Default	False	0	0	10	1
Opt (I:100)	False	0	0	100	1
Opt (I:400)	False	0	0	400	1

All changes are done in the number of trees used. Two versions of optimisation are used to compare their effects because they have similar performance.

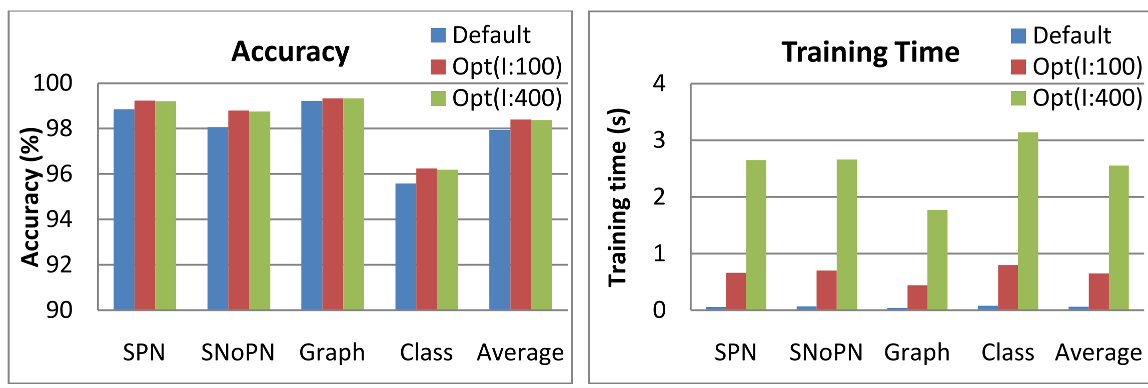


Figure 35. RandomForest: Optimise experiment

The accuracy is higher with 100 trees. Although with 400 trees there are still improvements compared with the default value, the accuracy actually decreased when compared with the 100 tree test.

RandomForest with 100 trees takes 11 times longer to train than the default setting which has only ten trees. The 400 tree version takes 42 times, which shows the increment in time is linear to the number of trees. Considering the improvement, we suggest using 100 trees since it does improve the performance and the time to train is acceptable. For the following experiments, the optimisation for RandomForest is fixed with 100 trees. The maximum testing time observed is 0.01 seconds, which is acceptable. All configurations can be safely applied in eager recognition.

Splitting Experiment

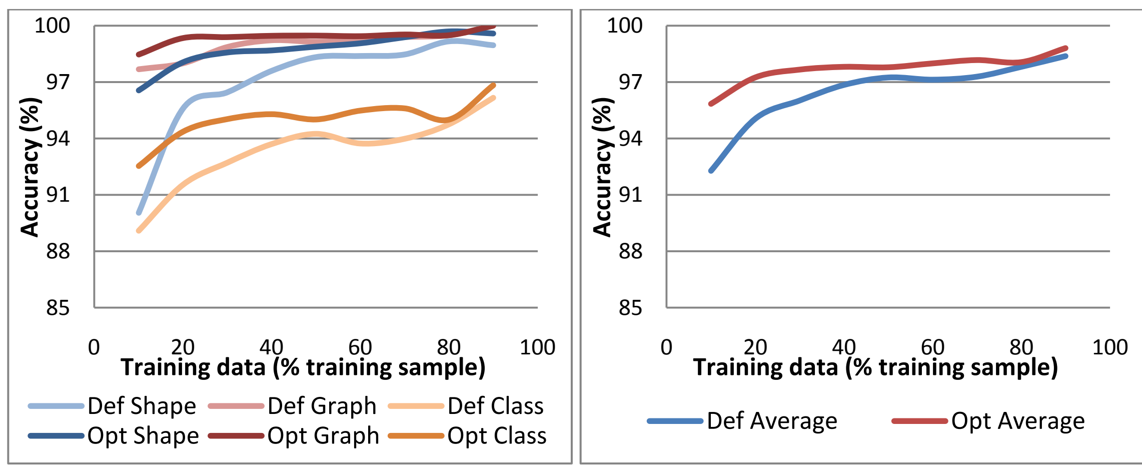


Figure 36. RandomForest: RandomSplitting

The RandomSplitting shows the optimisation generates better classifiers compared with the default one. It can make a 4% difference when the training examples are limited,

similar to Bagging. The trend is also similar to Bagging, however since the random tree is better than the default REP tree used by Bagging, the result is generally better.

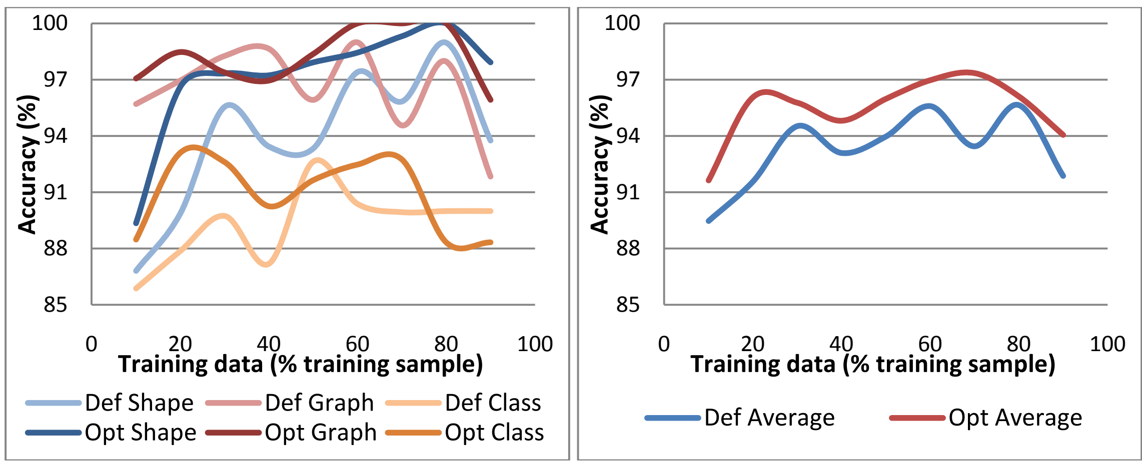


Figure 37. RandomForest: OrderedSplitting

Comparing with Bagging, the optimisation shows a better trend in general. In the OrderedSplitting situation, the optimised algorithm is strongly suggested. It improves the accuracy of the recogniser significantly in most situations. An interesting observation is that the average decreased with 90% data used. Because such behaviour cannot be found with RandomSplitting, it can only be explained by the fact that the last 10% data have styles which are not well captured by the previous ones.

4.3.3. LogitBoost (LB)

Similar to Bagging, boosting applies voting mechanisms to improve classifiers. However it tends to be more unstable (Witten & Frank, 2005). The generated classifiers can either be much better than the ones generated by Bagging, or overfit too much, which leads to complete failure. Furthermore, boosting is an iterative process. While Bagging generates new data and uses them to train different classifiers, boosting does the training iteratively. Each member of the voting committee is influenced by the ones generated previously, and they have different weights in voting.

The training process starts by assigning each training instance an equal weight. After the first round of classifier generation, the classifier will be used to classify the same instances. Each instance will be reweighted depending on the output: decrease the correctly classified ones, and increase the incorrectly classified ones. In the subsequent rounds, classifiers are built depending on the weight of instances – an instance with higher weight indicate that it is harder to classify, and vice versa – and will focus on the hard

ones. In other words, many classifiers will be generated, each with a different focus on the unsolved part of the problem. The weight of a classifier depends on its performance during the classification, a classifier with low error rate gives high weight, and vice versa. One thing to note is that if the error rate of a classifier exceeds 0.5, it will be deleted because it does not perform well. The occurrence of such a classifier indicates that classification is trying to recognise noises, which the algorithm will respond by terminating the iterations and return the built classifier.

Generally, a classifier is more accurate with more iteration. However such accuracy only appears on the training data; in the real world application overfitting can still occur. Boosting can turn simple algorithms into good models, as long as their starting error rate is lower than 50%. Among the many boosting methods, LogitBoost is the best choice for our problem domain, because it has better performance with multiple classes (Friedman, Hastie, & Tibshirani, 2000).

Basic Experiment

Table 6. LogitBoost options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Classifier	The base classifier to be used.	Other
Debug	If set to true, classifier may output additional info to the console.	Bool
Likelihood threshold	Threshold on improvement in likelihood.	double
Num folds	Number of folds for internal cross-validation (default 0 means no cross-validation is performed).	Int
Num iterations	The number of iterations to be performed.	Int
Num runs	Number of runs for internal cross-validation.	Int
Seed	The random number seed to be used.	Int
Shrinkage	Shrinkage parameter (use small value like 0.1 to reduce overfitting).	Double
Use resampling	Whether resampling is used instead of reweighting.	Boolean
Weight threshold	Weight threshold for weight pruning (reduce to 90 for speeding up learning process).	int

- Likelihood threshold (default: -Double.Max_Value)

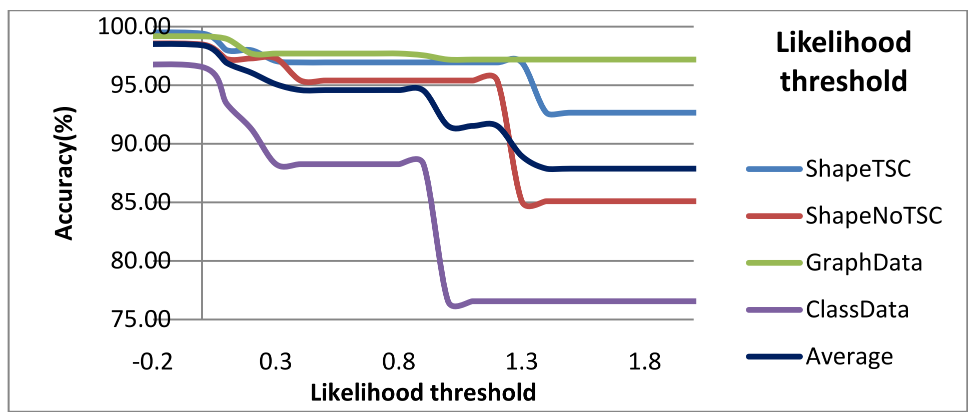


Figure 38. LogitBoost: Likelihood threshold

The performance of the classifier generated by boosting depends on how well it can model the problem, and this modelling ability comes from repeated iterations to generate trees which consider different situations. The aim of these iterations is to maximise the binomial log-likelihood.

This setting sets the threshold for the improvement of the average log-likelihood. During each iteration a new tree is generated, which improves the recognition rate on the unclassified samples. However since each iteration is operated on a smaller amount of data, the log-likelihood of the new tree is smaller. While each iteration improves the previous model on the weak parts, they will eventually overfit the model.

In WEKA, if the improvement in log-likelihood comparing with the previous tree is smaller than the threshold, iteration generation will be terminated. Default value is – Double.MAX_VALUE, which means no threshold is set (any negative number will have the same effect as there can only be positive improvement). According to Figure 38, setting this value will always decrease the accuracy, since the classifier creation is terminated before a full model is generated; as an advantage brought by this, the speed of each run will increase due to the absence of the would be iterations.

- Num folds (default: 0)

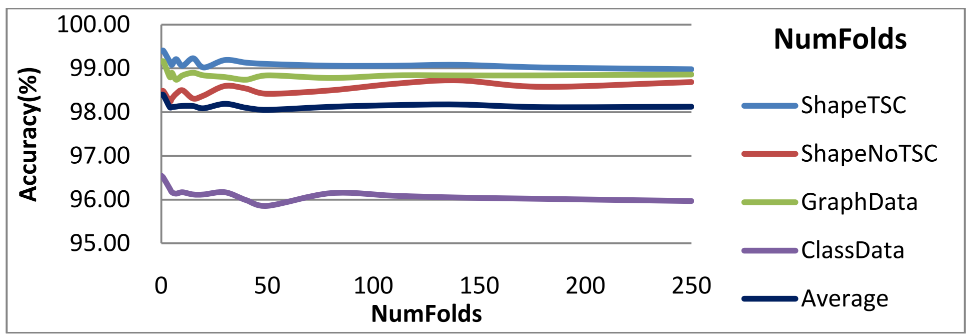


Figure 39. LogitBoost: NumFolds

Apart from the likelihood threshold, in the WEKA implementation another variable which decides the number of iteration to perform is the "bestNumIterations". NumFolds can be used to decide the bestNumIterations with cross validation, by looking at the number of cross validation which generates lowest error. The implementation only allows it to make a value smaller than the given number of iterations (which is set by NumIterations) for the bestNumIteration. Which means the experiment will be terminated before the default number of iterations is reached.

Hence, the same reason can be used to explain the drop in performance: because fewer iterations are performed. The default value indicates no cross validation is performed, which allows the experiment to be run fully.

- Num iterations (default: 10)

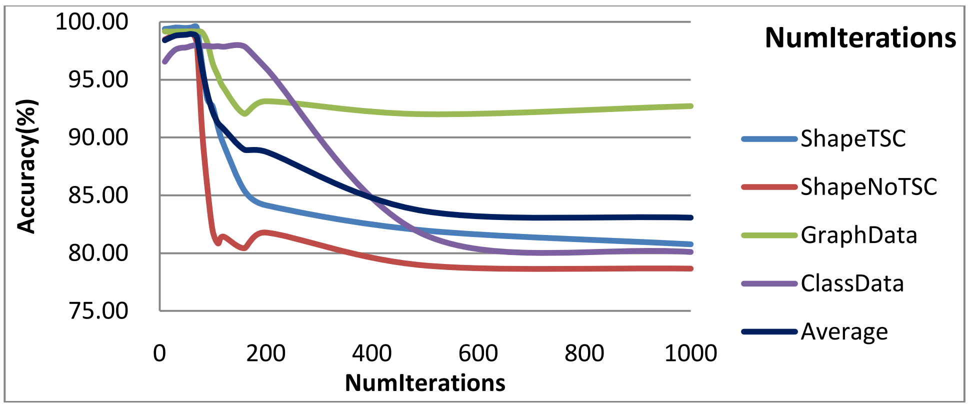


Figure 40. LogitBoost: NumIterations

This setting sets the number of iterations to run. Although the two settings explained have the ability to terminate the process before it completes, their default values disabled this ability. In this experiment the algorithm is run with exactly the number of iterations as set.

According to Figure 40, although the accuracy increased initially, there is obvious decrement after a certain threshold. The reason can be easily explained – it is due to overfitting. Hence, a complex diagram such as the class diagram in our dataset starts to drop later than all the others, because there are more features for it to learn before overfitting occurs.

- Shrinkage (default: 1)

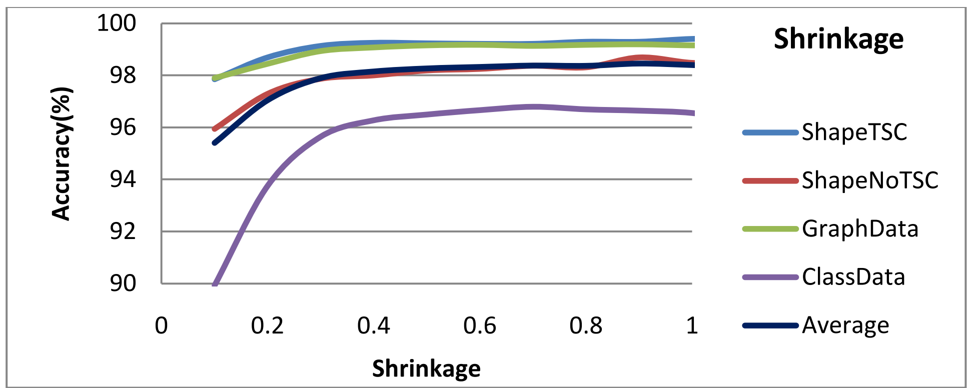


Figure 41. LogitBoost: Shrinkage

This setting "can be tuned to prevent overfitting" (Witten & Frank, 2005), because it affects the learning rate. Smaller value will reduce the speed of convergence, thus more iterations can be run. The suggestion of using values such as 0.1 may reduce overfitting, however the effect cannot be seen from this experiment. With our datasets, values between 0.5 and 1 are acceptable.

- Use resampling (default: false)

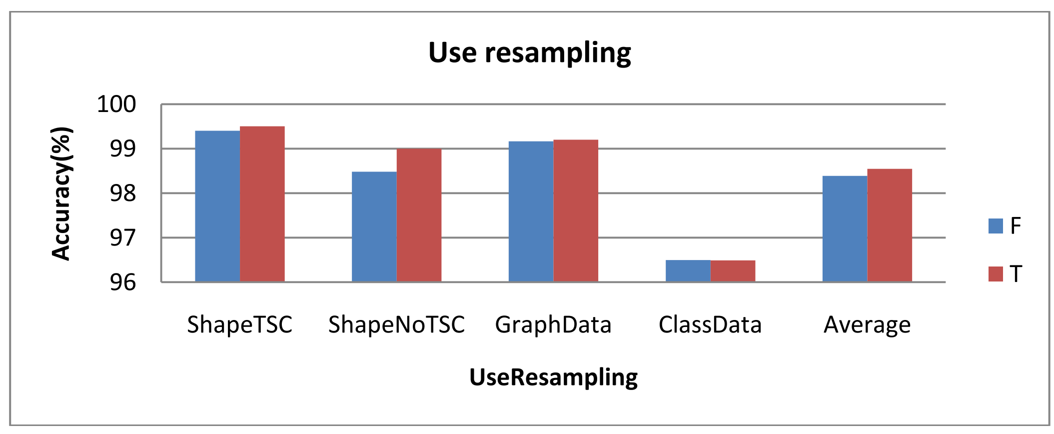


Figure 42. LogitBoost: UseResampling

Some classifiers may not support weighted values. This setting can resolve the situation by resampling the training set with weighted data to allow these classifiers to be boosted. Theoretically the process shall not affect the accuracy, however we can

see that the use of resampling improved the recognition rate in all situations, although only by a small amount. However, because we are not interested in applying base classifiers other than the default one, this is not included in the optimisation.

- Weight threshold (default: 100)

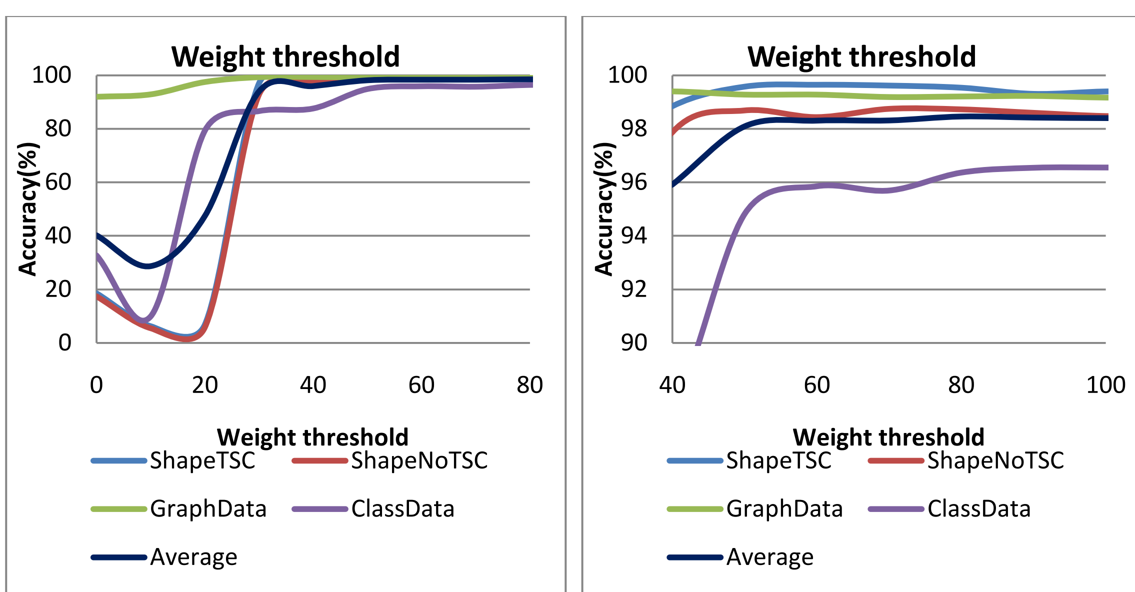


Figure 43. LogitBoost: Weight threshold

This setting decides the percentage of training data to be used in each iteration. The default value means all data are used, otherwise only the data which weights more than the threshold are used. The accuracy decreases if lower value is used, since only part of data is considered; however this accelerates the training process. According to the graph, applying more than 60% can give acceptable results. However, because some data are not used, this setting should not be changed except if the training time absolutely need be shortened.

- Ineffective settings

Classifier: The only other tree which can be used apart from the default decision stump is the REP tree, which did not perform as well. We believe the situation may be improved if resampling is applied, however for the same reason as in Bagging, this is not further explored.

Num runs: This value sets the internal number of runs for internal cross validation. Because by default there is no cross validation, setting this value does not have any effect at all. Although we can construct an experiment to see how this setting will help

the cross validation, because cross validation itself does not improve the accuracy, we decided to ignore this setting.

Optimise Experiment

WEKA Settings	Classifier	Debug	Likelihood Threshold	Folds	Iterations	Runs	Seed	Shrinkage	Resampling	Weight Threshold
Default	DecisionStump	F	-Double.Max_Value	0	10	1	1	1	F	100
Optimised	DecisionStump	F	-Double.Max_Value	0	70	1	1	1	F	100

Only the number of iterations provides significant improvement in the accuracy, which is taken into consideration. The value is set to 70 which is the maximum value in average.

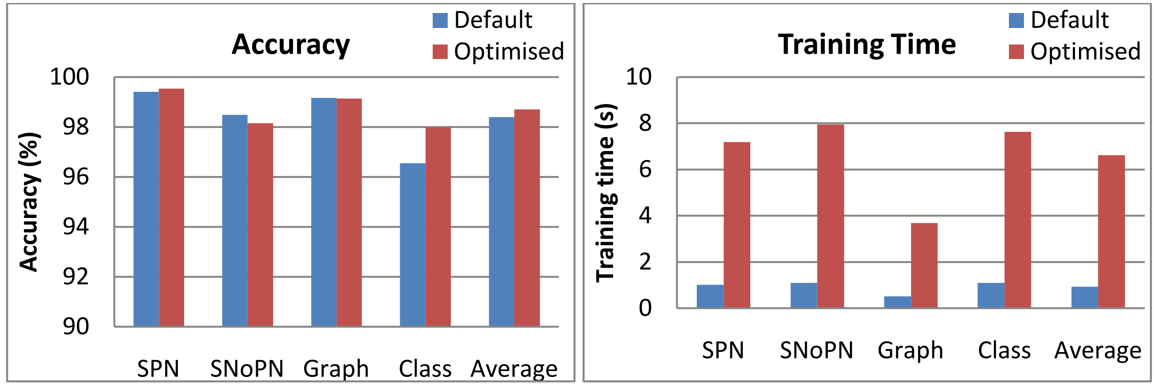


Figure 44. LogitBoost: Optimise experiment

ClassData is improved the most, while no significant difference for the other datasets. GraphData and ShapeData with no TSC-features even showed a reduction in accuracy. This may be due to overfitting, since the 70 iterations considered too many details. We believe the reason ClassData shows the greatest improvement is because it is more complex in both number of shapes and the way data is collected, and these are better modelled with more iterations. In other words, this optimisation is complexity dependent.

The maximum testing time observed is less than 0.01 seconds, thus all configurations can be applied in eager recognition. Training time has a linear relationship to the number of iterations, which is acceptable.

Splitting Experiment

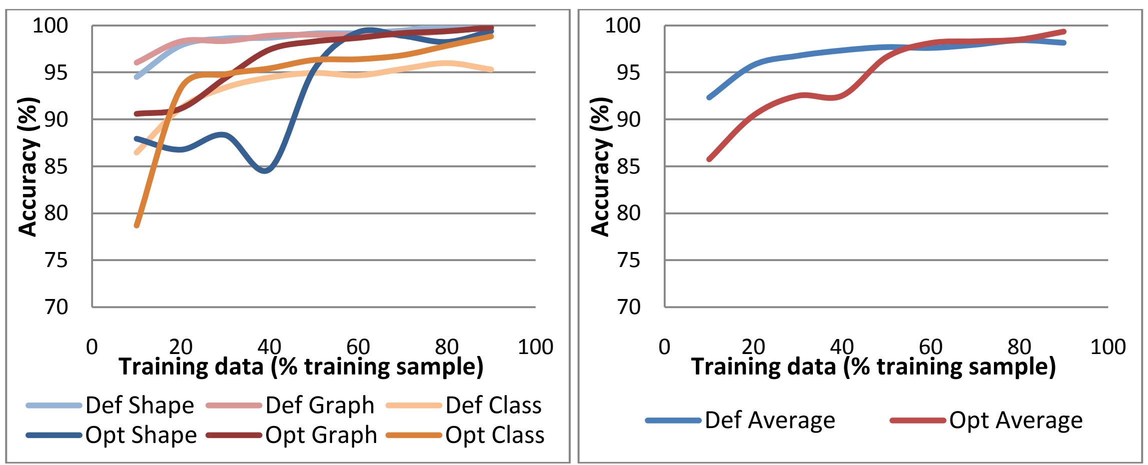


Figure 45. LogitBoost: RandomSplitting

The optimised result for RandomSplitting is bad at lower percentages. This behaviour is reasonable - applying a large number of iterations with only a small amount of data leads to overfitting. ShapeData is significantly overfitted, which is because they are simpler than other datasets; once the important relationships are learnt, the extra iterations are used in overfitting the model.

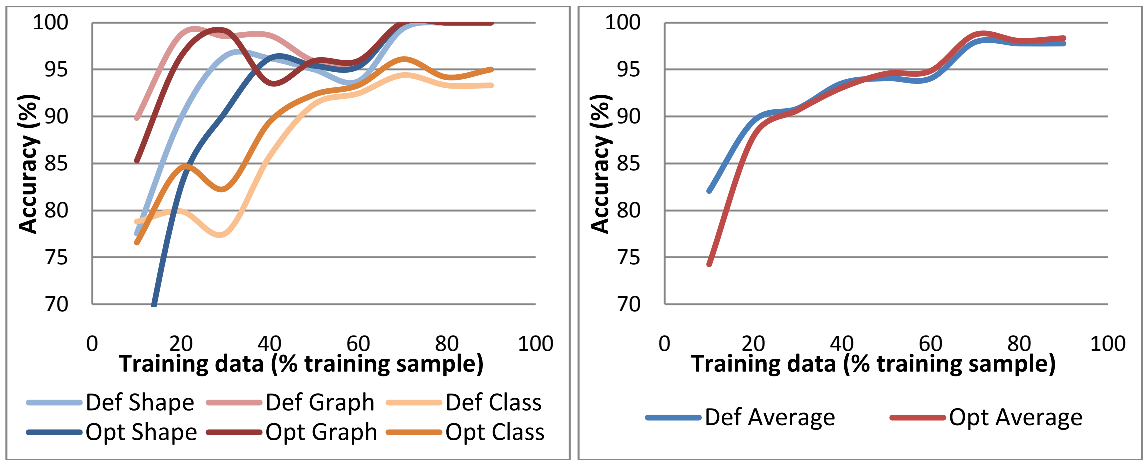


Figure 46. LogitBoost: OrderedSplitting

The behaviour is similar to RandomSplitting. The problem in ShapeData is even more significant; however, interestingly the optimised ClassData continually perform well, which shows that instead of overfitting, complex relationships are successfully modelled.

After these experiments we conclude the optimisation of LogitBoost has positive effects on the more complex datasets, however we decide not to apply it because the default is already performing well, and we are uncertain if the coming dataset is a complex one or

not. Since it is complexity dependent, dynamically adjust the setting would be very helpful.

4.3.4. Ensembles of Nested Dichotomies (END)

END is a voting system which combines many ND classifiers. A ND classifier can be considered as a binary tree. It deals with multi-class problems by reducing them into smaller two-class classification problems. With the same data, trees with different structures can be generated, because there is usually no real reason why one attribute should be split earlier than the others. These variations are thus all valid, although they vary in performance. Domain knowledge may contribute to the finding of better trees, however these forms of knowledge usually do not exist (Frank & Kramer, 2004).

Because all candidate NDs are valid and have strength in certain aspects, they are combined into ensembles. Due to the high growth rate of possible ND structures, an END uniting all possible ND is unrealistic. In the original paper (Frank & Kramer, 2004), the NDs are selected randomly. The resulting classification is the average of all NDs contained. They demonstrate that END outperforms C4.5 and logistic regression if they are directly applied to multi-class problems.

A deep tree takes more computation time than a balanced tree in the worst cases. To improve the performance, instead of choosing NDs randomly, they can be chosen only from balanced trees; this implementation "significantly improves runtime, with no significant change in accuracy" (Dong, Frank, & Kramer, 2005).

Basic Experiment

Table 7. END options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Classifier	The base classifier to be used.	Combo box
Debug	If set to true, classifier may output additional info to the console.	Bool
Num Iterations	The number of iterations to be performed.	Int
Seed	The random number seed to be used.	Int

- Classifier (default: ND)

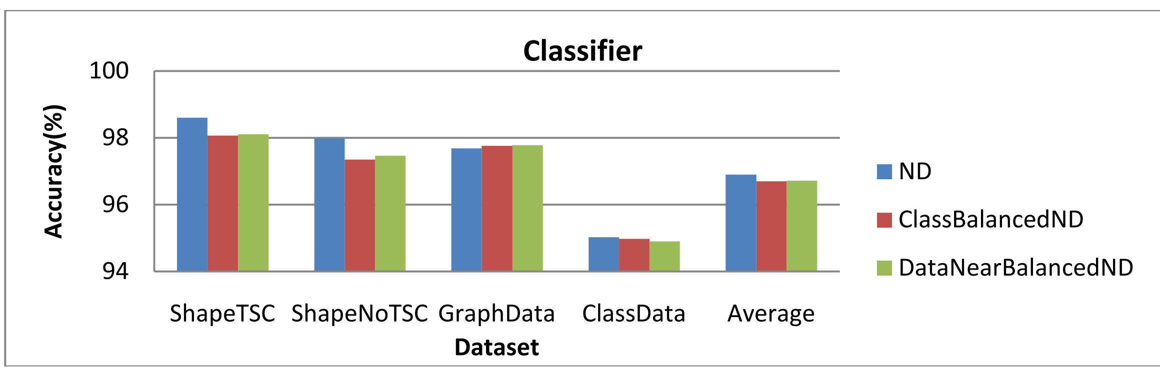


Figure 47. END: Classifier

This setting accelerates the process (Dong et al., 2005), however, the major focus is the improvements in accuracy. On average the ND has the best performance, but in GraphData it is outperformed by the other two classifiers. While this may indicate that in datasets with small number of classes the other classifier performs better, because in general ND has better performance, it is selected.

- Num Iterations (default: 10)

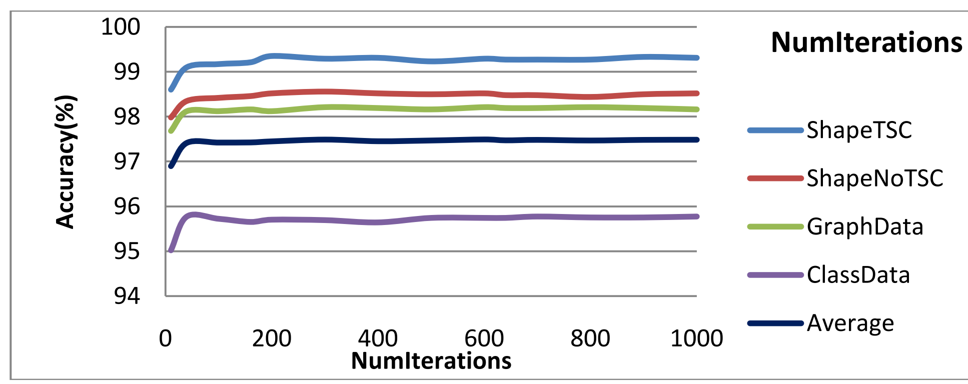


Figure 48. END:.NumIterations

A voting algorithm requires multiple trees to be effective. A new ND is generated with each iteration. Once the number of trees reaches a certain threshold, the change in accuracy will stabilise. Comparing this with the other algorithms in which certain dataset have different accuracy trends, all datasets seem to stabilise with the same number of iterations for END.

Optimise Experiment

WEKA Settings	Classifier	Debug	Iterations	Seed
Default	ND	F	10	1
Opt50	ND	F	50	1
Opt1000	ND	F	1000	1

Two optimised settings are tested, one used 50 iterations which is when the algorithms reach good performance, and another used 1000 iterations which is the average best performing setting.

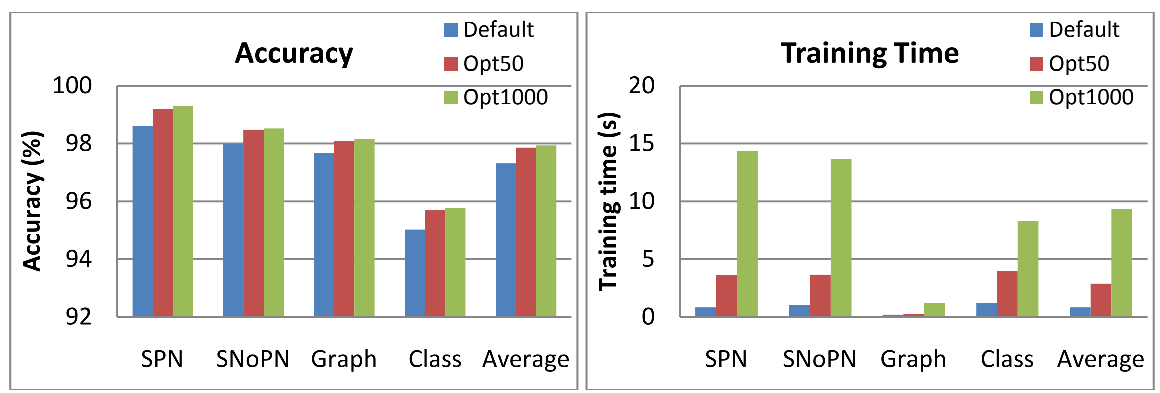


Figure 49. END: Optimise experiment

The accuracy for optimised algorithm increased for all datasets, while the training time is acceptable. Considering both figures in Figure 49, we believe the 50 iteration version is a better choice than the 1000 iteration version, because it is more cost effective.

Table 8. END testing time comparison

	ShapeData	GraphData	ClassData	Average
Default	0.00	0.00	0.00	0.00
Opt 50	0.02	0.00	0.02	0.01
Opt 1000	0.70	0.19	0.54	0.48

The optimisation can cause significant effects in END. According to Table 8, although the testing time of the default configuration would not affect the performance, differences can be observed with 50 iterations, and significant effect would be introduced if optimised with 1000 iterations. The effect is related to the number of targeting classes, however it requires 0.2 seconds for simple datasets such as GraphData, and even more for other datasets. This algorithm is hence inappropriate for eager recognition if a large number of iterations is required. On the other hand, because Figure 48 shows all classifiers stabilised at around 50 iterations, it may only require that much iteration for diagram recognition.

Because during the experiment END with 1000 iterations demonstrated better accuracy, it is selected to be further analysed in the following experiments.

Splitting Experiment

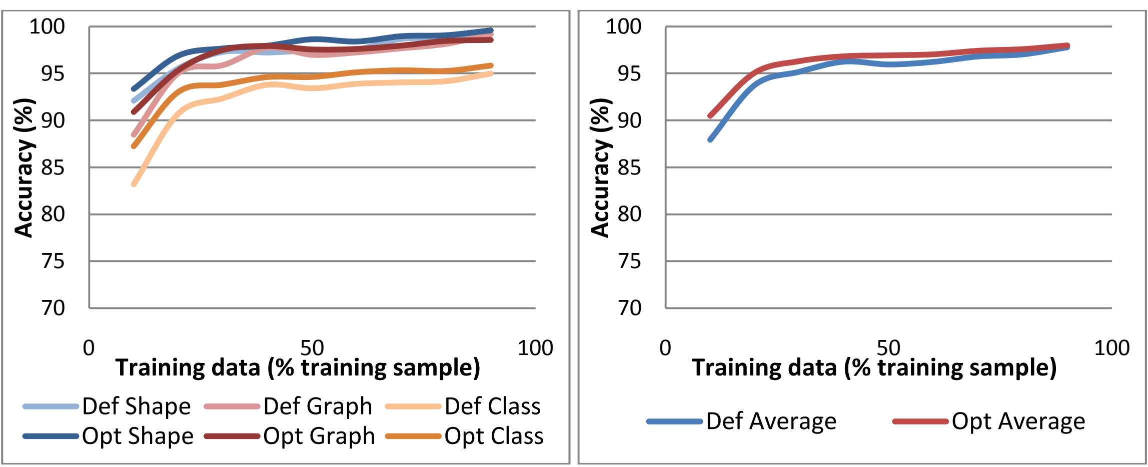


Figure 50. END: RandomSplitting

The optimised version performs better on most occasions. Figure 50 shows major improvement was achieved in ClassData, which is the most complex diagram. With the optimised version, the accuracy can reach 90% with only 10% testing data, which is very promising.

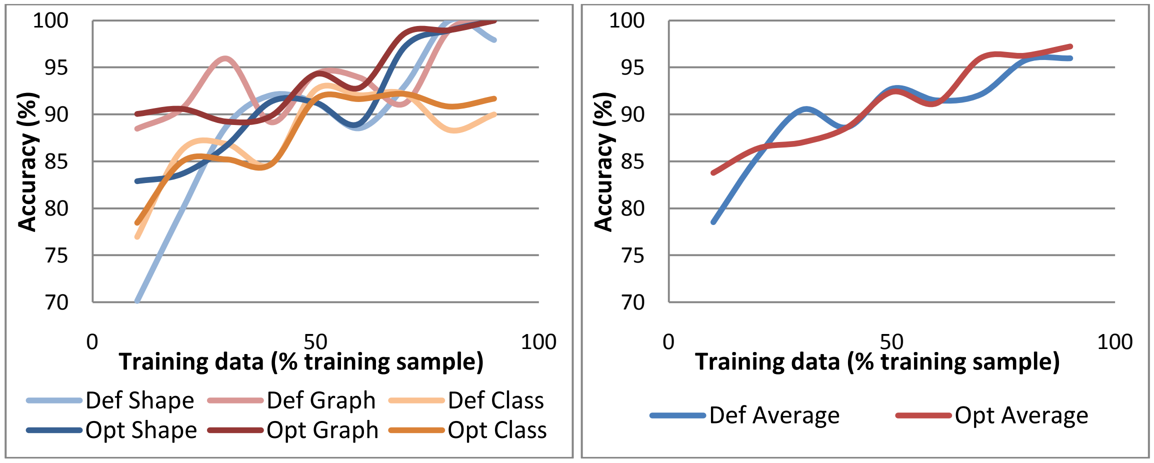


Figure 51. END: OrderedSplitting

The optimisation effect is less observable with OrderedSplitting. Although optimised END still performs better in general, it can also be significantly worse. At 30% OrderedSplitting this occurred for all three datasets and we can only assume that the behaviour is caused by the particular style of a participant which is easy to be captured by END but which is overfitting the data.

4.3.5. LogitBoost Alternating Decision Tree (LADTree)

Before the LADTree is introduced, consider the mechanism of ADTree first.

2002).

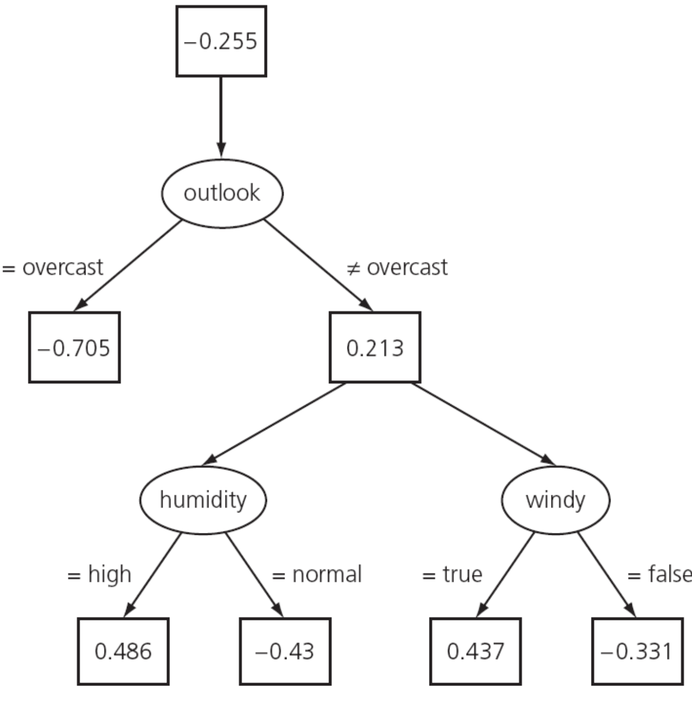


Figure 52. Example ADTree in WEKA(Witten & Frank, 2005)

ADTree is generally used in binary decision making; it is created to represent a collection of models in a more human readable fashion. It can be viewed as an AND/OR graph, or a sum of simple base rules (Holmes, Pfahringer, Kirkby, Frank, & Hall, 2002). An example is shown in Figure 52. It contains two kinds of nodes: prediction nodes (the rectangles) and splitter nodes (the ellipses). A splitter node splits instances into two pure subsets. A prediction node is the split result containing a value. To make predictions using an ADTree, we can simply filter the applicable prediction nodes, get the sum of their values, and decide the result based on whether the number is positive or not. The generation of ADTrees is normally done with boosting algorithms, but ADTrees are superior to boosting in that they can be merged together. This is suitable for use in multi-class situations.

The original generation of ADTrees is done via AdaBoost. However with LADTree, LogitBoost is used, which has two advantages. First, it natively supports multi-class situations. Second, it can be wrapped directly to numerical predictors, while AdaBoost "requires serious modification to the weak learner so that it can produce a separate prediction for each class value and also deal with class specific weights" (Holmes et al., 2002).

Basic Experiment

Table 9. LADTree options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Debug	If set to true, classifier may output additional info to the console.	bool
Num Of Boosting Iterations	The number of boosting iterations to use, which determines the size of the tree.	Int

- Num Of Boosting Iterations (default: 10)

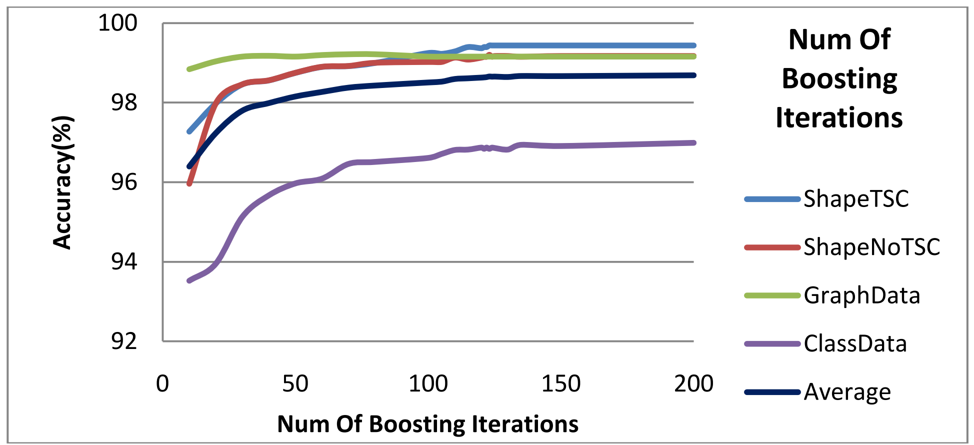


Figure 53. LADTree: Num of Boosting Iterations

Each iteration brings a set of new nodes to the tree – one split node and two prediction nodes. Hence, with only one iteration, the classification will only relay on that node, which leads to the low accuracy observed. More iteration brings more detail into the model, until the accuracy reaches a stable status.

Optimise Experiment

WEKA Settings	Debug	Boosting iterations
Default	F	10
Optimised	F	110

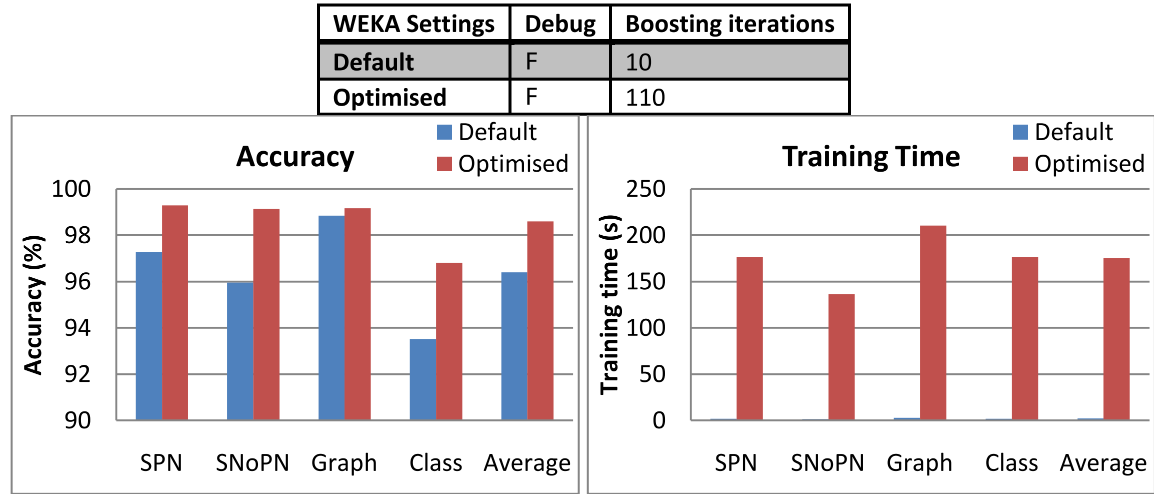


Figure 54. LADTree: Optimise experiment

The optimisation significantly affects the LADTree. Two percent improvement in accuracy can be found on average. However, the training time also increased significantly.

Since the maximum testing time observed in both cases is less than 0.01 second, we believe this optimisation is suitable to eager recognition. But user need to be aware of the long training required.

Complex diagrams are improved more significantly. While the improvement in GraphData is small, ClassData has accuracy improved over three percent. According to Figure 53, the accuracy of ClassData is still increasing with more iteration; hence we believe with more complex datasets, potentially even more iterations can be applied.

Splitting Experiment

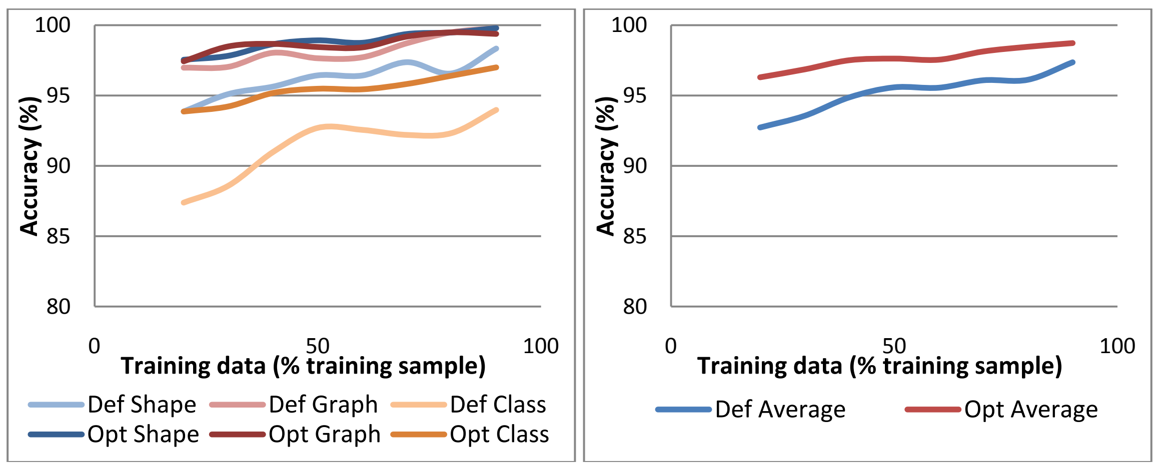


Figure 55. LADTree: RandomSplitting

RandomSplitting showed promising results; with the optimisation even 20% data can achieve over 95 % accuracy. However, RandomSplitting with 10% training data cannot be applied. Analysing the WEKA code we discovered that the problem occurred when the code tried to split but does not find a splitting node, which caused some unsigned data (m_search_bestPathInstances) to be used that produces null pointer exception. Because it only occurs in 10% RandomSplitting, we believe this is due to certain orders of the input data is not appropriately handled. Attempts to fix this were unsuccessful, hence the experiment started from 20%.

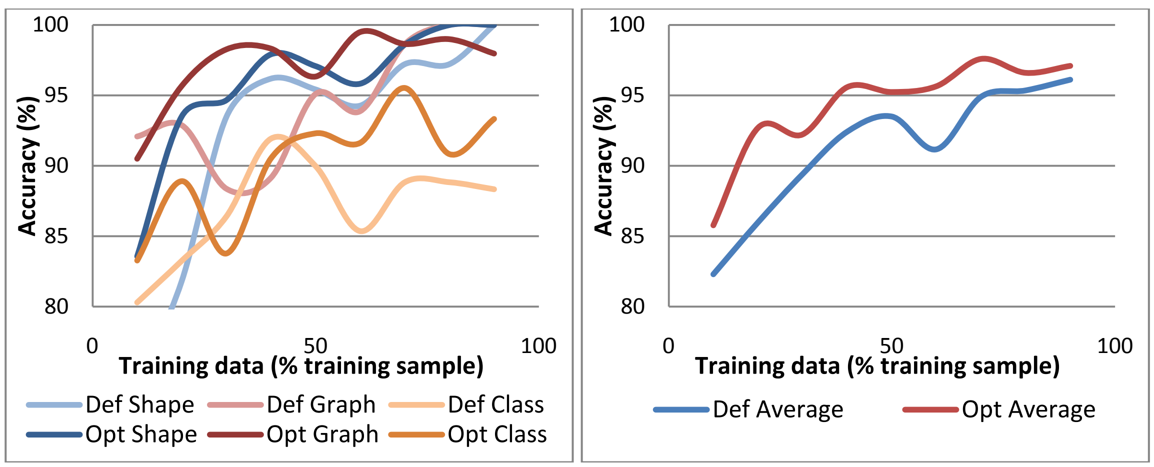


Figure 56. LADTree:OrderedSplitting

The optimised algorithm is generally better than the default ones. Generally with more training samples the optimised settings are performing better, however, at 90% OrderedSplitting, GraphData showed that the default setting is significantly better. We believe it is due to the overfitting because GraphData is relatively simple hence the tree started to learn the unimportant style information.

4.3.6. Logistic Model Trees (LMT)

Decision trees and regression methods are commonly applied in different problems. They have different strengths – trees tend to have low bias with high variance, and opposite behaviour can be found with regression methods (Landwehr, Hall, & Frank, 2005). LMT combines these two methods. Experiments have shown that LMT is more accurate than C4.5, CART and standalone logistic regression, and competitive with boosted C4.5 trees.

A LMT can be viewed as a tree with regression functions in each leaf. At each node of the tree, LogitBoost is applied to build a logistic regression function, with respect to the regression functions in parent nodes. Splitting occurs only if each split node has enough data to generate a logistic regression function. Once the tree is generated, CART algorithm is applied for pruning.

Basic Experiment

Table 10. LMT options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Convert nominal	Convert all nominal attributes to binary ones before building the tree. This means that all splits in the final tree will be binary.	Bool
Debug	If set to true, classifier may output additional info to the console.	Bool

Option Name	Option description in WEKA	Type
Error on probabilities	Minimize error on probabilities instead of misclassification error when cross-validating the number of LogitBoost iterations. When set, the number of LogitBoost iterations is chosen that minimizes the root mean squared error instead of the misclassification error.	Bool
Fast regression	Use heuristic that avoids cross-validating the number of Logit-Boost iterations at every node. When fitting the logistic regression functions at a node, LMT has to determine the number of LogitBoost iterations to run. Originally, this number was cross-validated at every node in the tree. To save time, this heuristic cross-validates the number only once and then uses that number at every node in the tree. Usually this does not decrease accuracy but improves runtime considerably.	Bool
Min num instances	Set the minimum number of instances at which a node is considered for splitting. The default value is 15.	Int
Num boosting iterations	Set a fixed number of iterations for LogitBoost. If >= 0, this sets a fixed number of LogitBoost iterations that is used everywhere in the tree. If < 0, the number is cross-validated.	Int
Split on residuals	Set splitting criterion based on the residuals of LogitBoost. There are two possible splitting criteria for LMT: the default is to use the C4.5 splitting criterion that uses information gain on the class variable. The other splitting criterion tries to improve the purity in the residuals produces when fitting the logistic regression functions. The choice of the splitting criterion does not usually affect classification accuracy much, but can produce different trees.	Bool
Use AIC	The AIC is used to determine when to stop LogitBoost iterations. The default is not to use AIC.	Bool
Weight trim beta	Set the beta value used for weight trimming in LogitBoost. Only instances carrying (1 - beta)% of the weight from previous iteration are used in the next iteration. Set to 0 for no weight trimming. The default value is 0.	double

- Error on probabilities (default: false)

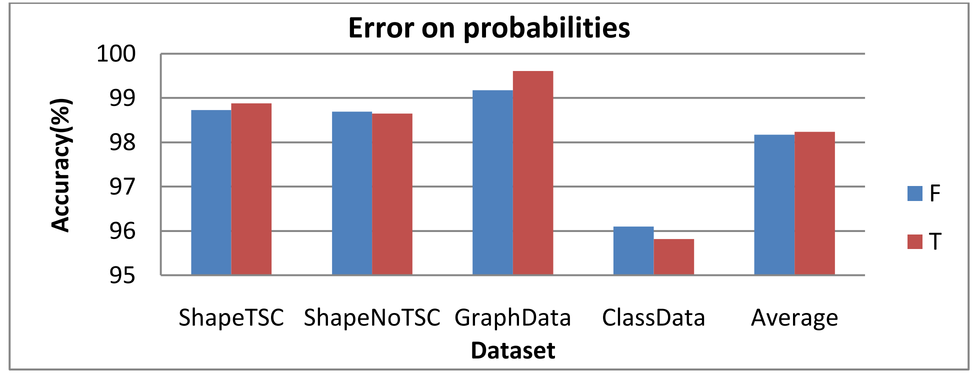


Figure 57. LMT: ErrorOnProbabilities

This setting allows users to select different kinds of error for the algorithm to minimise. Because the algorithms are focused on different perspectives, they will behave differently; however, the difference between them should be marginal. According to the result, the default setting, which minimises the misclassification error, is more suitable for complex data.

• Fast regression (default: true)

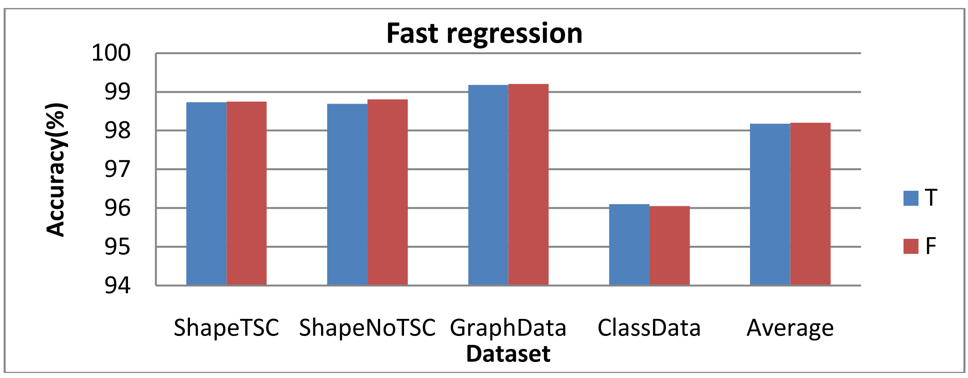


Figure 58. LMT:.FastRegression

The number of iterations for LogitBoost is decided with cross validation. Since LogitBoost is applied for each node, the cross validation process will need to be conducted at every node. Such a process is time consuming, thus by enabling this setting, the cross validation will be run only once, and the resulting number of runs will be used throughout the process.

No obvious difference in accuracy can be found, which suggests the default value, applying fast regression to save time is a better option.

• Min num instances (default: 15)

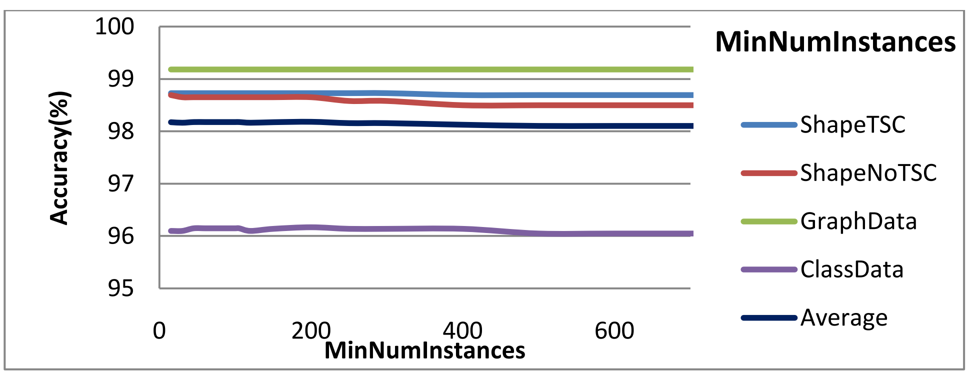


Figure 59. LMT: MinNumInstances

If instances in a node do not agree with each other, they are split further. However, if these instances are only a minor part of the whole dataset, they are likely to be noises which can cause overfitting. Hence, LMT allows users to set this attribute to decide the minimum instances a node must have to consider further splitting.

The default value is 15, however no significant performance change is observed with our experiment. The generated trees are also analysed, which shows the

modification of this attribute does not generate any difference in their structures. This is not the expected behaviour, because according to the specification the tree should not be growing if it is set higher than the number of training examples. As the average maximum performance occurs at 200, we selected this as the optimal setting.

- Num boosting iterations (default: -1)

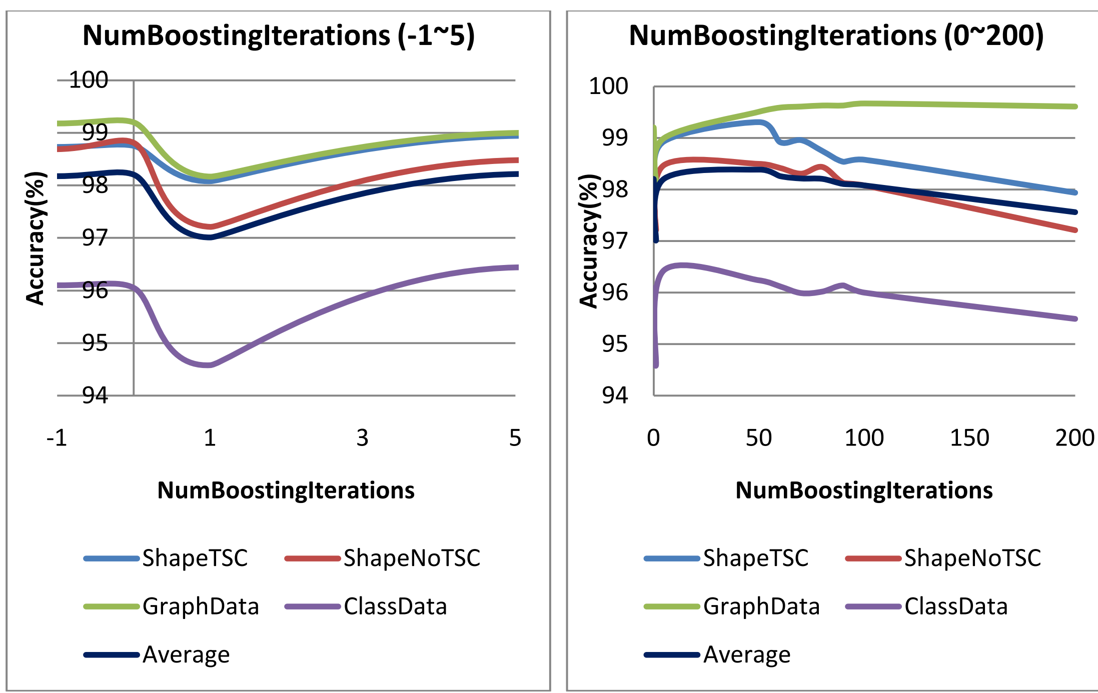


Figure 60. LMT: NumBoostingIterations

Values less than 1 indicate cross validation will be used to find the optimal number of iterations. Such a setting aims to allow the algorithm to find the best configuration suitable for the input data. However, according to the result we believe this default setting is not optimal. Furthermore the cross validation adds additional overhead to the training process.

Starting from 1, initially the performances are below the default setting. However the accuracy continues to improve, until a turning point is reached. According to the graph, complex datasets reach this turning point earlier. GraphData is the last to reach the turning, while ShapeData and ClassData display similar behaviour.

- Use AIC (default: false)

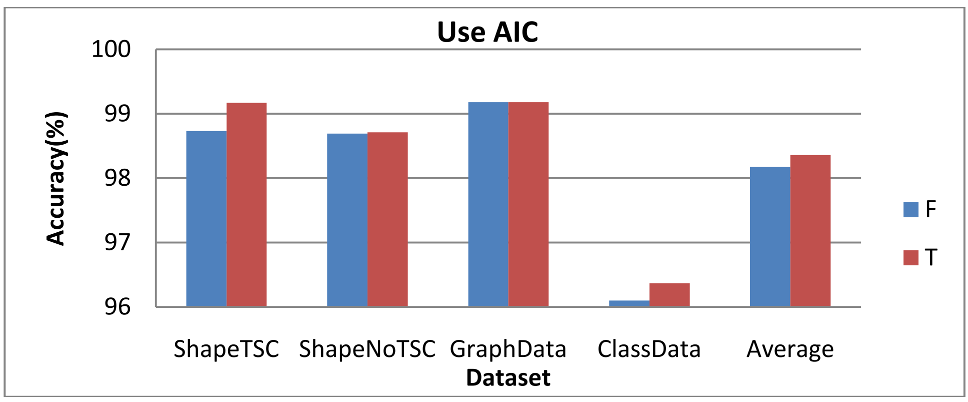


Figure 61. LMT: UseAIC

AIC is the abbreviation for Akaike information criterion, which is used for model selection, to find the fitness of a statistical model. It is used by LMT as a method to decide the best number of Logitboost iterations. It has a positive effect toward more complex models, but has no effect on GraphData which is relatively simple.

- Weight trim beta (default: 0)

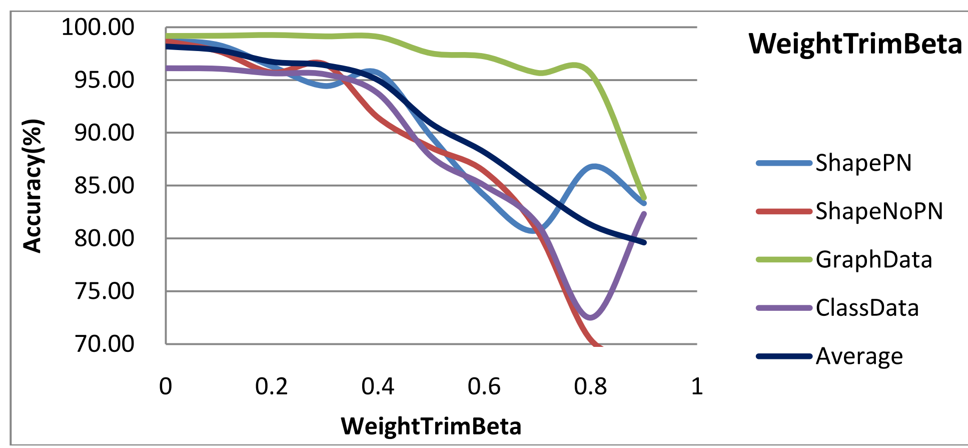


Figure 62. LMT: WeightTrimBeta

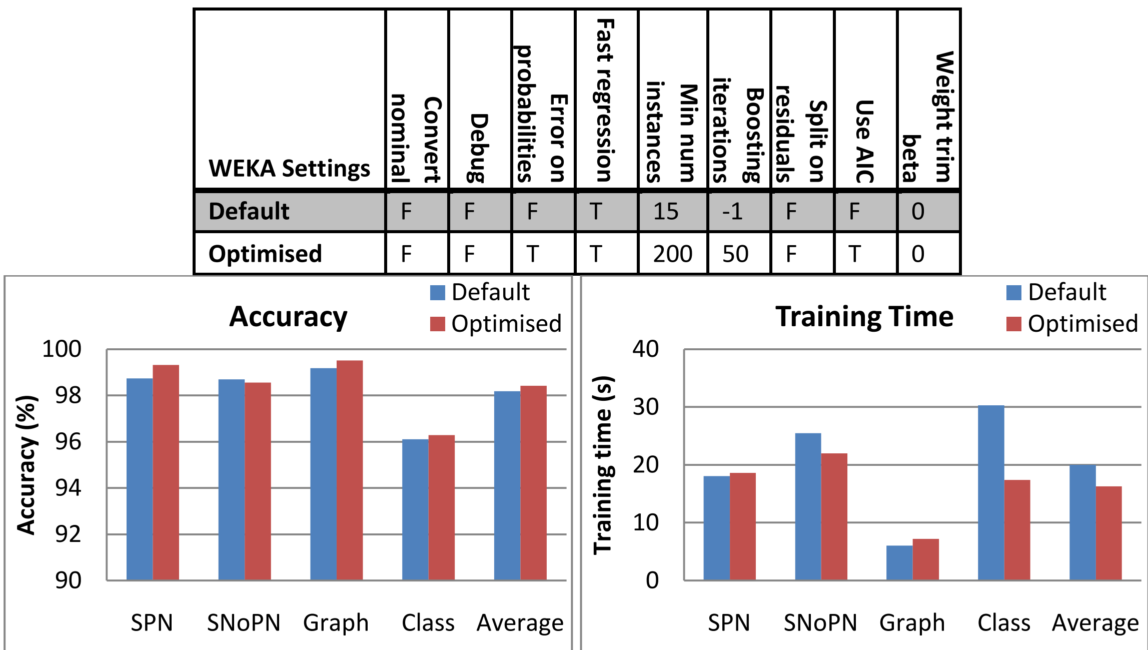
This setting accelerates the training by removing the instances which have low weight – which means they are already well classified. The default is zero, which means no weight trimming, and is the best choice in order to optimise the accuracy.

- Ineffective settings

Convert nominal: It converts every nominal attribute into sets of binary attributes. For example, considering a nominal attribute with five classes, this mechanism converts it into five binaries where only one is turned on each time. Since this format change does not change the data, it will not affect the accuracy.

Split on residuals: As described by WEKA, this setting lets user select how the tree splits. Although different trees may be generated, no significant difference appears in our experiment.

Optimise Experiment



WEKA Settings	Convert nominal	Debug	Error on probabilities	Fast regression	Min num instances	Boosting iterations	Split on residuals	Use AIC	Weight trim beta
Default	F	F	F	T	15	-1	F	F	0
Optimised	F	F	T	T	200	50	F	T	0

Figure 63. LMT: Optimise experiment

Most datasets have increased accuracy after optimisation, and the training time is similar to the default or even decreased. Hence we believe the optimisation is a good choice. The maximum testing time observed is 0s, which indicates all configurations can be safely applied in eager recognition.

Splitting Experiment

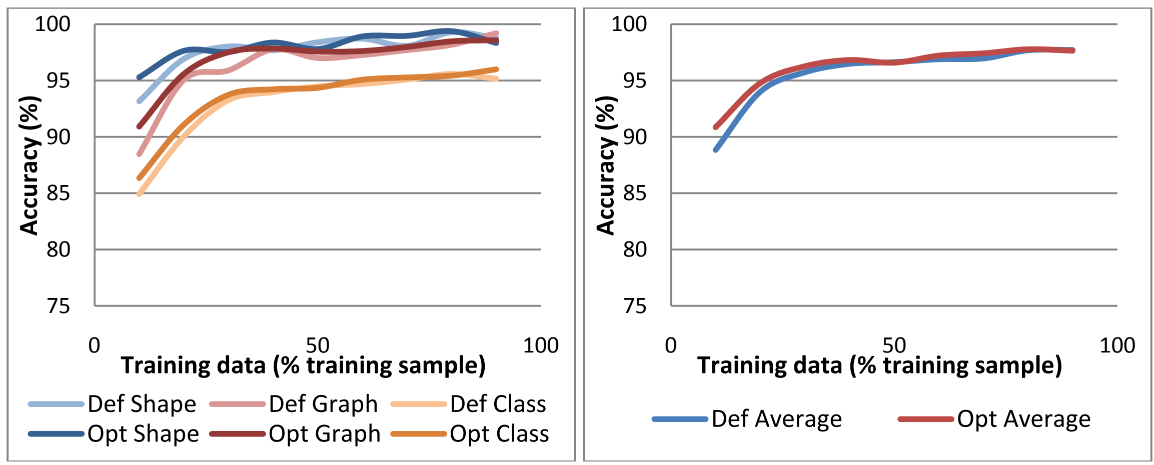


Figure 64. LMT: RandomSplitting

The optimised classifier is generally better, although not significantly. The difference in accuracy agreed with the result obtained from the 10 fold cross validation.

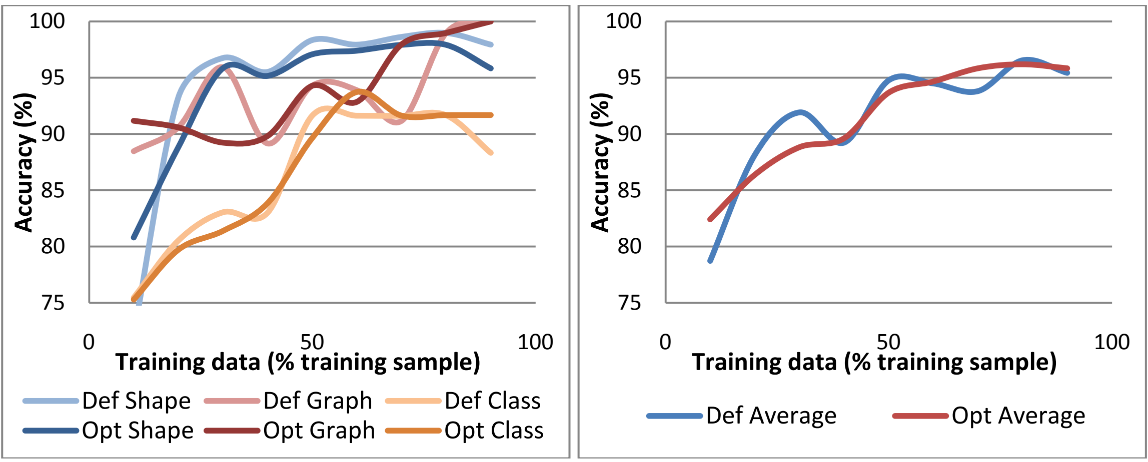


Figure 65. LMT: OrderedSplitting

In OrderedSplitting the optimised LMT appears to be more stable, however the default setting occasionally outperforms it. These occasions are usually caused by the directed dataset. Overall, the result of OrderedSplitting does not indicate a successful improvement in the optimised settings.

4.3.7. Bayesian Networks (BN)

Bayesian Networks is one type of probabilistic graphical model (Ben-Gal, 2007). It combines probabilities from many sources to form a model which can be used to classify the result. It has the directed acyclic graph (DAG) structure. A DAG is defined with nodes and directed edges; a node represent random variables, and a directed edge defines the dependence of the connected ancestors and descendants. As it is an acyclic graph, no direct circle is allowed, which means there exists no node which is the ancestor of itself. Another important element for Bayesian Networks is the conditional probability tables (CPTs). Each node has a CPT, which encapsulates the probabilities of the node with consideration of the status of its parents. Given a trained Bayesian network, one can input the variables, and the probability of each ancestor node will affect the probability of their direct children, until the possibility of the targeting variable is computed.

Basic Experiment

Table 11. Bayes network options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
BIF File	Set the name of a file in BIF XML format. A Bayes network learned from data can be compared with the Bayes network represented by the BIF file. Statistics calculated are o.a. the number of missing and extra arcs.	String (file location)
Debug	If set to true, classifier may output additional info to the console.	Boolean
Estimator	Select Estimator algorithm for finding the conditional probability tables of the Bayes Network.	Combo box
Search Algorithm	Select method used for searching network structures.	Combo box
Used ADTree	When ADTree (the data structure for increasing speed on counts, not to be confused with the classifier under the same name) is used learning time goes down typically. However, because ADTrees are memory intensive, memory problems may occur. Switching this option off makes the structure learning algorithms slower, and run with less memory. By default, ADTrees are used.	boolean

- Search Algorithm (default: K2)

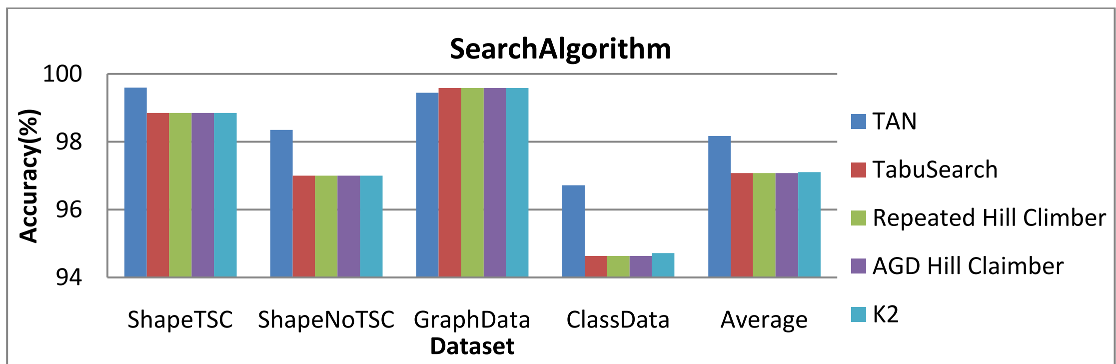


Figure 66. Bayes Network: SearchAlgorithm

This setting affects how the structure of the network is decided. Compared with K2 the default algorithm, TAN performs much better in general and it is selected as the optimised search algorithm.

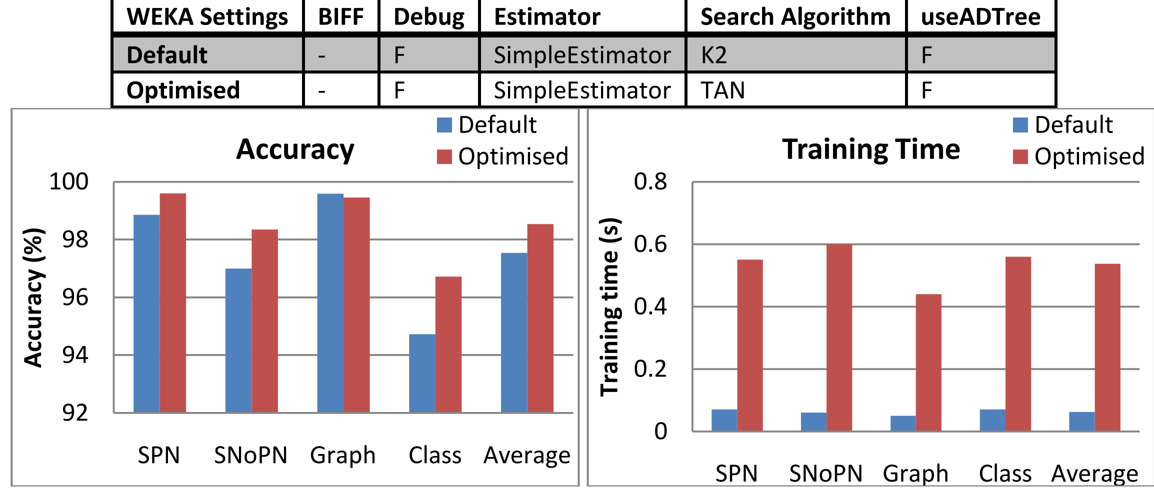
- Ineffective settings

BIF File: A BIF file contains information about the structure of a trained Bayesian network. If a BIF file is given WEKA will compare the structure of that file with the structure of the generated one, and return the statistics. This is not used because we are not interested in analysing the generated network.

Estimator: Different estimator algorithms may have different effect on generating CPTs; however among the given estimators, only the default simple estimator can work with our data.

Used ADTree: According to Table 11, this option accelerates the training process. However we have to use ADTree due to memory problems – otherwise, the memory usage will exceed 1GB which is the maximum memory that can be assigned to WEKA by our machine used in the study.

Optimise Experiment



WEKA Settings	BIFF	Debug	Estimator	Search Algorithm	useADTree
Default	-	F	SimpleEstimator	K2	F
Optimised	-	F	SimpleEstimator	TAN	F

Figure 67. Bayes Network: Optimise experiment

The more complex a dataset is the more it is improved. The maximum testing time observed is less than 0.01 seconds, thus all configurations can be safely applied in eager recognition. Although training time is increased, compared with other algorithms the training time of BayesianNetwork is relatively small. We believe the optimisation is a valid choice.

Splitting Experiment

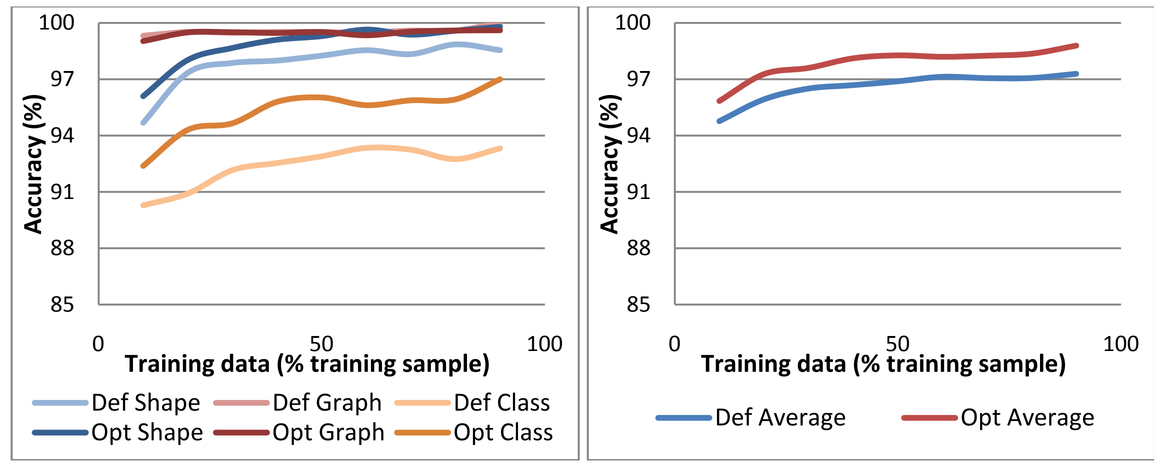


Figure 68. Bayes Network: RandomSplitting

The behaviour agreed with the 10 fold cross validation experiment; while the simple dataset display little change in accuracy, significant differences can be found with ShapeData and ClassData.

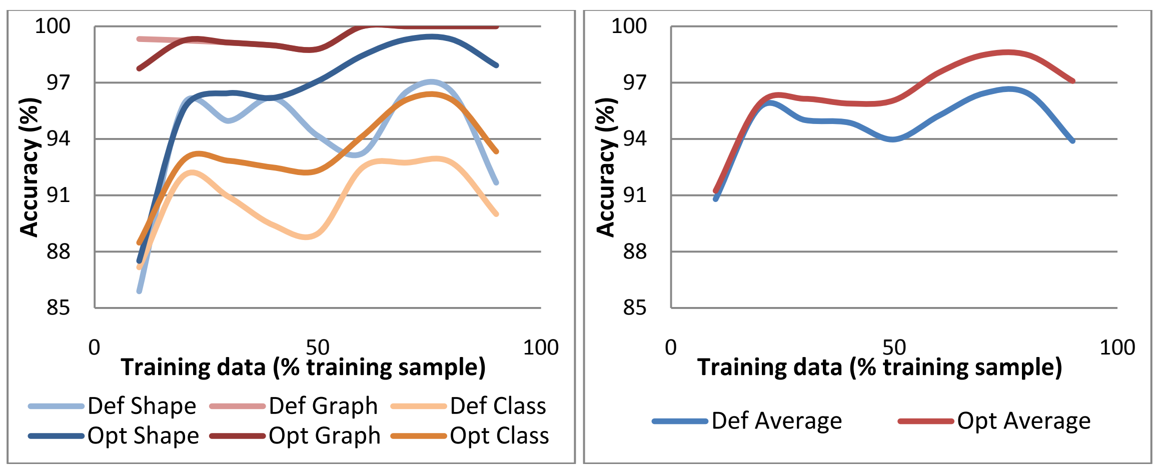


Figure 69. Bayes Network: OrderedSplitting

The same observation can also be found with OrderedSplitting. Although with a small amount of training examples the behaviour is not apparent, significant differences appear when more data are given. The optimisation of BayesianNetwork is strongly suggested.

4.3.8. Multilayer Perceptron (MLP)

MultilayerPerceptron is an artificial neural network model, which can represent nonlinear classifications. It can represent any expression from propositional calculus. The classification is done by sending attributes into a network of perceptrons. The structure of network and the weight of connection will cause the final perceptrons to be in different states for different input attributes.

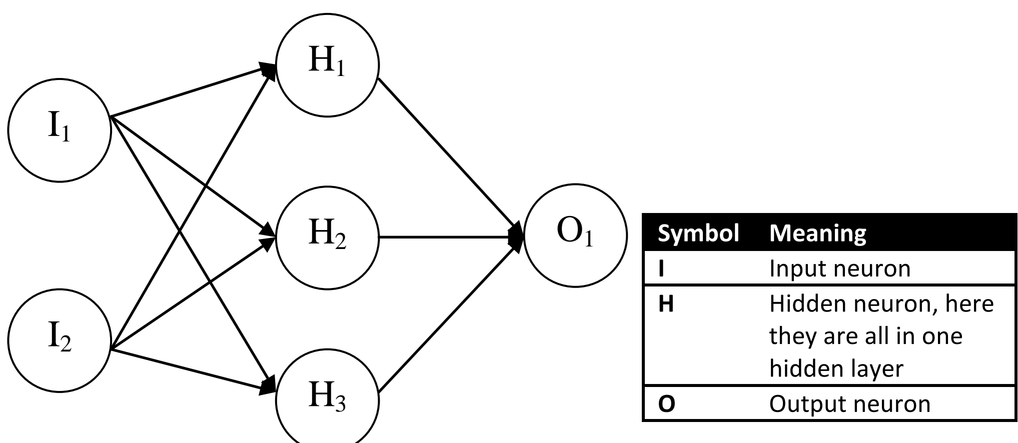


Figure 70. Example artificial neuron network

One standard method of training the artificial neural network is with back-propagation, in which the weights of connectors are adjusted based on the difference between desired output and real output (Witten & Frank, 2005).

Basic Experiment

Table 12. MultilayerPerceptron options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Autobuild	Adds and connects up hidden layers in the network.	Bool
Debug	If set to true, classifier may output additional info to the console.	Bool
Decay	This will cause the learning rate to decrease. This will divide the starting learning rate by the epoch number, to determine what the current learning rate should be. This may help to stop the network from diverging from the target output, as well as improve general performance. Note that the decaying learning rate will not be shown in the gui, only the original learning rate. If the learning rate is changed in the gui, this is treated as the starting learning rate.	Bool
Hidden layers	This defines the hidden layers of the neural network. This is a list of positive whole numbers. 1 for each hidden layer. Comma seperated. To have no hidden layers put a single 0 here. This will only be used if autobuild is set. There are also wildcard values 'a' = (attribs + classes) / 2, 'i' = attribs, 'o' = classes , 't' = attribs + classes.	Int/ string
Learning rate	The amount the weights are updated.	Double
Momentum	Momentum applied to the weights during updating.	Double
Nominal to binary filter	This will preprocess the instances with the filter. This could help improve performance if there are nominal attributes in the data.	Bool
Normalize attributes	This will normalize the attributes. This could help improve performance of the network. This is not reliant on the class being numeric. This will also normalize nominal attributes as well (after they have been run through the nominal to binary filter if that is in use) so that the nominal values are between -1 and 1	Bool
Normalize numeric class	This will normalize the class if it's numeric. This could help improve performance of the network, It normalizes the class to be between -1 and 1. Note that this is only internally, the output will be scaled back to the original range.	Bool
Reset	This will allow the network to reset with a lower learning rate. If the network diverges from the answer this will automatically reset the network with a lower learning rate and begin training again. This option is only available if the gui is not set. Note that if the network diverges but isn't allowed to reset it will fail the training process and return an error message.	Bool
Seed	Seed used to initialise the random number generator. Random numbers are used for setting the initial weights of the connections between nodes, and also for shuffling the training data.	Int
Training time	The number of epochs to train through. If the validation set is non-zero then it can terminate the network early	Int
Validation set size	The percentage size of the validation set.(The training will continue until it is observed that the error on the validation set has been consistently getting worse, or if the training time is reached).	Int
Validation threshold	Used to terminate validation testing. The value here dictates how many times in a row the validation set error can get worse before training is terminated.	int

- Decay (default: false)

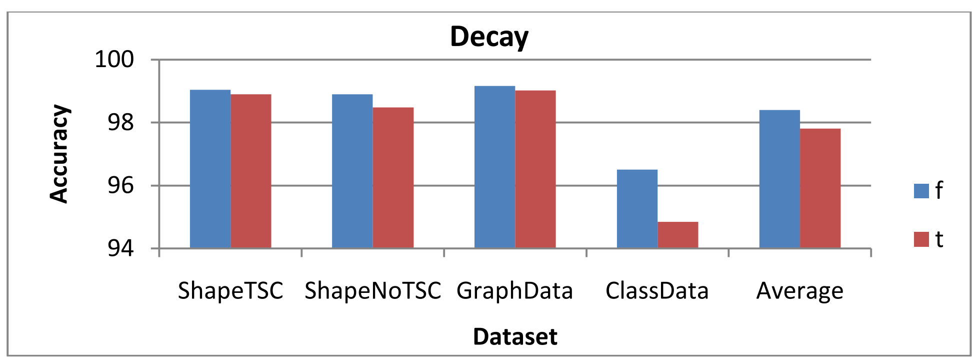


Figure 71. MultilayerPerceptron: Decay

Decay decides the learning rate with consideration to the input data. While WEKA suggested the increment in accuracy (Table 12), the behaviour cannot be found with our experiment; furthermore this setting can increase the training time, it is not recommended for diagram recognition.

- Hidden layers (default: 'a' = (attribs + classes) / 2)

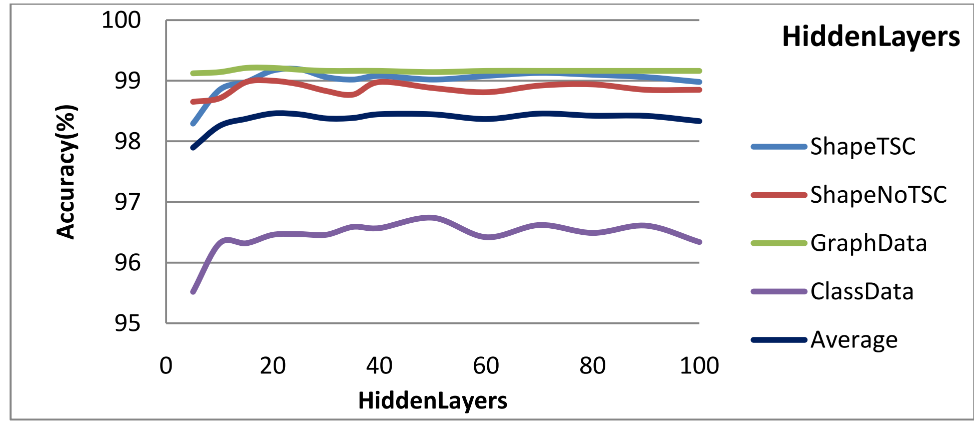


Figure 72. MultilayerPerceptron: HiddenLayers

According to the data we can see that generally ten or more hidden layers are enough for MultilayerPerceptron to perform well. It appears to be complexity dependent, as we may see the maximum accuracy for GraphData occurred at 20, ShapeData was slightly after 20 while ClassData was at 50. WEKA provides several wildcards for dynamic decision based on the number of features. However according to the data these wildcards do not return the optimal result; furthermore we found if the number of hidden layers becomes too large, WEKA stops loading them. The final decision is made to fix the value at 40, which is the maximum number the experiment machine can handle.

- Learning rate (default: 0.3)

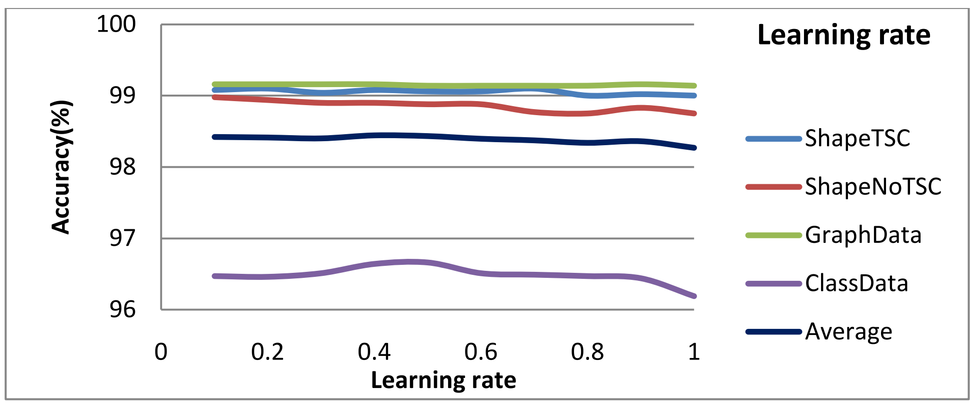


Figure 73. MultilayerPerceptron: LearningRate

The learning rate defines how much weight is updated in each connector each time. A small learning rate is inefficient, requiring a long time to train, while a large learning rate would emphasis the impact of noises. The result suggests the learning rate should be below 0.6. The result also shows the more complex datasets are affected more with the high learning rate, which is caused by the more noises existing within them.

- Momentum (default: 0.2)

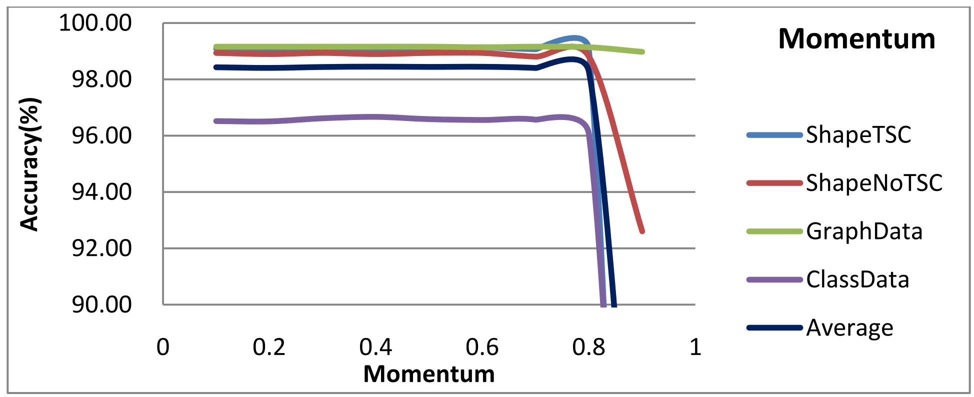


Figure 74. MultilayerPerceptron: Momentum

The momentum value can smooth the weight changing process. A large momentum causes the weight to be updated rapidly and vice versa. This can explain the decrement with momentum over 0.8 – the insignificant training styles can cause the weight to shift largely and cause overfitting.

- Nominal to binary filter (default: true)

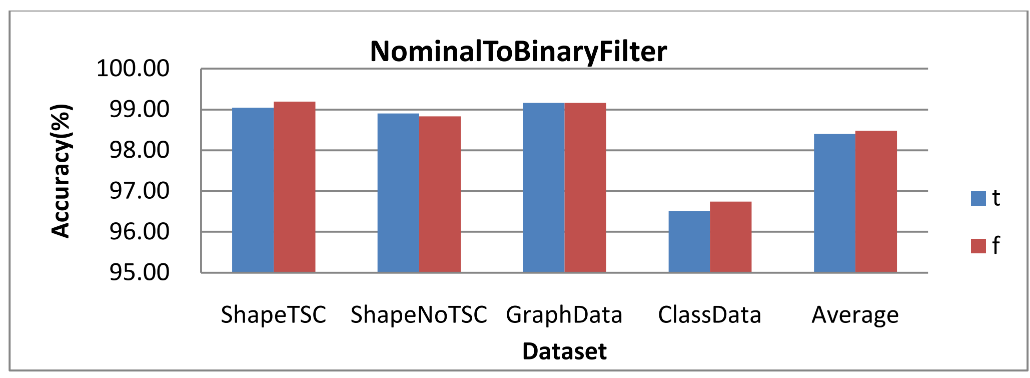


Figure 75. MultilayerPerceptron: NominalToBinaryFilter

While this is intended to accelerate the process, it also affects the accuracy measure. The result shows that turning it off can result in higher accuracy.

- Normalise attributes (default: true)

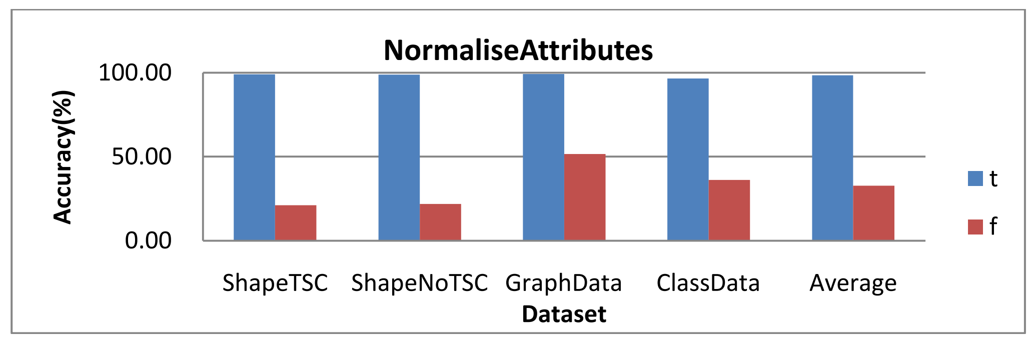


Figure 76. MultilayerPerceptron: NormaliseAttributes

According to the large variation in the resulting accuracies, the default setting, allowing normalisation, is obviously the better choice.

- Training time (default: 500)

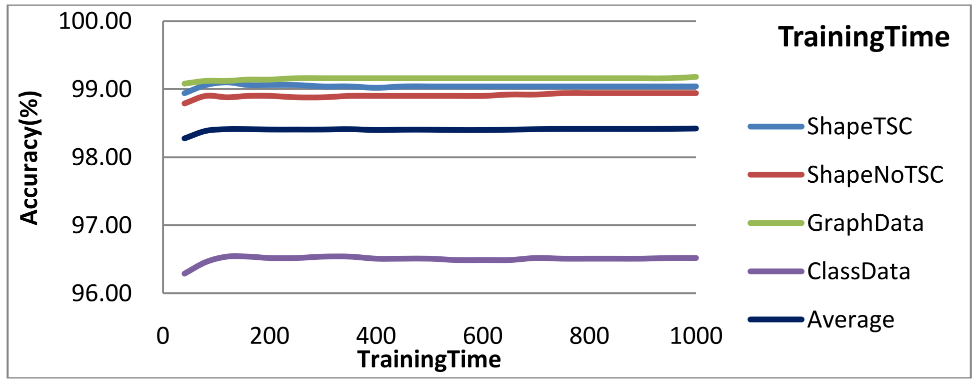


Figure 77. MultilayerPerceptron: TrainingTime

A certain amount of training time is required to achieve acceptable accuracy. The default value 500 is larger than required, however since not much difference will be made in changing it, we decided to leave it as default. The effect appears to be complexity dependent, for example while little effect can be seen in GraphData, ClassData demonstrated more change in accuracy and a longer span in the changing process.

- Validation set size (default: 0)

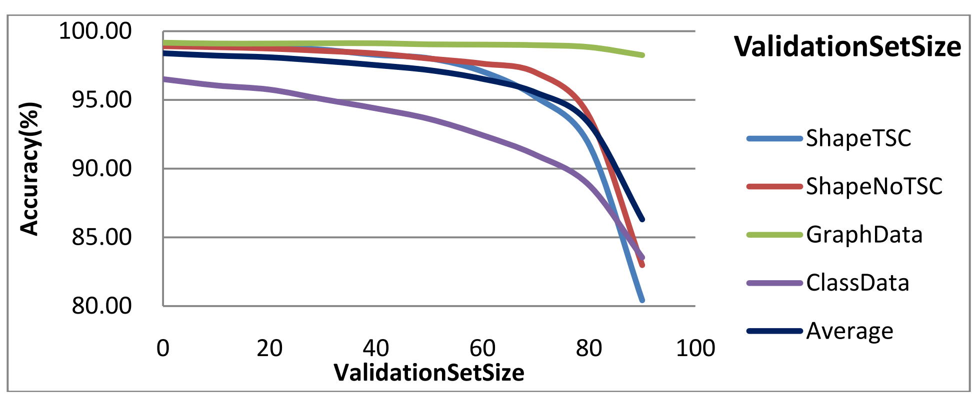


Figure 78. MultilayerPerceptron: ValidationSetSize

Higher value indicates more tolerance toward the error, which can explain why the accuracy decreases with the increased validation set size – because more errors are allowed. The default value has the best accuracy measure.

- Ineffective settings

Settings including Normalise numeric class, Reset and Validation threshold had no impact on accuracy in the experiments.

Optimise Experiment

WEKA Settings	Decay	Hidden layers	Learning rate	Momentum	Nominal to binary	Normalize attributes	Normalize numeric class	Reset	Seed	Training time	Validation set size	Validation threshold
Default	F	a	0.3	0.2	T	T	T	T	0	500	0	20
Optimised	F	40	0.4	0.4	F	T	T	T	0	500	0	20

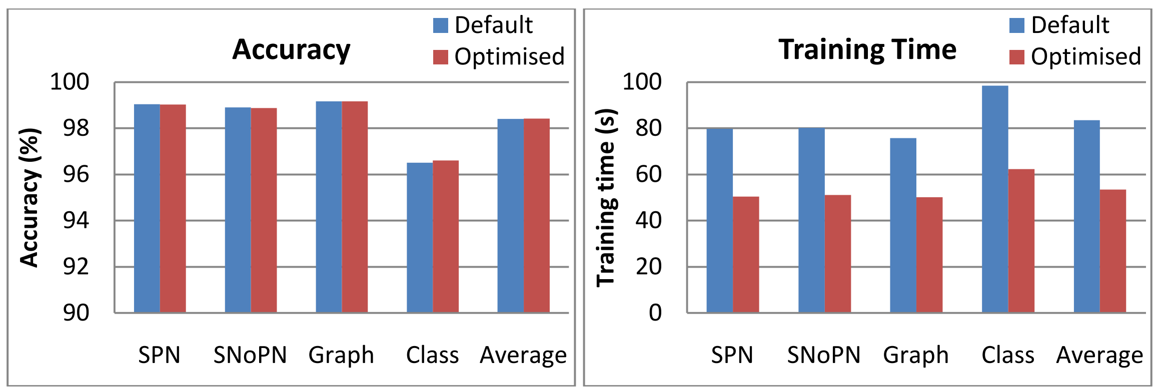


Figure 79. MultilayerPerceptron: Optimise experiment

Although there is little indication in the accuracy difference, the optimisation speeds the training time, which we believe is also a good reason to apply the optimisation when the accuracy is not decreased. The maximum testing time observed is 0.1s which is acceptable; hence all configurations can be safely applied in eager recognition.

Splitting Experiment

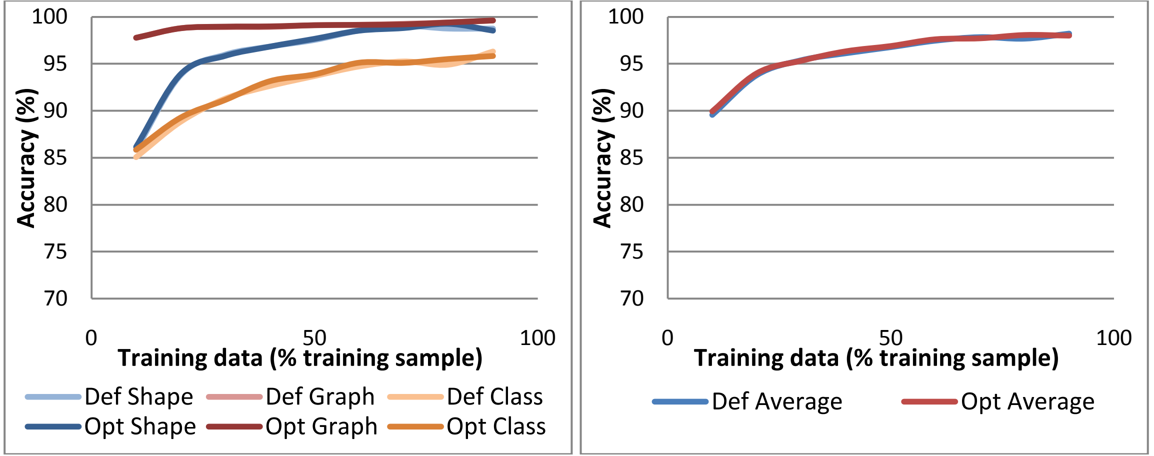


Figure 80. MultilayerPerceptron: RandomSplitting

Smooth trends occur with all three datasets. The accuracy tends to increase rapidly at the start before reaching a point, after which the growth rate reduces. No difference can be found in the optimisation, which is similar to the 10 fold cross validation situation.

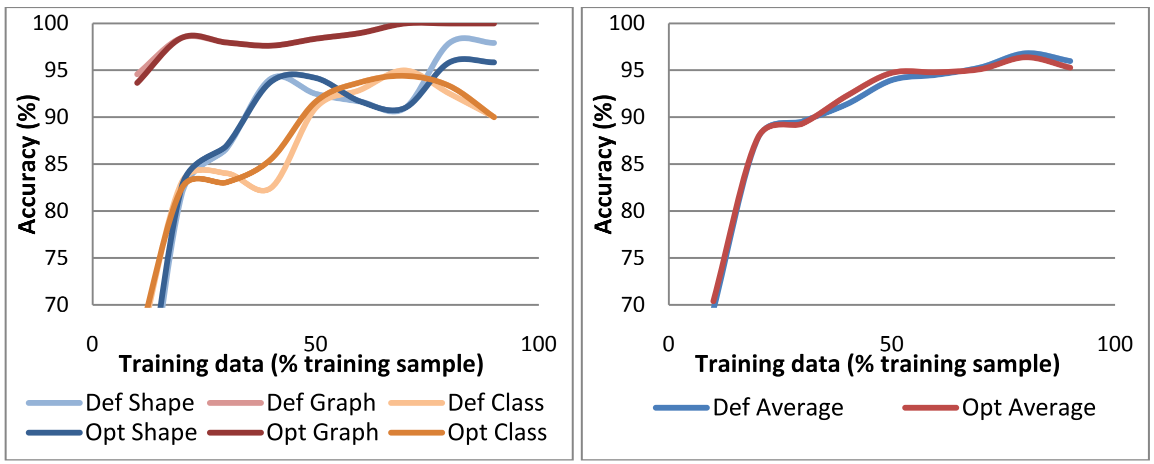


Figure 81. MultilayerPerceptron: OrderedSplitting

Again optimisation results agreed with the observation in 10 fold cross validation where there is basically no difference from the default settings. But more variations can be found between different percentages of splits in OrderedSplitting. While the performance is really bad with small number of training data, all datasets managed to have over 90% accuracy with more than 50% participant data, which may be considered the minimum number of samples required for MultilayerPerceptron.

4.3.9. Sequential Minimal Optimization (SMO)

SMO is an algorithm which trains support vector machines (SVM). A SVM uses hyperplanes for classification purposes. In a problem space, a hyperplane separates two groups of instances based on their class attributes. However, instead of simply finding any hyperplane to divide the instances, SVM finds the best hyperplane which maximises the distance from the plane to the nearest instances on both sides (Platt, 1998) . Such a process is a “very large quadratic programming (QP) optimisation problem”, which takes much time to calculate. The slow training is one of the reasons why SVM is rarely used before the introduction of SMO.

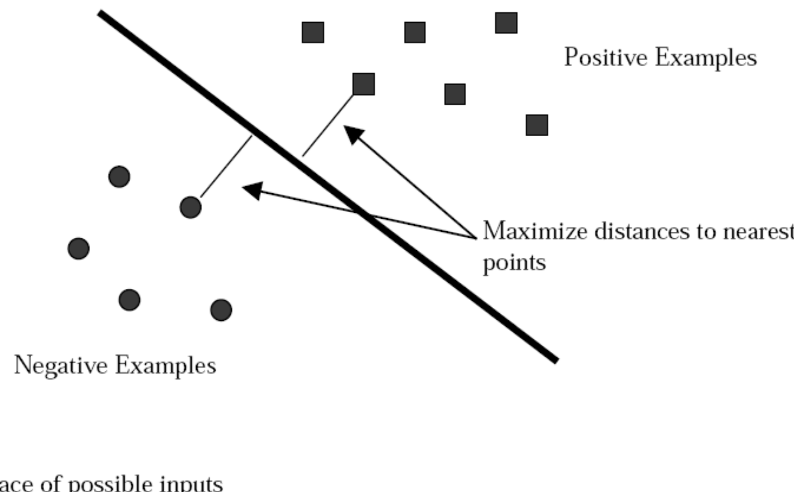


Figure 82. A linear support vector machine (Platt, 1998)

The large QP problem is broken into a series of smallest QP problems by SMO, which are solved analytically. The method made significant improvement to the scaling and computation time to the building of SVMs. Further improvements are inducted by Keerthi, Shevade, Bhattacharyya and Murthy (2001), which can speed up the SMO in many situations.

Basic Experiment

Table 13. SMO options(Hall et al., 2009)

Option Name	Option description in WEKA	Type
Build logistic models	Whether to fit logistic models to the outputs (for proper probability estimates).	Bool
C	The complexity parameter C.	Double
Checks turned off	Turns time-consuming checks off - use with caution.	Bool
Debug	If set to true, classifier may output additional info to the console.	Bool
Epsilon	The epsilon for round-off error (shouldn't be changed).	Double
Filter type	Determines how/if the data will be transformed.	Combo box
Kernel	The kernel to use.	Selection
Num folds	The number of folds for cross-validation used to generate training data for logistic models (-1 means use training data).	Int
Random seed	Random number seed for the cross-validation.	Int
Tolerance parameter	The tolerance parameter (shouldn't be changed).	double

- Build logistic models (default: false)

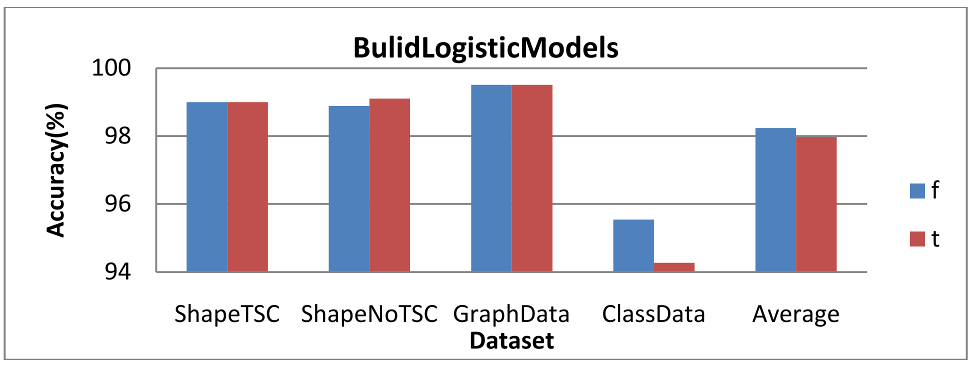


Figure 83. SMO: BulidLogisticModels

Default SMO output only the predicted class, and by enabling this setting it returns the probability of all participating classes. Such ability is not required in our study and since in average it decreases the accuracy, this setting is not selected. However it can be considered in systems such as voting or if the recognition is to be done in different levels.

- C (default: 1)

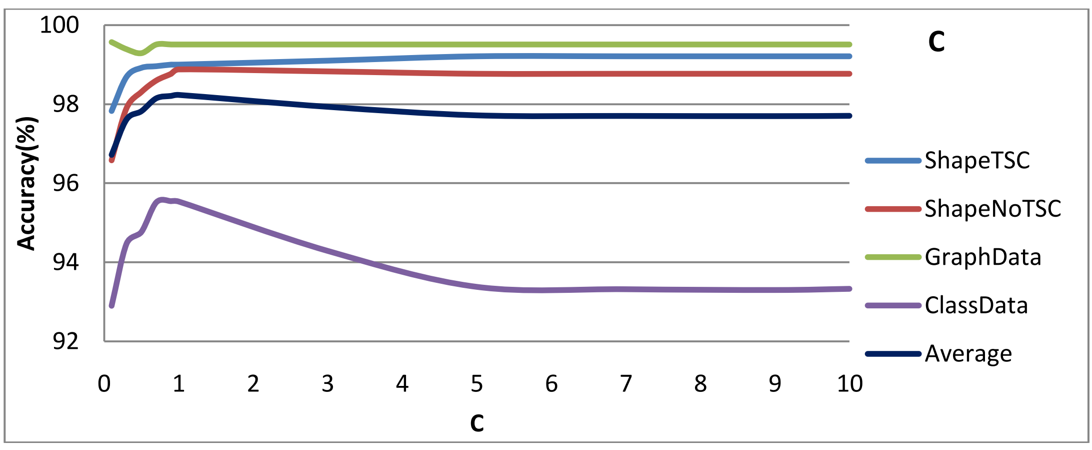


Figure 84. SMO: C

The complexity constant defines the tolerance of boundaries. Higher C indicates the hyperplane is softer which can lead to overfitting with the data, while small C means hard margin which may not give an acceptable result.

According to Figure 84, with values less than 1 the accuracy is generally increasing except for GraphData. Gradually they will reach a stable point where accuracy stops changing. This shows the behaviour depend on the number of classes. For simple datasets with fewer shape classes such as GraphData, both harder and softer margin is acceptable. We believe this is because less shape

classes make them easier to be separated. On the other hand, while ShapeData and ClassData can only perform well under certain tolerances, either lower or higher values will lead to misclassification. According to our data the complexity constant is averagely good at the default value, 1. Hence in most situations this should not be modified.

- Filter type (default: Normalize training data)

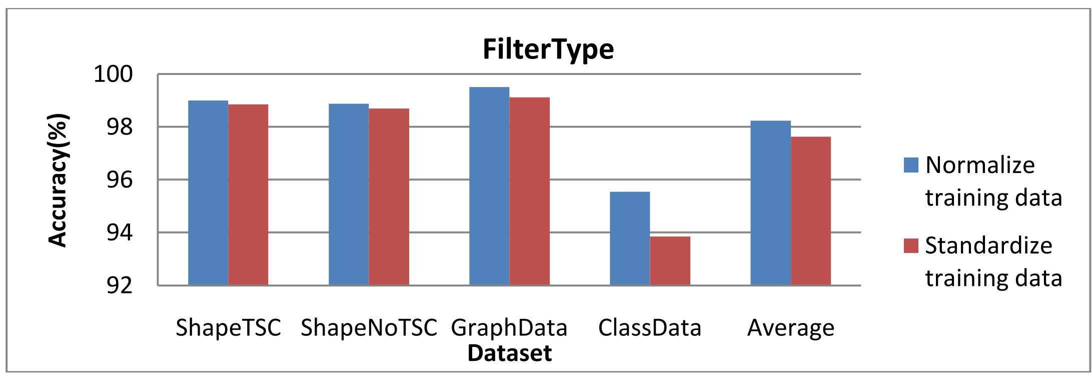


Figure 85. SMO: FilterType

This transforms the input data. Normalized training data gives higher accuracy in comparison with standardized training data. There is also another option "no normalization/standardization", however it cannot work with our feature set. Normalization has better performance, hence it is selected.

- Kernel (default: PolyKernel)

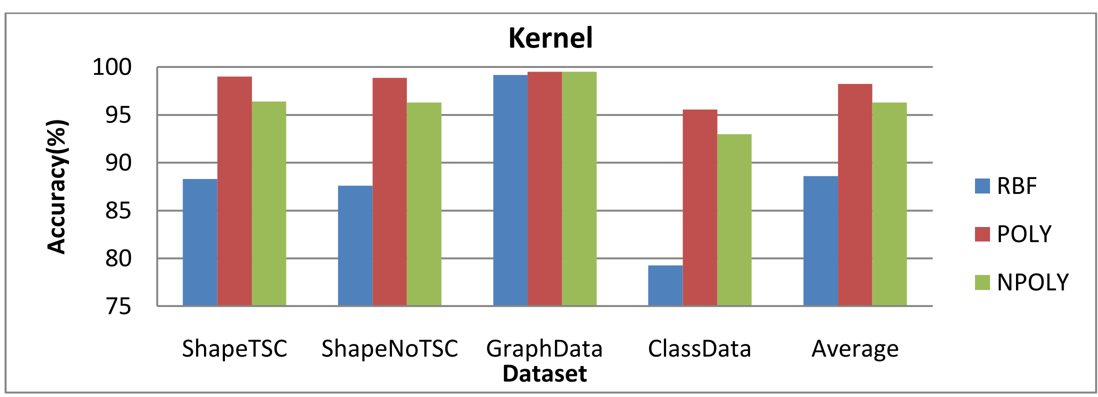


Figure 86. SMO: Kernel

Among all available kernels, poly kernel has demonstrated the best accuracy in every dataset. Although each kernel can be further tuned, we did not perform such experiment due to the time constraints.

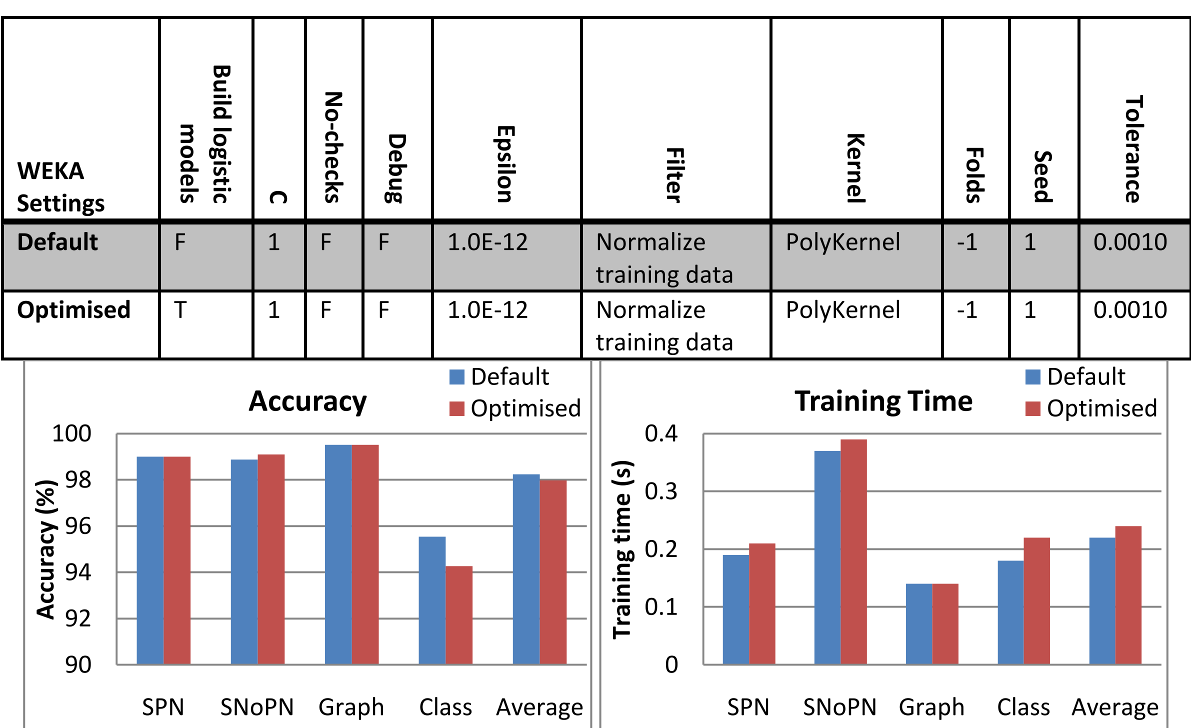
- Ineffective settings

  "Epsilon" and "Tolerance parameter" are marked as "should not be changed", as shown in Table 13.

  Checks turned off: enabling this feature causes exception during the training process, thus we did not obtain the comparison results for this setting.

  Num folds: no difference in accuracy can be observed.

Optimise Experiment



WEKA Settings	Build logistic models	C	No-checks	Debug	Epsilon	Filter	Kernel	Folds	Seed	Tolerance
Default	F	1	F	F	1.0E-12	Normalize training data	PolyKernel	-1	1	0.0010
Optimised	T	1	F	F	1.0E-12	Normalize training data	PolyKernel	-1	1	0.0010

Figure 87. SMO: Optimise Experiment

Although the default settings seem to be optimal, we decided to explore how the setting to build logistic models behaves, because it returns probability measures which can be useful in many situations. According to the experiment result, the accuracy decreases while the training time increases. The maximum testing time is 0.03 seconds which we believe is acceptable for eager recognition.

Splitting Experiment

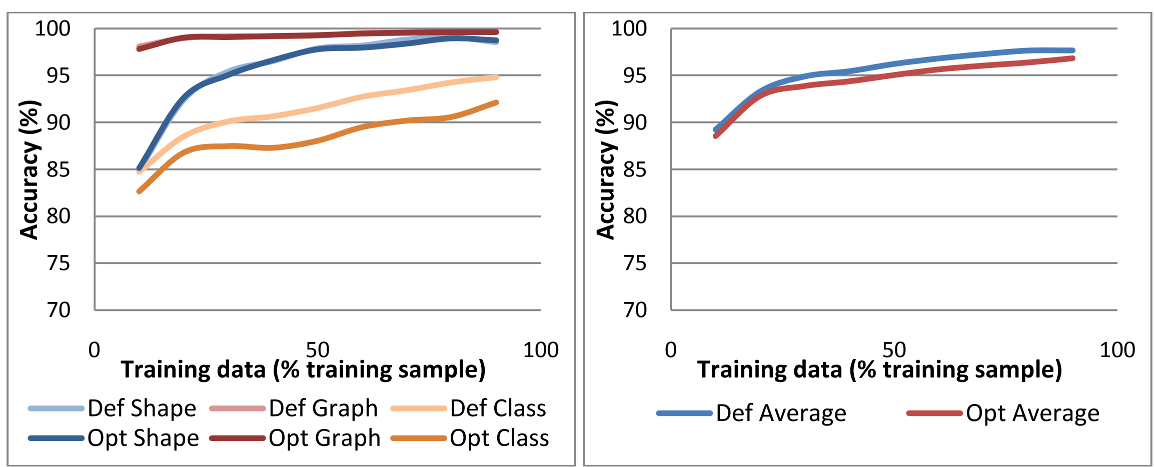


Figure 88. SMO: RandomSplitting

The optimisation in ShapeData and GraphData has similar performance, where no difference in accuracy can be found. Most accuracy differences are caused by ClassData. Since the difference is not attributable to the number of classes (because ShapeData has more classes than ClassData), we believe the result shows the difference in the complexity of dataset.

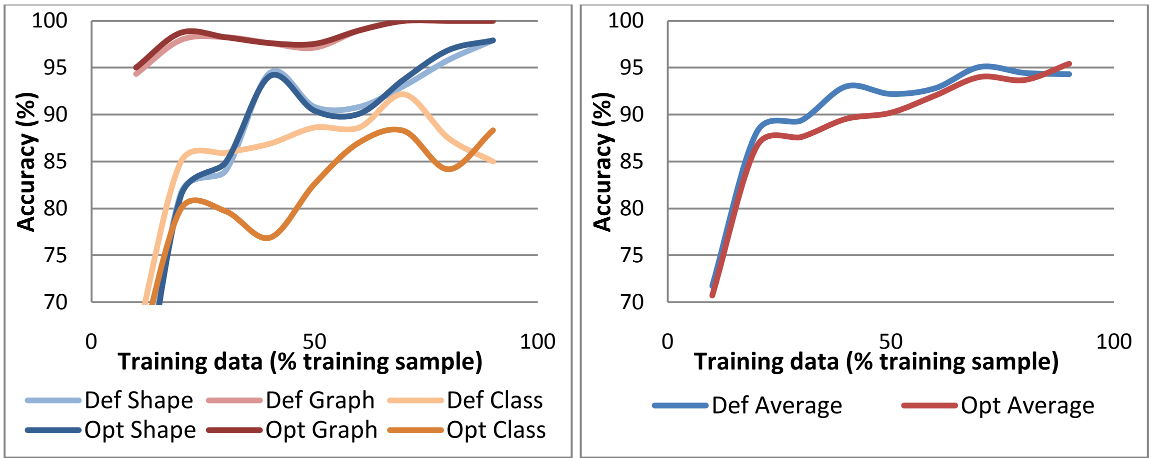


Figure 89. SMO: OrderedSplitting

Similar behaviour can be found in OrderedSplitting. A maximum of four percent difference in accuracy can be found, which further suggests that the feature should be turned off if is not required. On the other hand, this also shows SMO performs badly when the training examples are limited. It should only be used when enough examples are prepared.

4.3.10. Attribute Selection

The original plan is to set up attribute selection with 10 rounds of 10 fold cross validation on each optimised algorithm. However after several rounds of experiment, such settings turned out to be too slow as stated in section 3.2.2. We finally decided to conduct the experiment by comparing each algorithm with its default setting and the attribute selected enhanced version. Both are trained with 20% of the dataset, and tested on the rest. These data are randomised, and ten rounds are taken for each experiment, to filter off the noises.

AttributeSelectedClassifier is applied with all the selected algorithms. WrapperSubsetEval is used as the evaluator, in which the base classifier of it is adjusted to be the same as the classifier used in AttributeSelectedClassifier. BestFirst is used as the search algorithm because it has good performance. The experiment failed to run with LADTree. We believe the cause was the same as occurred in the RandomSplitting in section 4.3.5. Other algorithms ran smoothly and their performances are shown below:

Table 14. Results of Default settings vs Attribute selected classifiers

	Accuracy (%)			Training time (s)			Testing time (s)	
	Default	AttSel	AttSel-Def	Default	AttSel	AttSel/Def	Default	AttSel
MLP	93.81	94.19	0.38	19.64	6695.56	340.97	0.07	0.00
SMO	93.29	93.66	0.37	0.09	1630.36	18811.81	0.01	0.00
Bagging	93.75	92.56	-1.19	0.10	130.87	1308.73	0.00	0.00
RF	95.04	93.83	-1.21	0.02	163.81	9828.60	0.00	0.00
END	93.76	92.15	-1.61	0.23	563.98	2488.13	0.02	0.02
BN	95.92	94.04	-1.88	0.02	42.79	1834.00	0.02	0.00
LMT	95.25	93.01	-2.24	3.31	8042.61	2427.35	0.00	0.01
LB	95.79	92.79	-3.00	0.25	362.83	1470.92	0.00	0.00

According to Table 14, most algorithms have poorer accuracy when an attribute selected classifier is applied (labelled in red) except MultilayerPerceptron and SMO. Two reasons may explain this behaviour. First, most of our selected algorithms applied a certain voting algorithm, which requires variation. Selecting a subset from the original feature set may eliminate certain effective features and reduce this variation, and further reduce the accuracy. Second, most of the these algorithms generate tree structures, which splits based on the more valuable features, hence attribute selection is not only redundant but also can accidentally remove useful features. On the other hand, MultilayerPerceptron and SMO does not have the properties stated above, hence by eliminating bad attributes they show improvement in accuracy. However this improvement is not very significant.

Training times are always increased significantly compared with the default, with factors from 42 to 18,800. This is because to select attributes, experiments must be run first, and to make better decision experiments such as 10 fold cross validation are required. These evaluations increase the training time. However, for some algorithms, such as MultilayerPerceptron, SMO and BayesianNetwork, the testing time is reduced. We believe this is because all three algorithms use all features, and the reduction in the number of features increased their performance.

As a conclusion, we believe attribute selected classifier is not suitable for diagram recognition, due to the decrement in accuracy and increment in training time. However, if training time is not considered, applying attribute selected classifier can speed the testing time for some algorithms. Additionally, we also believe the reduction is mostly due to the nature of the selected algorithms. If attribute selected classifier is applied to simpler algorithms, or algorithms which do not have tree structure, the accuracy may be increased.

4.3.11. Algorithm Comparison

We have discussed all the selected algorithms and optimised each. According to the results certain algorithms performed better than the others. This section validates the effectiveness of the optimisations by considering all the experiments performed, and based on the result comparing the algorithms on their optimal configurations.

Validity of Optimisation

In the previous sections we have discussed the validity of optimisations with 10 fold cross validation and splitting experiments. Each experiment is conducted to address the limitations of another; however, they were discussed separately, which limited the overall validity of their results. On the other hand, accuracy is not the only aspect to consider; while most algorithms have acceptable testing time, in the previous experiments we have discovered the training time can increase significantly.

To show the effect of optimisation, the difference between the default setting and the optimised one are calculated. For 10 fold cross validation this is the difference between the accuracies. For OrderedSplitting, the accuracies for all percentages are averaged for both default settings and the optimised settings, and their difference is calculated; same process is done to RandomSplitting. The results are plotted in Figure 90. Positive values means the optimised setting are higher than the default setting, and vice versa.

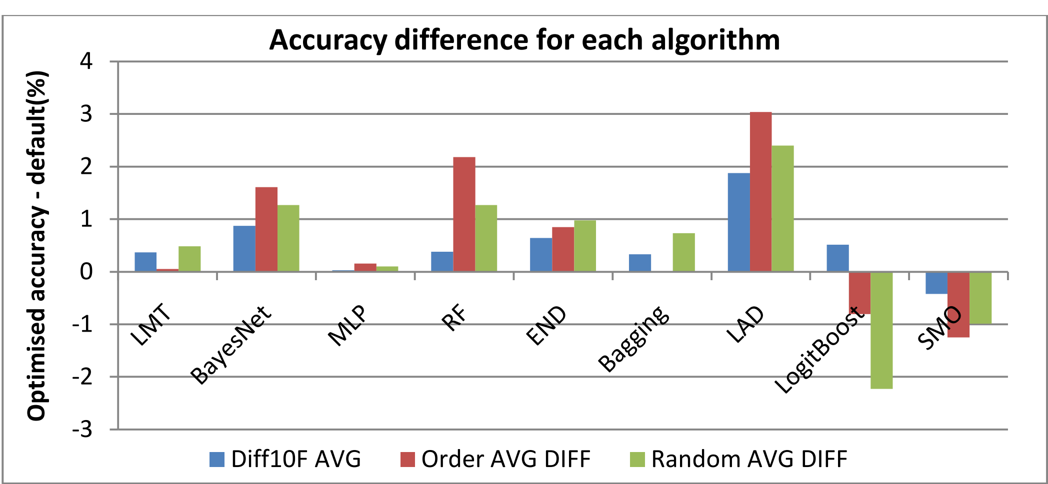


Figure 90. Accuracy difference for each algorithm

This difference shows how effectively an algorithm can be improved with optimisation. For example, LADTree certainly display a great improvement while the improvement on MultilayerPerceptron is limited. It is very interesting that although 10 fold cross validation shows improvement in LogitBoost, its accuracy is severely reduced with the optimisation under the splitting experiments.

Figure 90 does not take time into consideration. If an experiment considers training time important, it may have to consider how much value each section of time would have on optimising their algorithms. For this reason the optimisation gain is calculated with the following equations:

a) $\text{Improvement in accuracy (\%)} = \text{Optimisation accuracy} - \text{default accuracy}$

b) $\text{Optimise time factor} = \dfrac{\text{Optimisation time}}{\text{Default time}}$

c) $\text{Optimisation gain (\%)} = \dfrac{\text{Improvement in accuracy}}{\text{Optimis e time factor}}$

Firstly the accuracy improvements are taken. Averaging the three differences is not a perfect solution because it still does not guarantee to accurately simulate the overall behaviour of accuracy in optimisation; however we believe it will be less biased than if only one of the average accuracies is applied. This is represented by equation (a). Factor of increment in time is calculated with equation (b), with the training time information reported by in the optimising experiments which used 10 fold cross validation.

The gain is then calculated with equation (c), by dividing the average improvement in accuracy by the optimised time factor. The reason it is divided by the time-factor but not

time directly is because we are trying to evaluate the difference optimisation can make, but not the individual differences between the algorithms. The result, performance gain, can be explained as “the improvement each extra time unit will make to the accuracy”. The performance gain is plotted in Figure 91. Basically, the higher the performance gain, the more accuracy can be obtained with one unit of extra time. A negative value indicates the reduction in accuracy.

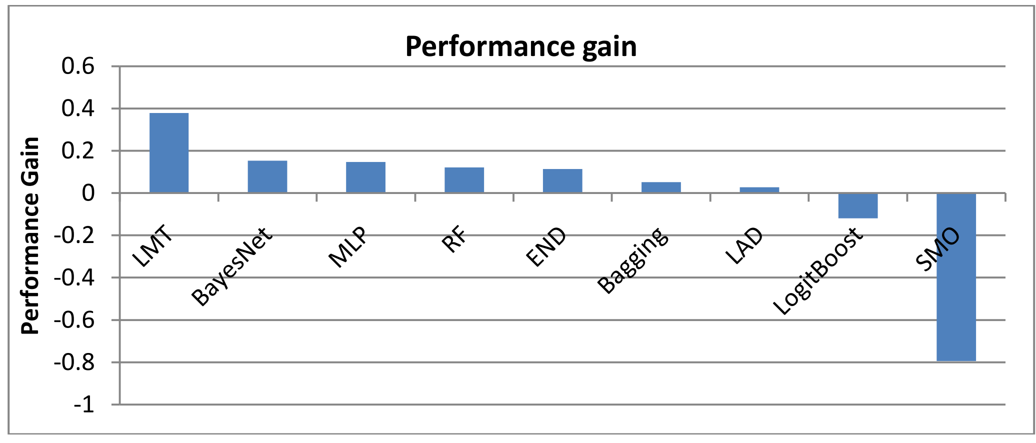


Figure 91. Performance gain

Figure 91 looks at the algorithms with a different perspective from Figure 90, as time is considered. While LMT was not particularly outstanding in the total accuracy difference, because it does not cost much to make improvement, it is suggested to always optimise it. On the other hand, although LADTree can be improved greatly with optimisation, the large time requirement reduces its applicability when training time is considered.

To further examine if the optimisation can be applied to all diagram domains, we analysed the effect of optimisation on FlowChart, which is the dataset not used in the optimisation study. 10 fold cross validation was taken to produce a more general expression of the result.

Table 15. The comparison between FlowChart and the averaged data from other datasets

		Bagging	RF	LB	END	LAD	LMT	BayesNet	MLP	SMO
FlowChart	Def	96.77	98.04	98.66	98.42	96.59	98.89	98.41	97.61	97.79
	Opt	97.45	98.86	99.07	98.52	98.86	98.83	99.09	97.58	94.29
	Diff	0.68	0.82	0.41	0.1	2.27	-0.06	0.68	-0.03	-3.5
Average of other datasets	Def	96.1	97.9	98.4	97.1	96.5	98.0	97.7	98.2	98.0
	Opt	96.5	98.3	98.9	97.7	98.4	98.4	98.6	98.3	97.6
	Diff	0.3	0.4	0.5	0.6	1.9	0.4	0.9	0.0	-0.4
Agreement		+	+	+	+	+	-	+	?	+

Table 15 shows the comparison result. Both the default and the optimised settings of each algorithm are applied to each dataset, and the differences between them are calculated. The results for ShapeData, GraphData and ClassData are averaged. We expect if the average of these three datasets demonstrates that optimisation improves the accuracy, the same behaviour should occur on FlowChart, and vice versa. These situations indicate that the results of FlowChart agree with the others.

Agreements can be found from seven out of nine algorithms, except LMT and MultilayerPerceptron. The case for MultilayerPerceptron is unclear because it has a difference scale of zero for the other datasets. The result for LMT is unexpected because according to Figure 63 the accuracy increased for all datasets. Overall for the majority of algorithms the optimisation results are applicable.

Algorithm Ranking

Although training time may be important under certain situations, the focus of this experiment is still on finding the maximum accuracy. In this section we aim to rank the algorithms based only on the accuracy. All algorithms are trained with their best configurations. 10 fold cross validation and splitting experiments are performed to ensure the correctness of the result. For the splitting experiments, both standard 10% to 90% RandomSplitting(RS) and OrderedSplitting(OS) are conducted.

On the other hand, algorithms can be overfitted with small training samples. Because in real world usage users are likely to find sufficient training examples, we explicitly ran another set of splitting experiments with 50% to 90% (R50 and O50), to ensure enough training examples are presented.

For each of the five experiments, the most accurate algorithm will have a ranking of one while the second has a ranking of two, and so on. Hence each algorithm will possess five different ranking values. To get the final ranking, these five values are considered together. Because the O50 and R50 are essentially duplicating part of the information within OrderedSplitting and RandomSplitting, their rankings are only considered half as important as the other ones. We believe this will give a more reasonable result than using the average of five. The average ranking is calculated with the equation below:

$$Average\ Ranking = \frac{OS + RS + 10fold + 0.5(O50 + R50)}{4}$$

Table 16. Algorithm rankings

	Accuracy					Ranking					
	OS	RS	O50	R50	10fold	OS	RS	O50	R50	10fold	Avg
BN_{Opt}	96.31	97.86	97.52	98.38	98.59	1	1	1	1	2	1.25
LB	93.06	96.90	96.90	98.17	98.89	4	4	2	2	1	2.75
RF_{Opt}	95.42	97.71	96.09	98.16	98.27	2	2	4	3	5	3.125
LAD_{Opt}	94.27	97.64	96.44	98.09	98.42	3	3	3	4	3	3.125
LMT_{Opt}	91.49	96.13	95.24	97.31	98.37	5	6	6	7	4	5.375
MLP_{Opt}	90.68	95.98	95.31	97.64	98.26	7	7	5	5	6	6.25
END_{Opt}	90.99	96.19	94.61	97.39	97.74	6	5	7	6	8	6.375
SMO	90.13	95.38	93.77	97.11	98.02	8	8	8	8	7	7.75
$Bagging_{Opt}$	89.76	95.30	92.46	96.26	96.45	9	9	9	9	9	9

The optimised BayesianNetwork demonstrates the best overall performance according to Table 16. Bagging has the worst performance in all tests, which made it the worst algorithm; however one thing to keep in mind is that altering the base classifier of Bagging can improve its performance, as indicated in Figure 27. The results are very consistent apart from LogitBoost which, although generally performs very well, produces a comparatively poorer performance with the full splitting experiments. This is because it tends to overfit the data when training samples are not enough. In comparison although RandomForest and LADTree have lower rankings, they have better performance with limited training samples.

4.4. Combined WEKA Algorithms

Section 4.3 analysed and optimised the selected algorithms. Many of these algorithms used a democratic approach, in which they firstly trained many classifiers with the same algorithm but made small variations to the data to make these classifiers different. When the testing data is given, each of the trained classifiers will give a classification which will be united to give a final classification. A similar approach is applied with the two mechanisms, Voting and Stacking. They are applied to further increase the accuracy of the optimised recognisers. However, instead of combining instances of the same algorithm, they are used to combine classifiers trained with different algorithms.

Both Voting and Stacking are time consuming. Limited by the time constraint and the computation power, their potential is not fully explored. Although both classifiers have adjustable settings, they are not tuned, because we are more interested in the effects of applying different numbers and combinations of algorithms. Experiments with Stacking

were terminated half way through the study, because its performance is consistently worse than Voting.

4.4.1. Voting

Instead of returning only the final classification, most classifiers in WEKA can return the probability for each stroke. Voting combines classifiers by averaging their probability estimates of each class and making decisions on the highest possible classification. Witten and Frank (2005) suggest it will deliver good performance if all selected algorithms are accurate; however, if the majority of the selected algorithms are not doing well, it can result in bad performance. We believe Voting can work well in our situation, as these classifiers are all specifically selected, tuned and return high accuracy.

The first experiment is to put all algorithms to vote. Then different combinations of algorithms are tested. Although we would like to explore all possible combinations, there are too many and cannot be achieved in the scope of this project.

Table 17. The algorithms used

	Default	Optimised
All	All algorithms	Not conducted because requiring too much time
5	BN, LB, RF, LAD, LMT	BN_{Opt}, LB, RF_{Opt}, LAD_{Opt}, LMT_{Opt}
3	BN, LB, RF	BN_{Opt}, LB, RF_{Opt}

The combinations are shown in Table 17. Different numbers of participating algorithms are analysed to see if removing algorithms will improve the performance. The experiment has both default and optimised versions, which can reveal if optimising the participating algorithms can improve the combined algorithms. The detailed settings can be found in Section 4.3 for each algorithm. Although Voting is a WEKA algorithm and it allows the change of its rules to combine the algorithms, this setting is not changed; the default rule "Average of Probabilities" was used.

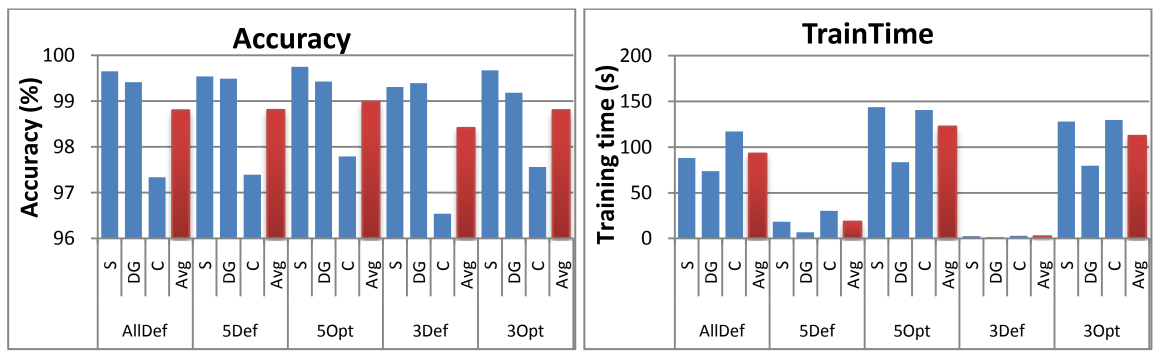


Figure 92. Voting: Combination and optimisation

According to Figure 92, optimising the participating algorithms improves the accuracy, with a trade-off of the increased training time. Although the number of participating algorithms does not directly affect the accuracy, different combinations do perform differently. Comparing only situations where the inner algorithms are with their default settings, the performance of the five best algorithms is better than the combination of all, which is better than the three best algorithms. To further analyse the effects of applying different combinations, an experiment is conducted.

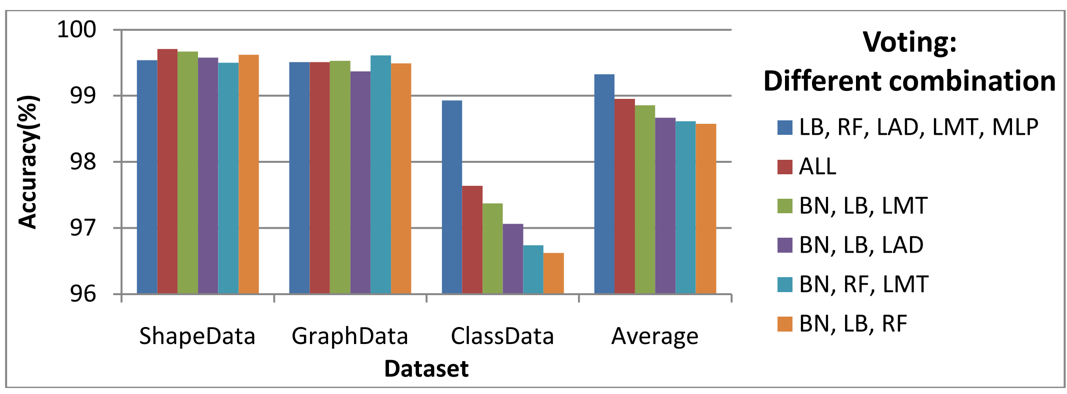


Figure 93. Voting: Different combination

Experiments in Figure 93 are conducted with 10 fold cross validation to filter the noise. While on average using three algorithms is still the worst and using five algorithms still yields the best results, different combinations of three algorithms return different results. The combination of BayesianNetwork, LogitBoost and LMT achieved the maximum average accuracy. Because the algorithms used are all in their default settings, the default accuracy with 10 fold cross validation is provided.

Table 18. The average accuracy with 10 fold cross validation

	MLP	LB	SMO	LMT	RF	BN	END	LAD	Bagging
Accuracy	98.4	98.4	98.2	98.2	97.9	97.5	97.3	96.4	96.1

According to Table 18, LADTree has much lower accuracy than RandomForest, however the Voting using it has better performance than the one using RandomForest, while the other two algorithms are the same. This indicates that the combination of best algorithms may not be the best Voting algorithm. On the other hand, according to Figure 92 increasing the number of algorithms to five increases the accuracy significantly, especially for more complex datasets such as ClassData. This shows that different algorithms can use their strengths to complement each other. However, among these strong algorithms there are still weaker ones, which if added may not achieve the maximum performance. This means the application of all algorithms produces poorer

performance than the combination of five. Finding the strength of each algorithm and combining them appropriately may be a way to optimise the performance.

4.4.2. Stacking

Stacking can be used to combine classifiers built with different algorithms (Ting & Witten, 1997). Instead of obtaining un-weighted votes as done in Voting, it uses a meta-learner which attempts to learn the reliability of different classifiers. The target algorithms firstly form classifiers (level-0 models) to classify the original dataset. Their classification results are then combined to form a new dataset, which is used by a higher level algorithm to form a level-1 model. To avoid the level-1 model being too optimistic in using the data which trained level-0 models, part of the original data is reserved specially for level-1 model training. After the training of the level-1 model is completed, these data are then combined with the original data, in order to be used by the level-0 algorithms so they can be better retrained to obtain better performance.

Similarly to Voting, we wanted to find out the relationship between the optimisation and the different combination of algorithms. The meta-learner used is J48 algorithm, while the number of folds used for cross-validation within Stacking is 10.

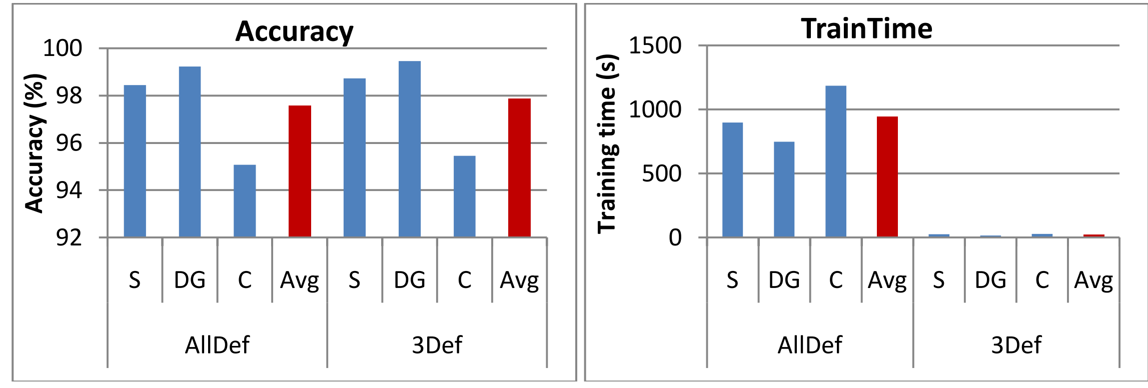


Figure 94. Stacking: Combination

Choosing the three best algorithms can improve the accuracy, in comparison to applying all algorithms. However, both accuracies are less than any result achieved by the Voting algorithm, with the same algorithms used. Our hypothesis on why this happens is discussed in section 7.1.2. In addition to the lower accuracy, for the same combination of algorithms Stacking requires much more time to train, making it less usable. Hence only a limited number of experiments were conducted on Stacking, because Voting appears to be more suitable to diagram recognition and it was decided to focus more on the Voting

mechanism. However, this experiment applied only the J48 tree as the meta-learner; more suitable algorithms may further improve the accuracy.

To further evaluate this result, splitting experiments were also performed for both Voting and Stacking: Voting still demonstrated superior performance. The comparison will be presented in 4.4.3.

4.4.3. Comparison with the Singular Algorithms

Voting and Stacking are only useful if they improve the accuracy. If a combined algorithm which requires more time to train does not perform better than using singular ones, it does not have much practical value. The usefulness of the combination algorithm can be observed in Figure 95. This study applied experiments we used to rank the algorithms in section 4.3.11. V3 is Voting with the three best algorithms, S3 is Stacking with the three best algorithms. For the singular algorithms, "Max" takes the highest accuracy for the category, "Min" takes the minimum and "Avg" averages them. None of the Stacking experiments are optimised because too much time is required.

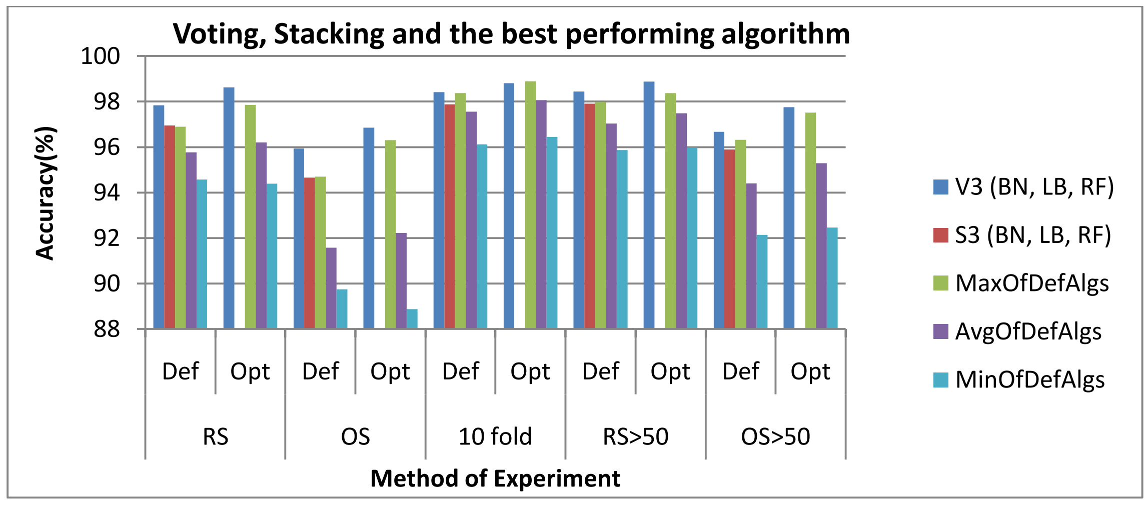


Figure 95. Voting, Stacking and the best performing algorithm

To ensure all situations are tested, different evaluation methods are applied. RandomSplitting (RS) and OrderedSplitting (OS) both average the performance from 10% to 90% with 10% interval, while we also considered the accuracy of averaging only the accuracies better than 50%. The traditional 10 fold cross validation (10 fold) is also applied.

The performance of Voting is clearly the best among all the algorithms. Stacking rarely outperforms the best algorithm, but it is always better than the average. Both Voting and

Stacking showed acceptable testing times for eager recognition. Voting is preferred between the two, because it delivers better performance, and takes much less time to train. On the other hand, according to previous experiments, we know the combination with the three best algorithms does not return optimal accuracy; furthermore, these combination algorithms are not optimised – the optimisation is only on the algorithms they used, but their own settings are not adjusted. Hence there are still improvements which can be done to Voting and Stacking.

4.5. Rubine

The mechanism used by Rubine (Rubine, 1991) is introduced in section 2. However, there is no evidence in the literature showing the features used by Rubine are the most appropriate ones. Additionally research demonstrated that adding features can improve its recognition accuracy (Plimmer & Freeman, 2007). As the algorithm demonstrated good performance in comparison to some more complex classifiers (Schmieder et al., 2009), we are interested to see if data mining can find a better selection of features to further improve its accuracy.

The version of Rubine used in this study was implemented by Plimmer and Freeman (2007).

4.5.1. Attribute Selection Setup

Attribute selected classifier was used to select better attributes for WEKA algorithms; however this classifier can only work with WEKA implemented algorithms, which does not apply to the Rubine algorithm used in this study. Hence, we decided to apply attribute selection separately.

CfsSubsetEval is used for attribute evaluation. It considers both the effectiveness of the attribute toward the classification, and also the inter-correlation between the features. Best-first search is used to search through the best attribute subset; although it takes more time than the others implemented in WEKA, it is more reliable in returning a better result.

The major limitation of this setup is that the returned attributes are not weighted. However this is only a preliminary study and the goal is to see if data mining is effective in improving the Rubine algorithm.

4.5.2. Results

A list of attributes is returned by attribute selection. The algorithms used does not consider the importance of each feature, hence these features are in the order that they appeared in the dataset. To analyse the effect of using different numbers of attributes, we run Rubine multiple times each with different number of selected features. Instead of performing 10 fold cross validation, three experiments are run and averaged. The first experiment trains and tests the classifier with the complete dataset (O), the second one trains with the upper half participants and test on the rest (TB), while the third one trains with the lower half participants and test on the rest (BT). The result is ploted in Figure 96.

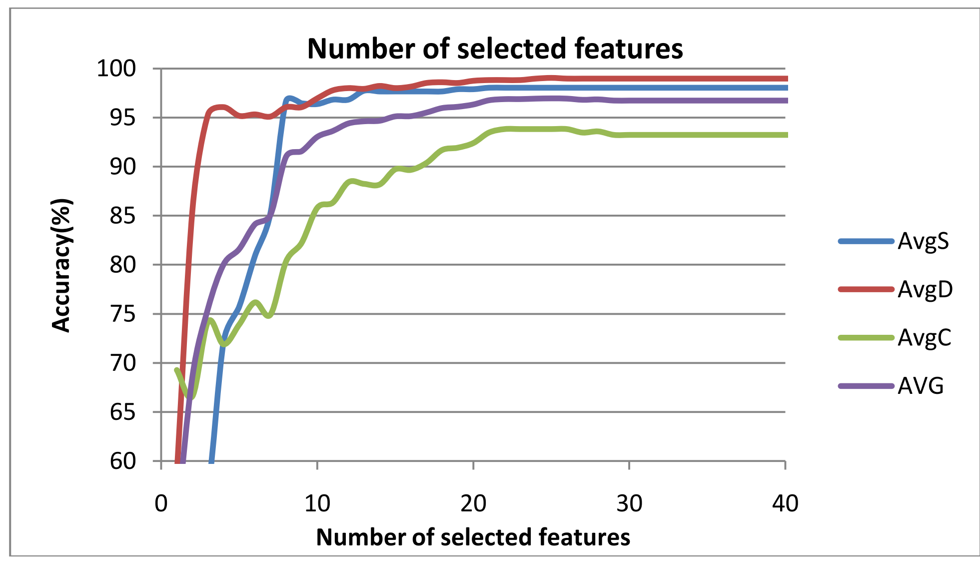


Figure 96. Accuracy vs number of attributes

The attribute selection method stops selecting more attributes once it considers all the important features are selected, which is why the trends for all datasets become static after certain values. These numbers are shown in Table 19.

Table 19. The maximum number of selected attributes

	ShapeData			GraphData			ClassData		
Training participant	All	1-10	11-20	All	1-10	11-20	All	1-10	11-20
Stopping value	19	21	15	26	20	20	22	29	20

Overall, the final accuracy follows the complexity of different datasets. However the growth trends of each dataset are different. This is because the returned list of features is not ranked, but in the order DataManager generates the features. We analysed the data and found that the sudden increments are all due to the presence of important features, which are data dependent. For example, the feature “29. Distance from First to Last Point”

caused the accuracy of GraphData to increase from sixtyish to ninetyish, but it doesn't have such a dramatic effect on ClassData, and it is not even selected by ShapeData.

Once the important features are found, a huge jump in accuracy occurs, which is what happened with both ShapeData and GraphData. On the other hand, ClassData started with a much higher accuracy, which is because its initial selected feature is better than the other datasets; however because the dataset is more complex, each feature makes a limited contribution to the accuracy, hence the growth rate is more stable.

To evaluate if the attribute selection can improve the Rubine algorithm, they need to be compared. In this experiment, only 11 of the original Rubine features are used, hence the number of attributes to be selected is limited with the same number, in order to make a fair comparison. The result is shown in Figure 97.

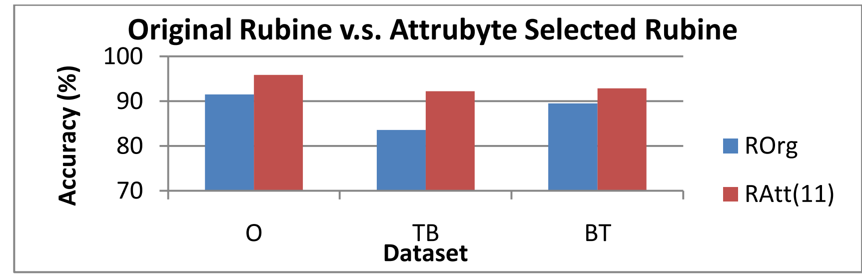


Figure 97. Orignial Rubine v.s. Attribute Selected Rubine(11)

The accuracy is significantly improved under every situation, and the attribute selected Rubine appears to be more stable. Furthermore according to Figure 96 the accuracy can be further improved. This shows strong evidence that Rubine algorithm can be improved with a revised feature set, which can be obtained with data mining in the training data.

On the other hand, can attribute selection improve an existing feature set? Most WEKA algorithms cannot be improved by it, as demonstrated in section 4.3.10. We assumed that it was because the nature of the algorithms used, and believed the technique can be applied to improve Rubine. However, the experiment above cannot answer this question because it only compares one subset to another. We therefore conducted an experiment by using all features.

Table 20. Rubine testing whole dataset

	ShapeData	GraphData	ClassData
All features	18.14%	28.73%	14.08%
All features exclude NP	13.61%	28.73%	27.27%

As shown in Table 20, very low accuracy was obtained. However, as mentioned Rubine is very sensitive to the input information and such low accuracy is probably caused by these unsupported features. We have also tried to give it a selection of features which we believe are all acceptable by Rubine and compared the difference in using all given features and selecting attribute among them. However, the result varies, and we are not confident that the variation in accuracy is due to the complexity of problems rather than to some unsupported features. In the end, we decided to take attributes from the original Rubine and perform attribute selection within these features.

Table 21. Attribute selection from original Rubine

	ShapeData	GraphData	FlowChart	ClassData
Original Rubine(11)	86.27%	96.31%	85.53%	77.38%
Attribute selection in Original Rubine	90.23%	95.29%	86.88%	80.55%
Average number of Attribute	5	6.5	5.5	4

Table 21 shows the result of this comparison. Each dataset is split into half, trained with one and tested with another, and the training set and testing set are reversed for another experiment. The resulting accuracy and the number of attributes used are averaged. The data shows that attribute selection outperforms the original feature set most of the time in our experiments.

4.6. Summary

In this chapter, nine algorithms are selected from WEKA. They are optimised toward three datasets, and tested with 10 fold cross validation and splitting experiments to show the validity of these optimisations. These algorithms are ranked as shown in Table 16. Attribute selection and algorithm cooperation are performed, in which the Voting mechanism showed the most promising results. Furthermore, attribute selection was applied to Rubine algorithm, which returned very promising results. These settings and optimisations will be used in the next chapter to create a recogniser generator which is easy to use and can create quality recognisers.

Chapter 5
Implementation

In the previous chapter, nine algorithms were selected from WEKA (Hall et al., 2009) and optimised with various means. Several were combined with the Voting mechanism to further increase the overall accuracy. All these experiments were done through the interface WEKA provided. However, the WEKA interface is complex. Because data are collected and labelled with DataManager (Blagojevic, 2009), it more intuitive if the training process can also be done within it. Furthermore, because the WEKA generated classifier cannot be directly used by other programs, an interface needs to be built between WEKA and C# programs. In addition, the other algorithm considered in this project, Rubine, needs to cooperate with WEKA.

These requirements lead to the implementation of our recogniser generator. The tool is named Rata.SSR (**R**ecognition **a**lgorithm **t**ools for ink **a**pplications: **S**ingle **S**troke **R**ecogniser), which is part of the Rata project and which will work in concert with the Rata.Divider that separates text and drawing (Blagojevic, Plimmer, Grundy, & Wang, 2010) at the University of Auckland. This chapter starts with the overview of the implementation, which is followed by the designed architecture. Next training and classifying processes are described separately. The chapter concludes with the opportunities for users to customise algorithms for Rata.SSR.

5.1. Overview

The goal is to build a recogniser generator which can generate recognisers that accurately recognise ink strokes. As mentioned in section 1.2 one of the objectives of this project is to reduce the cost in the development and implementation of recognisers; this means the generated recogniser should be easy to use, even by programmers without knowledge in recognition or data mining. It should also be extendable so advanced users are able to customise recognisers for their diagram domain. This section gives an overview of the design decisions made to meet these requirements.

5.1.1. Ease of Use

Ease of use is important for any software component. In order to achieve this, several steps are taken.

- Select a technology (programming language) and unite the different data structures and API styles toward it
- Hide the unnecessary details of the implementation
- Allow the code to be used through the most common data structure of the selected technology
- Minimise the size of the implemented component so it can be easily installed and carried.

Technology Choice

The first decision to make is what programming language this tool will be implemented in. The project is based on two C# based software, DataManager and InkKit Rubine (Plimmer & Freeman, 2007), and a Java based software, WEKA 3.7.1. Hence we have to choose between C# and Java. It is possible to build a hybrid system, for example using DataManager to generate feature sets and load them into WEKA for classifier generation, but such implementation involves the changing of data formats and sending the data between two different programming languages, which is both time consuming and complex.

C# is selected for three reasons. First, two out of the three base projects are implemented in C#. Second, the Microsoft Ink library, which is used by DataManager, is only .Net compatible; because there is no standard ink library for Java, we believe the application of the Ink library is beneficial. Third, the WEKA library can be translated into .NET compatible format through the use of IKVM.NET (Frijters, 2009). IKVM.NET runs a Java Virtual Machine within C#, and to optimise the performance it sets up much Java code for direct use by .NET libraries. The version of IKVM.NET used is 0.40.0.1.

The project is fully programmed in C#. The translated library allows C# code to access the WEKA library. Because we are developing .NET application under Windows, Visual Studio becomes a natural choice. The version we used is Visual Studio 2008. Windows Vista is used, which offers tablet compatibility.

Model for Novice

Users of the recogniser generator may not be data mining professionals. Hence, instead of making them explore the WEKA interface or tune the settings, it is more suitable to provide them with a set of fully tuned algorithms. The best configurations from the previous chapter are taken to make this list of algorithms. Thus, users can ignore the configuration details. In addition, because the number of algorithms is reduced compared with that offered by WEKA, it is more feasible for users to explore all the algorithms offered by Rata.SSR and select the best among them.

Furthermore, the interfaces for DataManager, WEKA and InkKit are different. Their programming interface should be united so users can use different algorithms through the same code. To match the most common use case of C#, the selected input is *Strokes* from the Ink library, and the output is *String* which returns the classification results.

Encapsulation

Not only should the programming interface be united, but also the user interface for generating the recognisers. Because the training process utilises the features generated from DataManager, it is natural to have the program built within it. The relationship for the training process between DataManager and Rata.SSR is shown in Figure 98a.

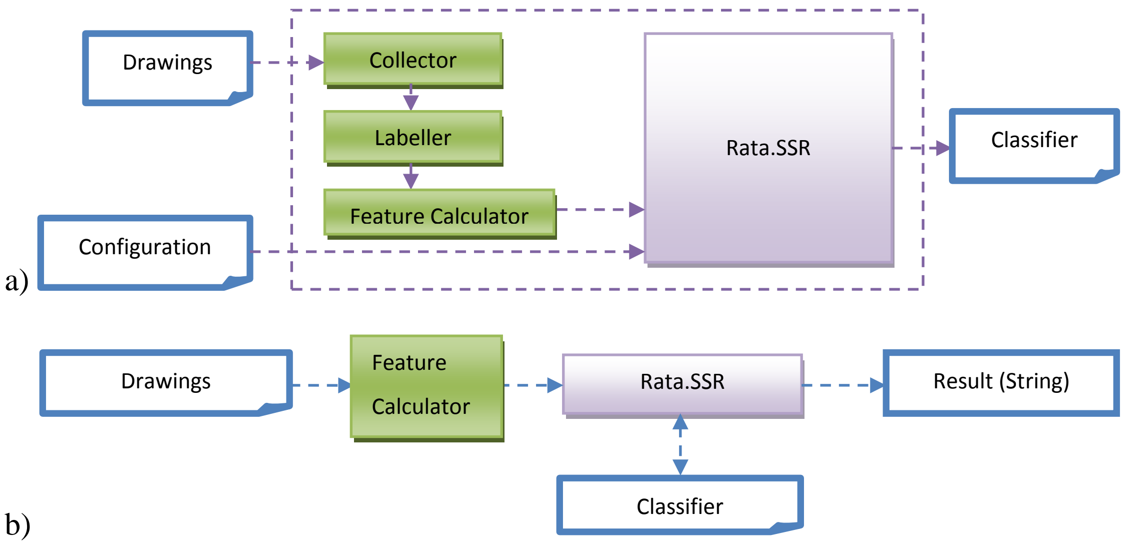


Figure 98. Rata.SSR use case. (a)training (b)classifying

The green rectangles in Figure 98 are the components provided by DataManager, which are namely DataCollector, Labeller and FeatureCalculator. However DataManager provides more components than these three, which are unnecessary to Rata.SSR and large in size. As stated earlier in this chapter, Rata.SSR not only has to be used to train

classifiers, but also needs to be the bridge between the built classifiers and the application code. This means that under certain situations such as client site, only the classifying functionality is required. The classifying process uses only the FeatureCalculator functionality, as shown in Figure 98b. To reduce the size of the recogniser, the first solution we had is to make Rata.SSR dynamically generate DLLs which are non-trainable recognisers that can be directly used. While such implementation is simple to use, if each recogniser is a different DLL, it becomes difficult to swap or add recognisers at runtime. Additionally, if two recognisers are to be used together, they will contain duplicated code such as that used for feature calculation.

Hence we decide to make Rata.SSR itself the bridge. In this case these recognisers act like components and Rata.SSR is the central recognition engine which can generate these components as well as use them, thus individual recognisers can be freed from having duplicated functionalities. Furthermore, because the generated recognisers are usually small in size, updating or changing the recognisers is much simpler.

Redundant components should be excluded, because they are not required in client site. The final decision is to only take the FeatureCalculator from DataManager. Although training requires the use of DataCollector and Labeller, because after the feature calculation of DataManager the information can be saved in CSV files, Rata.SSR is implemented to read these files and load the calculated features. This implementation eliminates the need of the unnecessary components, which reduces the size and simplifies the programming interface. Rata.SSR thus becomes a standalone project which does not need DataManager to operate. On the other hand, although it can be used separately, the user interface for training is still implemented within DataManager to keep consistency.

The tool is made into a DLL library which is easier to transfer and use. Four other DLLs are required to use this DLL, which are: IKVM.OpenJDK.ClassLibrary.dll, IKVM.Runtime.dll, WEKA.dll and Microsoft.Ink.dll. The total size of these four DLLs with our current version is 39.7MB, which is not very large. The Rata.SSR itself is only 303 KB, which is very portable in terms of size.

5.1.2. Extensibility

While many decisions are made to ensure Rata.SSR can be used by novice users, it is also important to consider the advanced users with data mining knowledge. Although good

algorithms are selected and optimised in Chapter 4, these optimisations are hardcoded to capture the general aspects of the diagram domain; as indicated by the experiment results it is possible to make further optimisation for individual diagrams. On the other hand, algorithms not analysed in this study may be suitable for certain problems. These extension opportunities should be provided.

The open source nature of WEKA allows it to be extended by data mining experts. The implemented algorithms can then be used in Rata.SSR if the WEKA.dll is updated. Additionally, experts can use the WEKA explorer interface to customise their recognisers. Because we unified the file structure of the exported classifiers of Rata.SSR and WEKA, the customised WEKA classifier can be loaded by Rata.SSR and used to recognise C# strokes. This configuration enables users to make further classification for individual classifiers, as well as allowing all algorithms provided by WEKA to be used.

Although the settings are fixed for the preset algorithms, extensibility can also be included in class construction. This is explained later in this chapter for individual algorithms.

5.2. Architecture

One important aspect of Rata.SSR is to combine WEKA and Rubine together. Although WEKA can be used within C# with the help from IKVM.Net, it still has different data structures, classes, and handles from Rubine. While it is possible to specify different programming interfaces for each algorithm and to allow users to decide how to use them, such an approach is not very user friendly. Instead we want to unify the programming interface, to only let users see and use the most appropriate C# classes which are necessary for diagram recognition.

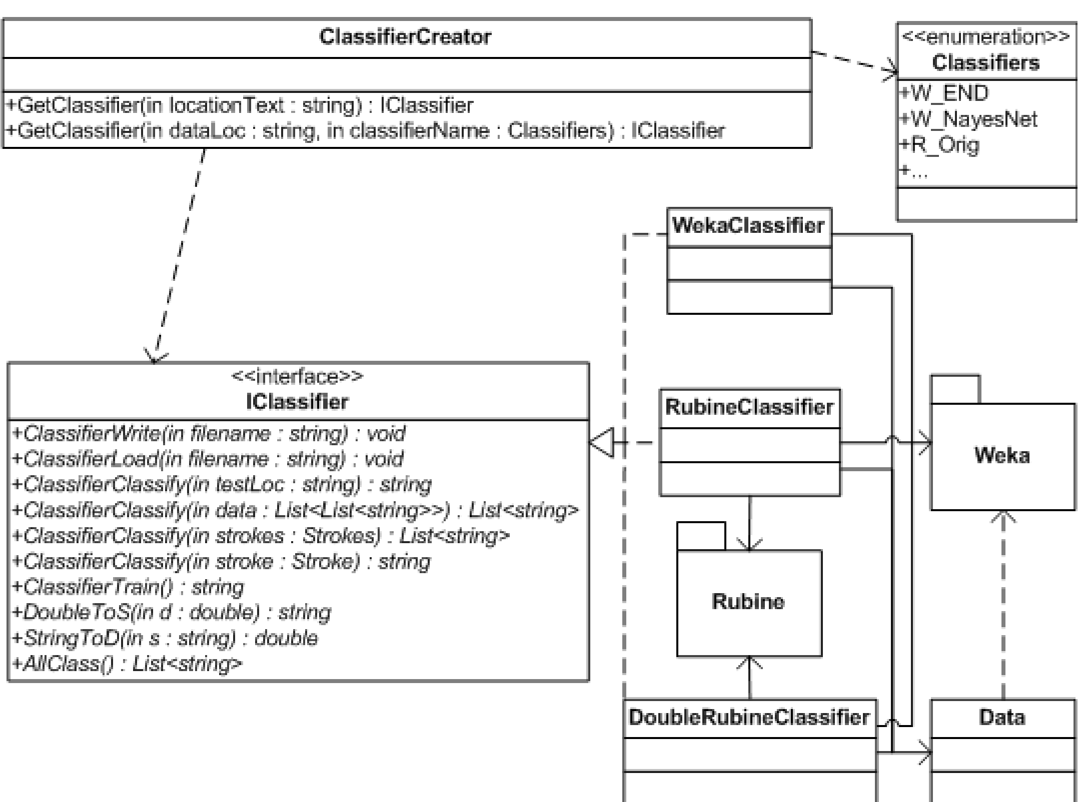


Figure 99. System architecture

The architecture is thus designed as shown in Figure 99. In total we have implemented three classifiers: *WekaClassifier*, *RubineClassifier* and *DoubleRubineClassifier*. Although they utilise different mechanisms, their I/O is unified by the *IClassifier* interface. This is an application of the simple factory design pattern.

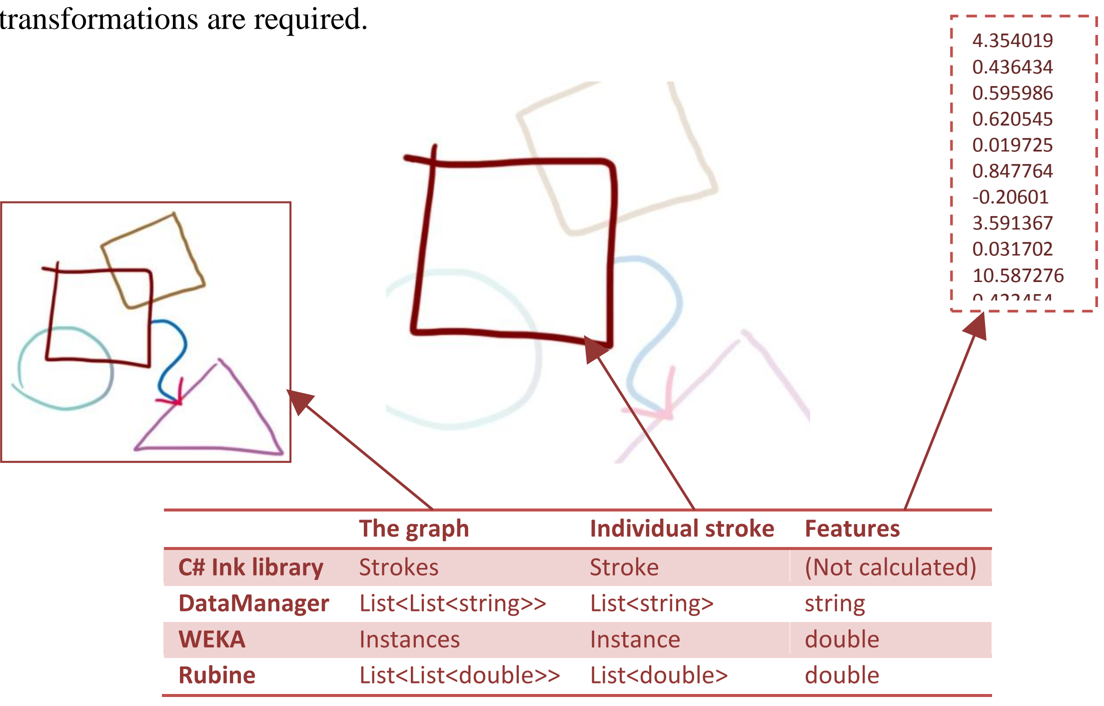
5.2.1. Simple Factory

The simple factory design pattern provides flexibility; for example, by specifying an *IClassifier* object, both *WekaClassifier* and *RubineClassifeir* can be used. Furthermore, the interface encapsulates the implementation details; all operations apart from the construction of classifiers can be done through the use of *IClassifier* interface. And the construction is done through *ClassifierCreator*, which is the factory; it utilises a list of Enum named *Classifiers* to decide the algorithms and implements a series of switches to construct these algorithms. The reason an Enum is used is to prevent the mistyping of classifier names, and also to allow programmers to quickly check the available algorithms. With the cooperation of *IClassifier* and *ClassifierCreator*, a user can use Rata.SSR without knowing the specific algorithms.

The design pattern also made it easier to extend the supporting algorithms. A new algorithm only has to implement the members of *IClassifier* and add configurations to *ClassifierCreator* and the *Classifiers* Enum. As can be observed from the class diagram, these classifier classes do not implement all functionalities of the classifier, but rather they unify the interface for the backend algorithm modules. The real classifications are still done through these backend modules.

5.2.2. Data Class

Data is the central element of data mining; it is required in both training and classifying processes. In our design they started being *Strokes*, which can be used by FeatureCalculator to generate a matrix of feature values. However because WEKA, DataManager and Rubine uses different data formats as shown in Figure 100, certain transformations are required.



	The graph	Individual stroke	Features
C# Ink library	Strokes	Stroke	(Not calculated)
DataManager	List<List<string>>	List<string>	string
WEKA	Instances	Instance	double
Rubine	List<List<double>>	List<double>	double

Figure 100. Data formats

List<List<String>> is the default structure used by DataManager to store the calculated features. Each *String* is a calculated feature, where a list of *Strings* stands for one *Stroke* and a list of *Stroke* is a section of *Strokes*. In WEKA the default structure used is *Instances*. Each *Instances* contains multiple *Instance* each standing for a single stroke. Each *Instance* contains a number of doubles representing the features. Often these formats need to be interchanged, which is achieved by considering the *Instances* as a 2D array of *doubles* and directly changing their values by parsing the string values to double, or vice versa. One special case is the nominal attributes; while they are stored with their

original values as *String*, they must be turned into digits when stored as *double*. Their transformation requires the header information from the targeting *Instances* to get the correct mapping.

The calculated features need to be saved for future analysis. DataManager allows the file to be written out as XML format which is supported by Microsoft Excel and which can simplify the process of analysing the data within. WEKA supports the use of both CSV and ARFF file formats, and Excel allows the manual transformation of XML into CSV. While it seems CSV is the most convenient format to use, during the implementation we noticed that ARFF files showed better performance in saving and retrieving information. As WEKA provides functionality to transform CSV into ARFF, it is included in Rata.SSR to allow the user to decide the format based on requirements.

Because these functionalities are required by both WEKA and Rubine algorithms, instead of implementing them within each, they are united into the *Data* class. Apart from implementing the structure and format, this class is also optimised for common requests from algorithms, such as retrieving the name of features and the number of entries, to ensure the information can be easily obtained.

5.2.3. User Interface

An interface is developed for the training of algorithms. As described previously the training is tightly associated with DataManager, it is implemented to have the same interface as DataManager and can be accessed from within it.

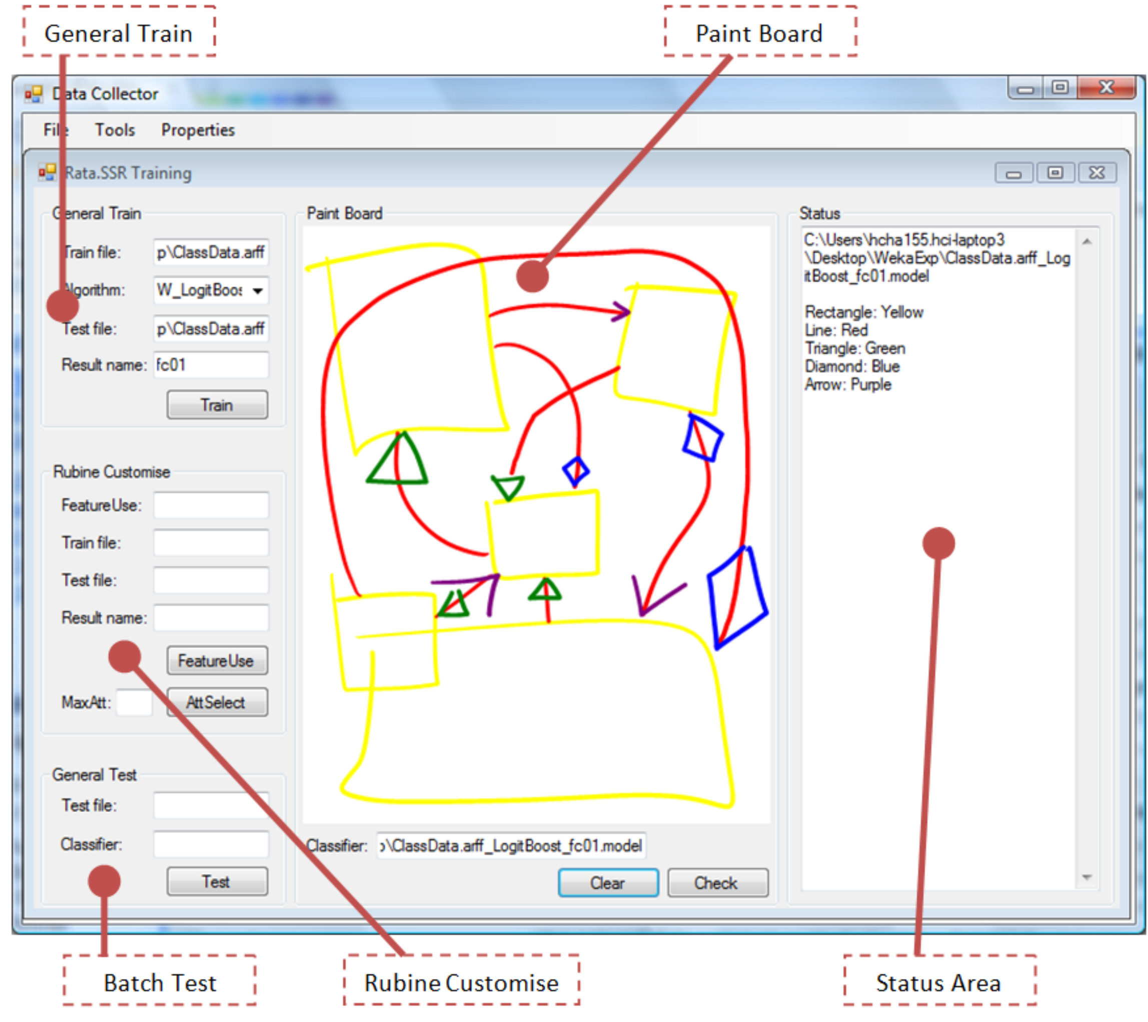


Figure 101. The Rata.SSR training interface

Because the interface is for development use, it does not consider usability or beautification. Five groupboxes are included in the interface:

Table 22. The description of each groupbox in Rata.SSR training interface

Section name	Section description
General Train	The generic training area which can train both WEKA and Rubine algorithms
Batch Test	The generic classifying area which supports IClassifier implemented recognisers
Rubine Customise	Allowing the customisation of Rubine
Paint Board	Allows manual painting and performs eager recognition
Status Area	Displays status of the program

5.3. Training

The training process is simplified by the *IClassifier* interface, which allows the interface to be unified. Through the interface in Figure 102, users can input training data to generate a classifier in the specified location. It contains 5 controls.

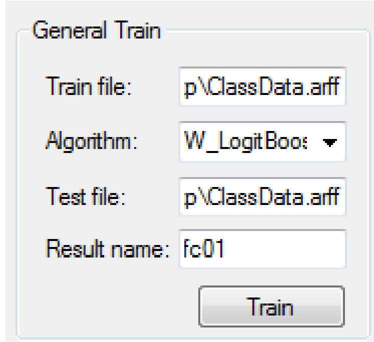


Figure 102. Rata.SSR training interface: General Train

Train file and *Test file* both opens a file loader on click, which is configured to only accept ARFF or CSV files generated by DataManager's feature calculation. Apart from the extracted FeatureCalculator, this is the only place Rata.SSR is directly linked to DataManager. This relationship is built with the passing of files. We have considered obtaining data directly from the FeatureCalculator in DataManager to avoid the I/O time; however, the component can only calculate features for the project opened in DataManager. Because in each session only one project can be opened, such implementation is inefficient. In addition, calculated features are usually stored, to avoid the need of recalculating them the next time they are required. Hence we decided to make Rata.SSR load these stored files, which allows different data to be used within the same session.

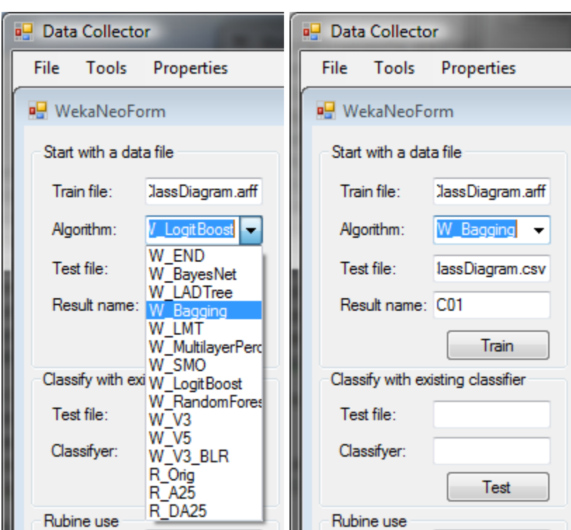


Figure 103. Rata.SSR training interface: Algorithm

The *Algorithm* ComboBox dynamically extracts the entries from the *Classifiers* ENUM specified in *ClassifierCreator* and list them for users, as shown in Figure 103. At the current stage the ones starting with W are WEKA algorithms and the ones starting with R are Rubine generated. The list can be easily modified by just modifying the *Classifiers*

Enum. The *Result name* textbox allows users to specify a name for the generated classifier. In default the classifier will be written out to where the training data resides with the specified name. Once all form controls are filled, the train button can be clicked for classifier generation.

This user interface is only an application based on the code interface implemented. If the goal is only to train a WEKA recogniser, only training data and the targeting algorithm are required, as shown in Figure 104.

```
IClassifier classifier = null;
Trainer (ClassifierCreator.Classifiers targetAlgorithm){
   string trainData = “file loaction”;
   classifier = ClassifierCreator.GetClassifier(trainData, targetAlgorithm);
   classifier.classifierWrite(”location to save the file”);
}
```

Figure 104. Training a recogniser

Once the classifier is selected, *ClassifierCreator* is aware of the classifier type it belongs to; the following processes are hence classifier dependant, and are implemented differently for different classifiers.

5.3.1. Data Process

Data need to be transformed, cleaned and processed to fit the individual requirements of different classifiers. In the case of WEKA, firstly the name of the class attribute (which is what the algorithm tries to predict) needs be specified and assigned to the training set. The reason that we used the name but not the index is because the data structure generated by DataManager may change.

Data are processed once they are loaded. Information including NAN and Infinity are substituted with empty values to ensure the validity of data. In addition, the meta-attributes (which are used in DataManager to record information such as participant number or diagram names) are changed into dummy values in order to nullify their effects in training. Although these meta-attributes normally are not selected by tree generating algorithms as they do not carry meaningful relationships, algorithms which consider all attributes will consider them and overfit the model to cause misclassification.

In contrast, Rubine is significantly affected by empty values; their presence causes the trained algorithm to perform badly. We have tried to fill the empty values with pseudo-values including the mean, the maximum and the minimum value of the category; however, none of these attempts can resolve the problem, the generated classifiers are still behaving erratically. The final decision is to remove these features, which is the implementation used by *RubineClassifier*.

A search through the data reveals that all these features are Temporal Spatial context features (TSC-features), which is sensible because there are always the starting stroke (no other stroke) and the finishing stroke (no next stroke). While it is possible to filter the information by only recognising when both the previous and the next strokes are presented, such an approach means a stroke is only recognised after another is drawn, which is not consistent with the WEKA recogniser and may cause confusion to the user. Furthermore, the starting and finishing strokes would never be recognised in this situation. To address the problem, a different classifier utilising the Rubine algorithm is built, which is the *DoubleRubineClassifier*. It contains two versions of Rubine classifier, one is trained with the original data and the other with TSC-features removed; the decision on which one to use is based on the input data.

Apart from the TSC-features, empty values can also occur due to flaws in feature calculation. While a full search and fix may be applied, such work is out of the scope of this project. Because the aim is to decide if Rubine can be improved by attribute selection, we simply ignore the instances which contain empty values by removing them from the training process; this action successfully resolves the erratic behaviour of trained classifiers. Zero values also affect the accuracy of the Rubine algorithm. They are dealt with by substituting the value with 0.000001 which is close to zero but does not break the classifier. Furthermore, Rubine cannot deal with nominal values, which are also removed.

5.3.2. Data Mining

The selected algorithms are initialised with *ClassifierCreator*. After the data are prepared, the data mining process can be started. In WEKA it involves only the calling of the *buildClassifier* function within each initialised classifier, and a classifier will be trained with the specified algorithm and data.

The Rubine algorithm used in this experiment is taken from the implementation of InkKit (Plimmer & Freeman, 2007). Figure 105a shows its simplified structure. *Classes* is the final recogniser, which contains a number of *Class*. Each *Class* contains multiple instances of *Example*. Each *Example* is created with points deduced from strokes, which will be transformed into a number of features by itself. *Class* utilises these features and form statistics for deciding the shape classification, while *Classes* compares the statistics reported by each of its *Class* and makes the final classification.

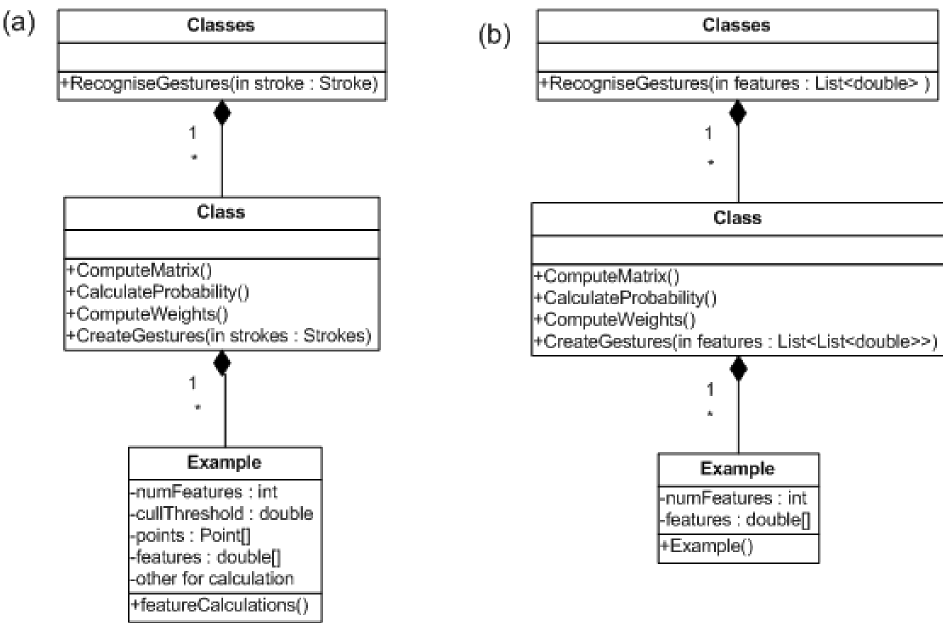


Figure 105. Simplified structure of Rubine. a) the original implementation, b)the modified version

It is unnecessary and time consuming to recalculate the features in *Example* because already we are presenting the algorithm with features. Hence the contents of these classes were modified, as shown in Figure 105b. Instead of taking strokes to train, it takes *List<List<doule>>*, which stands for a list of strokes which is already calculated into features and stored in double. All features are modified from the external source, instead of calculated within the *Example* class using *featureCalculations* method. Once the *Class* has obtained all data, it can calculate and generate the classifier itself.

As shown in section 4.5 attribute selection can optimise the performance of Rubine, hence, before the training process, we allow the choice of applying attribute selection. Training data need to be specified, together with an integer stating the maximum number of features to be selected. An *AttributeSelection* class from WEKA is initialised to do the selection, which uses the same evaluation and search method as used in section 4.5.1. A list of best features are returned, which will be trimmed by the specified maximum

number. However because the returned list is not ranked, this trimming does not consider the strength of each feature. The *DoubleRubineClassifier* follows a similar approach, but has two passes of attribute selection, one of which includes TSC-features and the other does not.

5.3.3. Preliminary Testing

Test data can be specified in General Train (Figure 102) for preliminary test to see the effect of testing data. Users specify the location of the test file, which will be used for testing when the classifier is trained.

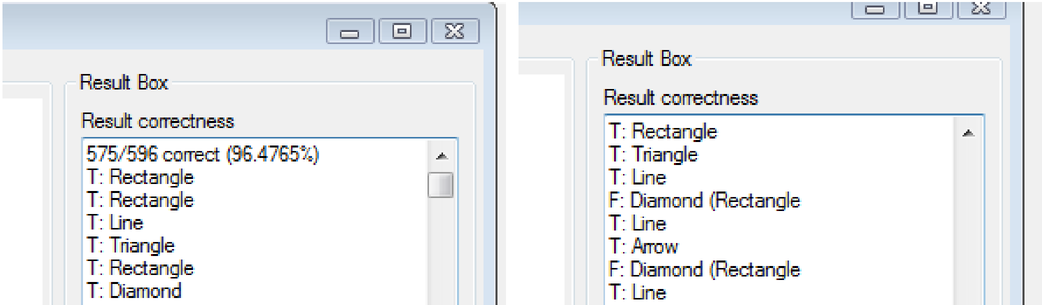


Figure 106. Rata.SSR training interface: Status area showing classifying results

The testing results are displayed in the *Status Area*, as shown in Figure 106. It shows the total number and the correctly classified strokes, which are used to calculate the percentage accuracy. For each stroke it returns information indicating whether or not it is successfully recognised, and shows the classification results. For misclassified strokes it will show the correct classification in brackets.

5.3.4. Output

A trained classifier needs to be written into a file for storage and future use. The *ClassifierWrite* method in *IClassifier* handles this functionality. Because each classifier is different, we decide to give them different filename extensions in order to distinguish between them.

For WEKA classifiers, initially we only write the classifier itself out. However WEKA is very strict about the format of data in classification; situations including the mismatching of attribute numbers or different order of nominal values can cause the data to be considered as a different format, terminating the classification process. While this information is stored in the header of the *Instances*, it cannot be extracted from the trained classifier. Hence other methods need to be applied to complete such a task. One method

attempted was to serialise the *WekaClassifier* class which contains all the information. To serialise a class, not only the class itself but all classes used by it must contain the Serializable attribute. This cannot be achieved with our implementation, because although the Java classes are transferred into C# compatible library through the use of IKVM.NET, they still do not contain the Serializable attribute.

In the end we decided to use the *InputStream* and *OutputStream* provided by Java. We followed the format WEKA used to output files – writing the classifier and the header information together into the file stream. Hence the resulting files have the same structure as that generated from WEKA explorer. To further match them the filename extension is decided as ".Model" which is used by WEKA.

Rubine is implemented in C#. We made all classes contained Serializable. Apart from serialising the classifier, the combination of features used is also serialised to ensure the loaded classifier can use the same features as used in training. The file has the extension "RBModel" for *RubineClassifier* and "DRBModel" for *DoubleRubineClassifier*.

The loading of the written classifiers can be done through *ClassifierCreator* which, based on the extension name of the loaded file, can assign the loading to the appropriate class.

```
1. IClassifier picClassifier = null;
2.
3. AlgorithmLoader{
4.   picClassifier = ClassifierCreator.GetClassifier("file location");
5. }
```

Figure 107. Loading a recogniser

5.4. Classifying

The major part of a classifier's life cycle is in the classifying process. The classifying is done through the *ClassifierClassify* method in the *IClassifier* interface. Four versions of *ClassifierClassify* are provided.

Table 23. The different versions of ClassifierClassify

Required input	Description of usage	Result
String	It takes a *string* specifying the location of data generated by DataManager; which returns the recognition result as shown in Figure 106. It exists because most collected data are written out as files. This is the technique used behind the Batch Test shown in Figure 101.	String (information of the input file)
List<List<String>>	Accepts *List<List<string>>* as it is the default data format generated by DataManager's feature calculation. By enabling this it is simple to directly recognise the calculated features in DataManager.	List<String> (results of each input)
Strokes	Accepts *Strokes* which can be directly retrieved from an *InkOverlay* presented in the Microsoft Ink library. Because *Strokes* is the standard object to encapsulate digital ink in C#, by providing this feature no modification is required for developers.	List<String> (results of each input)
Stroke	To reduce the work required from developer, the last version is supported which requires only one single *Stroke*. Because in C# a *Stroke* has link to the *Strokes* it belongs to, this utilises these data. If TSC-features exist then it will use them, otherwise it will only use the single stroke.	String (results of the input)

While all versions return classification results, these results are presented in two different formats. Most return the classification result directly, either in a list or a single *String* depending on the number of inputs. The file acceptance version returns only one single *String* which includes all the information about the particular file. This is because we believe that when users are inputting a file directly, instead of wanting the result of each separately, they are more likely to want an overall result in which they can quickly find the errors and the overall accuracy.

Although all versions undergo certain aspects of the classification, in this section we will focus on the operation with Strokes, which covers the whole lifecycle of classification.

5.4.1. Feature Calculation

When a *Strokes* instance is obtained, both WEKA and Rubine firstly send it to the *Data* class for feature calculation. WEKA utilises the header information to obtain the features required, while Rubine sends the list of features; both of these can be loaded from the saved file. Calculating only the required features accelerates the process. Furthermore, based on this information, the calculated features are in the same format as in the classifiers, hence comparisons can be performed. The meta-attributes are substituted with dummy values, as done in section 5.3.1, to ensure they match the training samples. Correction of data, such as done in section 4.1.5, is not required, because although these undesired values may cause misclassification, they will not affect the trained classifier.

DoubleRubineClassifier contains two classifiers, each using a different list of features. Both classifiers are used for feature calculation at this stage. This is because the FeatureCalculator calculates features for *Strokes,* which contains multiple *Stroke* at a time to ensure the inclusion of TSC-features. The reason all features from the TSC-feature excluded classifier are calculated is because it is easier to decide which feature belongs to which input stroke.

A special case exists for the *ClassifierClassify* version which uses the generated data file. Because data is already generated, feature calculation is not required; however the features within the file can differ from that which trained the classifier. As mentioned above, WEKA is very strict about the input data. Rata.SSR hence compares the header of the file to what is stored in the classifier, and terminates the classification process if they mismatch. For Rubine the list of feature used is applied as a reference to select data from the input features. The process can handle situations where unused features are presented; however if features required do not exist, the process terminates with notification on the situation.

5.4.2. Classification

For WEKA, because the calculated results are in *List<List<String>>*, they are firstly transformed into *Instances*. Each *Instance* within it is then sent to the classifier for classification. Because the result is in *double*, it is modified to *String* which represents the name of the classification for easier use. These recognised *Strings* are returned to users as the classification result. In the case of *RubineClassifier*, the *List<List<String>>* is modified to *List<List<double>>*. The same process as WEKA occurs afterward.

DoubleRubineClassifier has one extra step to decide which classifier to use. Initially the decision is based on the calculated features: if zero is present in some features then the TSC-features are not used. However we then found that in testing the misclassification is not caused by zero values but by the absence of data, such as the situation of the first or the last stroke where TSC-features are missing. The code is thus changed so the first and the last stroke are used with the classifier without TSC-features, and all other strokes which are drawn between use the TSC-features included version.

5.4.3. Result Operation

For the convenience of users the classification results are presented in *String*. However, it may be simpler to make further operations if *double* is required; therefore we have made two methods, *doubleToS* and *StringToD* in *IClassifier*. They handle the transition between the proper *String* results and the corresponding *double* which is the order in which these *String* are listed in the classifier.

In addition, we also allow users to retrieve a list of supported classes in the classifier. For WEKA this is retrieved from the header information. For Rubine, because initially it does not allow such information to be retrieved, this functionality is added to the *Classes* in Figure 105 to go through each *Class* contained and retrieves their names.

5.4.4. Sample usage

Figure 108 shows the code required to classify one ink stroke when it feeds in, assuming an *IClassifier* instance is either created or loaded through the specification in Figure 107.

```
1. inkOverlay_Stroke(object sender, InkCollectorStrokeEventArgs e){
2.   //…process stroke if required
3.    string recognitionResult = classifier.classifierClassify(e.Stroke);
4. }
```

Figure 108. Classify one stroke at a time

Under the simplest situation where one stroke is recognised at a time, the code presented in Figure 108 is sufficient. The stroke event is the key to recognition. Manual triggering is unsuitable because the aim is to support eager recognition. For efficiency reasons we have tried to generate timeout events so the recognition can take place during participants' thinking process; however, if too many strokes are in the stack, the delayed recognition may take a long time to compile. Hence we decided to use the stroke event, which triggers recognition when a stroke is finished. Because the recognition process is within 0.2 seconds, it is not observable. To further ensure correctness, a manual recognition button is also provided, which recognises the whole graph at once to better utilise TSC-features. This can also be considered as an implementation of lazy recognition. The reason the whole graph recognition is not provided in the stroke event is because while the recognition of one stroke is unobservable, multiple strokes can add the time.

Applying recognition to multiple strokes can be done by substituting the *e.Stroke* in Figure 108 to the *ClassifierClassify* method which takes *Strokes* and returns *List<String>*.

Paint Interface

When training a recogniser, it is likely all the collected drawing samples will be applied, because more data can better represent the targeted population. In such cases there may not be extra data prepared for testing. Hence instead of applying 10 fold cross validation, a painting interface is provided to allow users to perform a preliminary exploration of the generated recognisers.

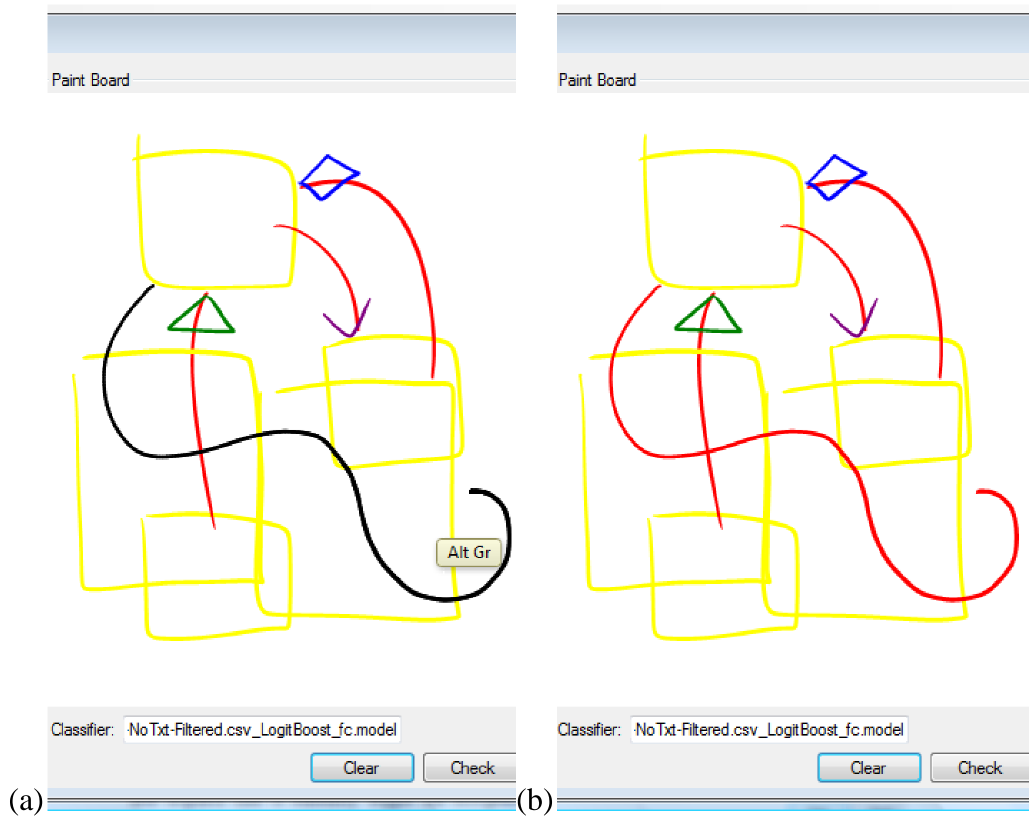


Figure 109. Rata.SSR training interface: Paint interface example use. (a. Before b. After)

Users only have to specify the classifier to be used. By applying the code in Figure 108, each drawn stroke can be recognised. This utilises the recognition techniques as shown in Figure 108, and once a stroke is recognised, it is coloured with a switch statement which selects appropriate colour based on the result of the recognition.

5.5. Extension Opportunities

As stated in Chapter 4, in WEKA individual datasets can still be further optimised, and there are still many unexplored algorithms. However, to make such an optimisation, users

not only have to conduct a series of experiments to find the trend of different settings, but also need to learn the interface of WEKA. If these options are to be implemented into Rata.SSR, the interface will be as complex as shown in Figure 110. As discussed, the freedom users have is limited in Rata.SSR; however the implementation allows them to apply the trained classifiers from WEKA in Rata.SSR. This allows experts to explore the possibilities in optimisation as well as reduces the complexity for novices.

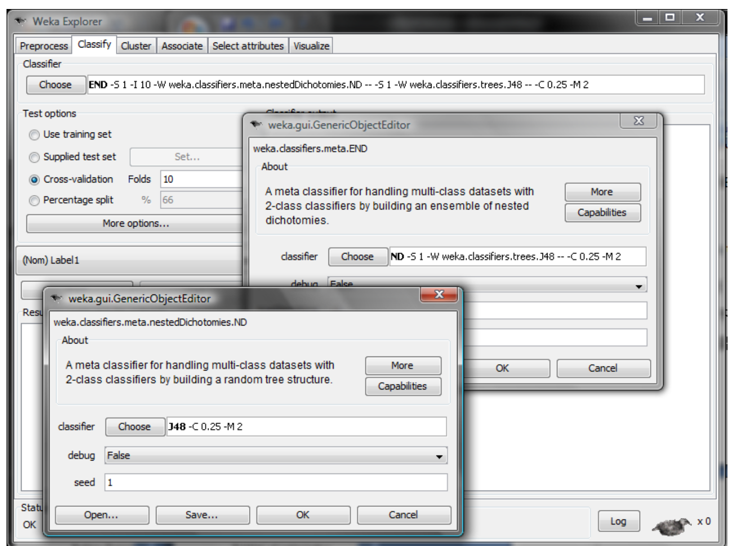


Figure 110. WEKA explorer to setup END algorithm

On the other hand, because Rubine does not have such an interface, to allow more flexibility it can be configured with the constructor of *RubineClassifier* and *DoubleRubineClassifier*.

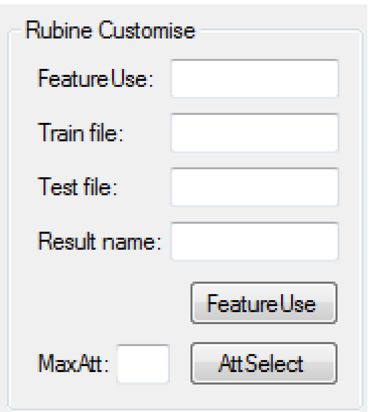


Figure 111. Rata.SSR training interface: Rubine Customise

Similarly users have to specify the file for training data. Test data and result position are optional. This interface provides two ways of training – to train with selected features, or to apply attribute selection.

For the manual decision, users can specify which features to use. The input can be single numbers or a series of numbers; for example, “82, 18, 2, 9-13” will select the eight corresponding numbers. Each number stands for a feature calculated by FeatureCalculator, for example, "82. Openness"(Blagojevic, 2009). Although these features are numbered, in the dataset generated by FeatureCalculator these features are not ordered with their numbers. For this reason we had to produce a mapping method to ensure all features are following the order when they are trained for Rubine – otherwise it randomly assigns values and causes misclassifications. Allowing the specification of features is useful in situations such as when users know some attributes are important but are never selected by attribute selection, or if users want to apply certain existing configuration such as the original Rubine. On the other hand, for attribute selection users are allowed to specify the maximum number of attributes to pick.

5.6. Summary

This chapter explains the design concepts and the underlying mechanisms of Rata.SSR. It focuses on being easy to use while retaining the flexibility. We have applied the simple factory design pattern to ensure different classifiers can be used through the same interface, which reduces the complexity of using them. Although a list of best algorithms is provided, it is possible to make customisations through different means. Several applications are shown, and the recogniser generator will be used to generate a few recognisers, which will be used to compare with the existing recognisers in the next chapter.

Chapter 6
Evaluation

To evaluate whether the goal of improving sketched diagram recognition with data mining has been met, the algorithms generated by Rata.SSR are compared with existing algorithms. Chapter 4 explained our methodological process to find quality recognisers and optimise them for best performance. Chapter 5 explained how the process is wrapped into a DLL which can be used for dynamic recogniser generation. This chapter describes the comparison between recognisers generated by Rata.SSR and several existing recognisers.

Section 6.1 outlines the setup of the evaluation. We then conducted two experiments, which are recorded in section 6.2 and section 6.3. The first compares our generated recognisers with the existing ones, to see if any statistical difference exists; the second tests the difference between different methods in data collection. The results obtained are discussed with a few more observations during the experiments.

6.1. Outline

The best algorithms are selected from our studies to be the candidates for evaluation. The experiment will be conducted through the Evaluator implemented in DataManager (Schmieder et al., 2009), comparing the selected algorithms against externally built classifiers including CALI (Fonseca et al., 2002), OneDollarRecogniser (Wobbrock et al., 2007), PaleoSketch (Paulson & Hammond, 2008), DTW(Wobbrock et al., 2007) and Microsoft Recogniser (Microsoft, 2009). The experiment is focused on accuracy.

6.1.1. Dataset for Experiment

Four datasets are used including GraphData, ShapeData, ClassData and FlowChart. The presence of FlowChart is important because the other three datasets have participated in the optimisation experiment. Although in section 4.3.11 we examined whether or not FlowChart shows the same behaviour as the other recognisers, comparing the result against other recognisers under the same experiment setting can further demonstrate how well the optimisations can be generalised.

Previous study shows that differences in drawing styles do make a difference to recognition rate (Alvarado & Davis, 2004), which can also be observed from the different performance revealed in OrderedSplitting and RandomSplitting. Hence although 10 fold cross validation can be used in Evaluator, we decide to manually make training sets and testing sets for different experiments. Each dataset is divided in two, each half containing ten participants. For each experiment of each dataset, each half will take turn to be the training and testing data, to reduce the noise. Hence a total of eight experiments are conducted for each classifier. This also reduces the computation time required.

A second study is undertaken to further evaluate the difference between in-situ-collection and isolated-collection. The details of this study will be specified in section 6.3.

6.1.2. Evaluation Process

All evaluations are done with the Evaluator implemented within DataManager (Schmieder et al., 2009). To use a recogniser in Evaluator, it needs to implement an interface to allow Evaluator to access its training and testing procedures. Each recogniser also needs to be added into a configuration file for specific setting adjustment. In addition, non-trainable recognisers have different formats and name schemes, these needs to be configured in order to ensure the result of the evaluation is meaningful.

Once these configurations are completed, the experiment can start. The Evaluator is tied to DataManager, which means users can only use the data opened in the project. Each recogniser can choose to accept a single stroke at a time or a whole diagram at a time. Users can also specify different groups of the participants for training and testing.

After setting up the datasets, the Evaluator then runs each algorithm. The trainable algorithms will be trained and tested with specified participants for each, while the non-trainable algorithms will simply be tested with the participants for testing. The result will be exported as an XML file showing information including the percentage of correctness and participant statistics. It also allows strokes to be exported in bitmap which colours the misclassified strokes for visual analysis.

6.1.3. Recogniser Configuring

As mentioned each recogniser needs to have its configuration files, to notify the Evaluator of its existence, and to ensure its classification can be understood by Evaluator. Because

non-trainable recognisers have pre-defined terminology and shape sets, they need be configured so different names can be matched. For example, if an ellipse in our data is called oval by a recogniser, we have to make sure ovals are considered to be correct recognitions for ellipses. On the other hand, because the names of shape classes in trainable recognisers are the same as in the training data, they only have to be added to notify their existence. As an exception, although Rata.SSR is trainable, because it does not follow the traditional approach of trainable classifiers, it is combined with the Evaluator with a slightly different method.

Rata.SSR

Rata.SSR can be used to train classifiers, which can then be used to classify digital ink strokes; however, the trained classifiers themselves are not trainable. Hence, to cope with eight different experiments, eight different classifiers are firstly trained with the training instances to be used in the evaluation. These trained classifiers are then loaded in the evaluation.

CALI (Fonseca et al., 2002)

Twelve shape classes are supported by CALI, as shown in Figure 112.

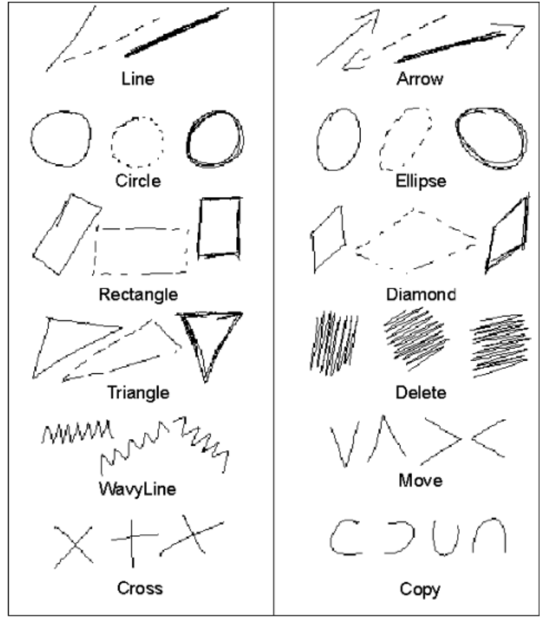


Figure 112. Shapes recognisable by CALI (Fonseca et al., 2002)

Because the classifier is hardcoded, even if the diagram domain contains only three shape classes, CALI will still try to use all twelve shape classes to classify them. Some shape

classes are more easily merged, for example, with diagrams containing only Ellipses, the results of Circles can be intuitively mapped into Ellipses.

However, more complex situations exist. For example in an experiment on FlowChart, several instances of lines and arrows are returned as Copy, which is one of the supported shape class as shown in Figure 112. Because Copy is not included in FlowChart, we have to map it into one of the classes; however, no matter which class is selected there will be a loss of accuracy. This is the disadvantage of hardcoded approach. The final decision is to map the unused class to the most frequent misclassified class. Consider the previous example; if in Copy there are two lines and four arrows, we classify Copy as arrows.

Such a configuration makes the result of CALI overly optimistic, because the decision is actually based on the result. And in real situations this result may not always occur. For example, in another FlowChart experiment, the same misclassification occurs, but this time there are more lines. Although we decided each dataset should follow the most frequent occurring classes, there is no guarantee this behaviour will happen in real usage; because after all, these are misclassified strokes. However, without such an operation the classification results will be significantly lower. On the other hand, we believe that by providing CALI with such an advantage, if it performs similarly to Rata.SSR, there is better reason to select Rata.SSR.

PaleoSketch (Paulson & Hammond, 2008)

We optimised the performance of Paleosketch as what is done in CALI. Furthermore, PaleoSketch offers the opportunity for users to turn off part of its shape recogniser. For example, a user can turn off the Rectangle classifier if no rectangle is presented in the experiment.

A very special case is the existence of Polyline and Polygons. In PaleoSketch to address the problem of not being able to recognise an infinite number of shapes, they decide to use Polyline and Polygon to provide extensibility. For a stroke which has multiple turnings and does not fit any other shape classes, it is a Polyline if it is not enclosed, otherwise it is a Polygon. Both Polyline and Polygon returns the number of turnings. For example, Polyline(2) means a line with one turning. Although the number of turnings is provided, to enable Polyline, users can only enable all numbers at the same time, and the same applies to Polygon. Hence, during the evaluation, if in a dataset there are shape

classes which are not supported in PaleoSketch, Polygon and Polyline are turned on, and different numbers will be mapped to their optimistic classes. For example, Polyline(3) may be mapped to Arrows while Polyline(4) is mapped to Lines.

Microsoft Recogniser (Microsoft, 2009)

The same optimising process used in CALI was applied in Microsoft Recogniser. However, apart from ordinary shape classes, Microsoft Recogniser recognises many shapes as “Other” and “Failed”. A large portion of strokes fall into these two categories, for example, in one experiment consisting of 335 strokes, 147 are classified as Other while 131 are classified as Failed. Because there is no specific relationship between these two classifications to any shape classes, they are both considered as misclassified, and not mapped into any result.

Once all setups are completed, the appropriate experiment data is used. In this evaluation we focus on the accuracy of individual algorithms.

6.2. Experiment against Other Classifiers

In this experiment we aim to compare the classifiers generated by data mining with the existing recognisers. For the data mining algorithms, we have selected the two best classifiers, LogitBoost and BayesianNetwork, together with the worst classifier, Bagging. Two versions of combined classifiers were also selected. They are both Voting algorithms, one with the top ranked three algorithms and another selects three top performing ones excluding BayesianNetwork, to examine the difference in performance.

Rubine algorithm was also selected in the experiment, with the original setting and the attribute selected version. We have prepared two versions of attribute selected Rubine, one selects 11 features which is the same number as what was applied in the original version, and another uses 25 features which demonstrated good performance in the Rubine attribute selection study in section 4.5.

We tested our recogniser against the five other recognisers described above. The results are shown in Table 24. As stated, each dataset is separated to two, each with 10 participants. For example, if training participants are “1-10”, then participants 11-20 are used in testing.

Table 24. Evaluation result

Training participants	GraphData		ShapeData		FlowChart		ClassData		Avg
	1-10	11-20	1-10	11-20	1-10	11-20	1-10	11-20	
WEKA classifiers									
V3_1(BN$_{Opt}$, LB, RF$_{Opt}$)	100.0	100.0	96.6	97.5	99.4	99.7	93.2	97.2	**98.0**
V3_2(LB, RF$_{Opt}$, LAD$_{Opt}$)	99.5	99.1	96.6	97.5	100.0	97.9	94.2	97.5	**97.8**
BN$_{Opt}$	99.5	100.0	94.1	97.9	98.5	99.1	91.2	94.5	**96.9**
LB	96.7	99.1	91.2	97.9	99.1	97.9	90.2	95.8	**96.0**
Bagging$_{Opt}$	97.9	97.4	93.3	97.5	93.1	94.3	90.2	95.8	**94.9**
Average	**98.92**		**96.01**		**97.9**		**93.98**		
Rubine series									
Double Rubine 25 feature	99.5	100.0	97.9	98.7	99.1	95.8	92.9	89.6	**96.7**
Attribute Rubine 25 feature	99.5	99.1	97.9	98.7	99.1	95.5	92.5	89.6	**96.5**
Attribute Rubine 11 feature	97.9	98.3	95.8	98.7	83.2	91.0	84.1	84.5	**91.7**
Rubine Original (11F)	95.8	96.6	85.8	90.4	82.6	89.8	70.4	85.5	**87.1**
Average	**98.3**		**95.5**		**92.0**		**86.1**		
Existing recognisers									
PaleoSketch	89.7	90.4	92.9	96.3	93.7	88.4	84.2	82.1	**89.7**
$1	93.0	94.9	85.0	85.8	80.2	85.3	81.8	84.5	**86.3**
CALI	87.2	87.4	83.3	84.1	88.9	79.4	81.5	75.9	**83.5**
DTW	89.7	87.8	80.4	82.9	76.3	77.6	78.5	78.6	**81.5**
Microsoft Recogniser	28.8	25.9	40.0	40.0	14.9	15.2	14.7	19.0	**24.8**
Average (Exclude MS)	**90.0**		**86.3**		**83.7**		**80.9**		

The WEKA classifiers are clearly the most accurate classifiers of all. The classification result follows the ranking in Table 16. We can see even the worst performing algorithm, the Bagging algorithm, achieved better performance compared with the existing recognisers.

A Z-test was conducted to see whether the test results are statistically significant. A total of 2252 strokes were used in the test. Between V3_1 and V3_2, the standard error is 0.00302 while the p-value is 0.508; this is not significant. However comparing V3_1 with BayesianNetwork, the standard error is 0.00332 while the p-value is 0.000922. This shows strong evidence that V3_1 has better performance than BayesianNetwork. This confirms that V3_1 is statistically superior to all algorithms with lower average performance than BayesianNetwork.

PaleoSketch, CALI and MS recognisers are all optimised toward the experiment results as explained in the previous section. Among them Microsoft Recogniser has the worst performance, which is because many instances are classified into the two categories, “Other” and “Failed”, which the optimistic mapping is not applied.

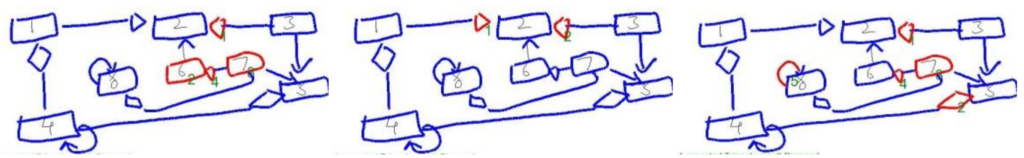


Figure 113. Result of BN, LB and Bagging on CLASS drawn by participant 17(red means wrongly recognised)

Although on average Bagging is the worst performing algorithm, it sometimes outperformed better algorithms, as is shown in Table 24. This shows it can correctly recognise some strokes when the better algorithms cannot. As an example, Figure 113 shows how three algorithms perform on the same graph. This indicates that even the worst performing Bagging algorithm can make improvement toward the better algorithms in combined classifiers such as Voting.

The revised Rubine classifier is successful. A Z-test is conducted between the Original Rubine and the 11 feature selected Rubine. The standard error is 0.00647 while the p-value is 1.15×10^{-11}, which shows attribute selection significantly improved the Rubine algorithm. While for complex diagrams they generally do not perform as well as the WEKA algorithms, they have very good performance in a dataset collected with isolated-collection such as ShapeData.

6.2.1. Accuracy of Individual Shapes

The percentages of recognition rate in different shapes are also recorded.

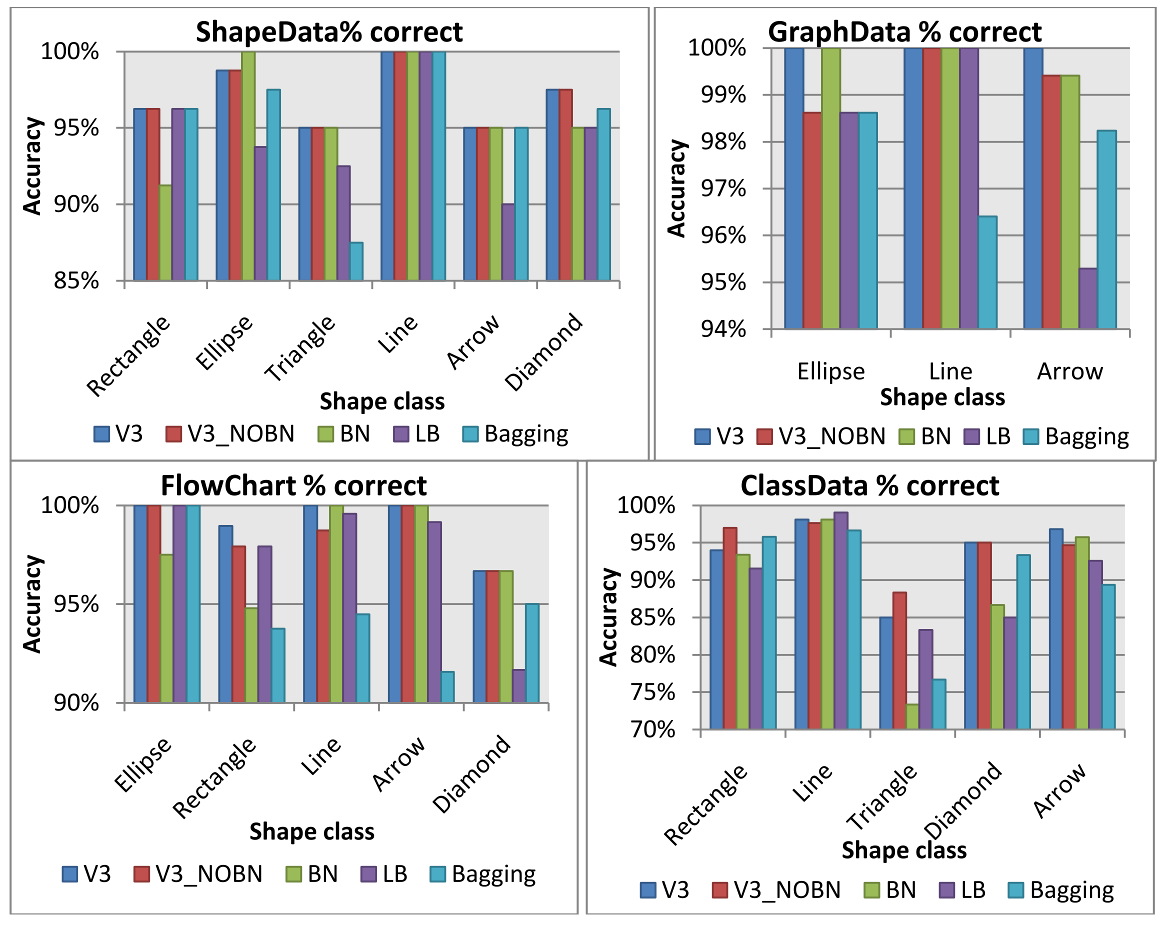


Figure 114. Recognition accuracy for individual shape classes

According to the graph, triangle is the most complex shape to recognise over all datasets. On the other hand, although in general the recognition rate follows the rankings for the algorithms, they differ in individual shapes. For example, Bagging outperformed LogitBoost in several cases including the Diamond in FlowChart, while they both outperformed the best algorithm, BayesianNetwork, in the Triangle in ClassData.

6.2.2. Performance Difference

In the evaluation, the two subsets of ClassData demonstrate significant differences in accuracy consistently. To find the reason the results are further analysed. Table 25 shows the accuracy obtained in evaluation for individual participants, together with the familiarity (how frequently used) of using the tablet and different diagrams.

Table 25. Accuracy (average of the five WEKA algorithms used in evaluation) in all datasets compared to participant skills (Skills in Likert scale, 5 used frequently <-> 1 never used before)

Participant	Tablet Skill	Shapes	Graph		Flowchart		Class	
		Accuracy	Skill	Accuracy	Skill	Accuracy	Skill	Accuracy
1	5	96.6	4	100.0	3	100.0	4	98.6
2	3	100.0	2	99.0	3	95.6	2	91.7
3	5	100.0	3	100.0	2	97.5	3	98.5
4	5	99.1	3	99.2	3	99.3	3	97.3
5	5	100.0	3	99.2	3	100.0	3	97.3
6	3	94.1	3	99.1	3	96.4	3	97.2
7	3	97.5	4	100.0	4	100.0	4	94.6
8	3	100.0	4	100.0	3	92.9	3	95.5
9	3	100.0	4	100.0	4	96.9	5	95.3
10	4	89.1	3	95.0	1	99.3	3	95.7
Avg 1-10	3.9	97.6	3.3	99.2	2.9	97.8	3.3	96.2
11	5	98.3	5	100.0	5	100.0	3	98.1
12	2	98.3	4	100.0	3	97.7	1	84.6
13	1	77.5	3	91.6	1	91.7	2	89.6
14	1	84.1	3	100.0	1	98.8	4	98.6
15	4	100.0	4	100.0	5	99.4	4	92.5
16	1	99.1	2	98.3	3	98.2	4	88.5
17	4	97.5	3	99.2	3	97.6	3	88.9
18	3	99.1	3	100.0	4	100.0	3	98.0
19	1	99.1	5	98.3	4	98.0	2	96.9
20	3	90.8	4	100.0	4	99.4	3	83.3
Avg 11-20	2.5	94.4	3.6	98.7	3.3	98.1	2.9	91.9

Although some participants who reported low skill in drawing a particular diagram do achieve low accuracy in that diagram, counter-evidences can also be found. Overall no significant correlation is observed. We then decided to visually analyse the misclassified strokes individually to see if they demonstrate different behaviour from correctly recognised ones.

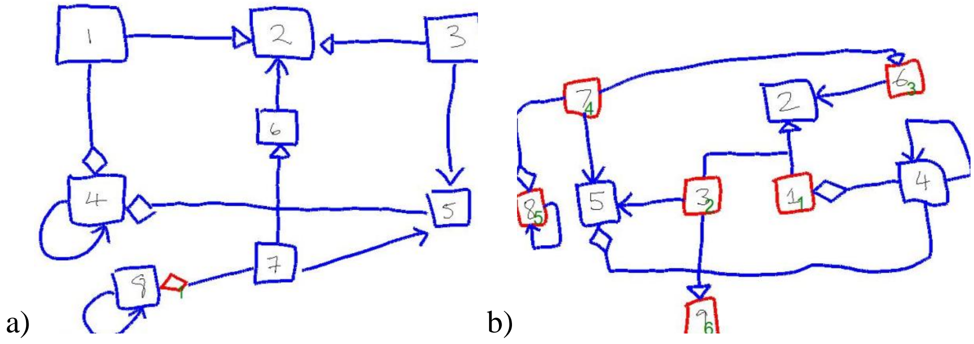


Figure 115. Two examples from LogistBoost in ClassData(1-10), strokes in red are the misclassified ones

The study shows that the difference in accuracy is due to ClassData(1-10) frequently classifying rectangles as diamond. For ClassData(11-20), there are only two misclassified rectangles: one has been drawn with a 45 degree rotation because it is close to the edge of the drawing area, which makes it resemble a diamond; and another only has three sides, which is recognised as a line. These cases can be misclassified even by human observation. On the other hand, for ClassData(1-10), we found that the misclassified rectangles are either badly formed, or are slightly smaller than the other rectangles. However not all rectangles with these characteristics are misclassified, for example, Figure 115 shows the drawings from two participants in ClassData(1-10) with LogitBoost; the size of rectangles 5, 6 and 7 in Figure 115a is not much different from the misclassified ones in Figure 115b, but they are correctly classified.

Many misclassifications exist between triangles, diamonds and arrows across different datasets. No visually observable differences are found between the correctly classified ones and the others. We believe this demonstrates that apart from visual effect, other features are contributing to the result.

Drawings from the participants 1-10 are better formed. This may be explained with their higher tablet skills. As many of the misclassified shapes are badly formed, the experience in using tablet may be important.

Overall, these problems are not occurring with all participants, but are concentrated on individual ones. As can be observed in Table 25, large differences exist between participants. It shows some participants have different drawing styles from the others. On the other hand, the difference in experience can also affect the result. Furthermore, apart from the testing data, training data can also impact on the recognition accuracy; because data mining finds relationship among training data. If the training data used are all well formed and have similar size, the trained classifier will tend to recognise this kind of information, and perform badly on data with a different style. We believe this is what happened with the different subsets of ClassData.

6.3. Using the ShapeData to Recognise Other Datasets

To evaluate the effect of isolated-collection, we used the LogitBoost algorithm and the V3 (BN_{Opt}, LB, RF_{Opt} version) algorithms to conduct the experiment. The experiment is conducted in GraphData, FlowChart and ClassData. Each dataset is divided in half, each

with ten participants to form a total of six testing sets, and each testing set contains ten participants. Training data are selected with four methods as described in Table 26.

Table 26. Different schemes for training example

Scheme name	Explanation
Original data	Training examples are from the same dataset as testing examples.
Fit Shape	ShapeData are used as training examples. Shape classes which do not appear in testing examples are removed. For example, when testing with GraphData, triangles, diamonds and rectangles will be removed
All shape	ShapeData are used as training examples.
Fit shape, TSC included	Same as FitShape, however TSC-features are not removed

Similar to the previous evaluation, if the testing data uses participant 1~10 then the training data will be trained with participant 11~20. This is applied even to ShapeData because the same participant may still have similar drawing behaviour. However, ShapeData contains fewer shapes in total compared to the other datasets; because this may affect the performance, we did extra experiments to train algorithms with the full ShapeData for each experiment involving ShapeData.

We hypothesised that the result from the original data should always be better than the others, because it considers the relationship between the elements of diagrams. Furthermore, experiments using the full ShapeData should have higher accuracy than the ones which only applied it partially, because the addition of training data increases the accuracy, as demonstrated in the splitting experiments done in section 4.3. In addition, algorithms trained with AllShape should be less accurate than FitShape, because the existence of unnecessary shape classes may confuse the recogniser (Rubine, 1991; Schmieder, 2009). Finally, the inclusion of TSC-features should decrease the accuracy, because the relationship learned in ShapeData is not applicable to other datasets.

Table 27. Comparison of diagram collection and shape collection

	Training sample	Split	GraphData		FlowChart		ClassData		Avg
			1~10	11~20	1~10	11~20	1~10	11~20	
LogitBoost	Original Data	Partial	96.7	99.1	99.1	97.9	90.2	95.8	**96.5**
	Fit Shape	Partial	94.2	97.4	96.7	88.6	75.1	81.4	**88.9**
		Full	97.5	97.9	96.1	93.7	84.2	81.7	**91.9**
	All Shape	Partial	96.7	97.4	96.4	92.5	71.8	83.8	**89.8**
		Full	94.6	96.2	95.8	92.2	82.8	81.4	**90.5**
	Fit Shape, TSC included	Partial	91.7	93.3	90.4	85.3	65.4	57.0	**80.5**
		Full	94.6	94.1	92.8	84.1	62.0	61.8	**81.6**
V3 (BN_{Opt}, LB, RF_{Opt})	Original Data	Partial	100.0	100.0	99.4	99.7	93.2	97.2	**98.3**
	Fit Shape	Partial	93.8	97.0	96.7	93.1	87.2	81.0	**91.5**
		Full	97.1	97.0	96.7	92.5	83.8	85.2	**92.1**
	All Shape	Partial	97.1	97.9	95.5	93.4	82.2	84.5	**91.8**
		Full	94.2	93.7	96.4	92.2	82.2	81.4	**90.0**
	Fit Shape, TSC included	Partial	-	-	96.7	94.9	85.9	80.0	**89.4**
		Full	98.7	96.6	97.6	92.5	78.1	79.0	**90.4**

A total of 1772 strokes were used in this study. A Z-test was performed between LogitBoost trained with Original data and with the Full FitShape (which has the highest value). The standard error is 0.00553 while the p-value is less than $1x10^{-14}$. This shows strong evidence that collecting data with in-situ-collection is superior. The difference in performance is more significant in complex datasets such as ClassData. In addition, the increment in training samples (cases were Full data are used) does increase the average accuracy in five out of six cases. However, the difference is marginal. While this may be caused by noise in the order of data, it may also reflect that the training data is sufficient to maximise the accuracy (Rubine, 1991).

The comparison between AllShape and FitShape gives no clear indication of which one is better. It may be acceptable for FlowChart and ClassData because they have only one less class than ShapeData, however, because GraphData contains only three classes, half the amount of ShapeData, we would expect a large difference. Because the training and testing samples are following the same order, the effect from noise would not be the only reason. After visually analysed the result, we found that although the AllShape did confuse some shape with non-existing shape classes, it made fewer mistakes between existing shape classes. Two possible reasons exist for this. First, while using AllShape increased the number of classes, it also increased the number of training examples; although these examples do not directly relate to the existing shape classes, they may increase the distinction between them. Second, while FitShape allows the algorithms to

focus on the existing shape classes, they may tend to find more detailed information which does not exist in the testing data, and cause overfitting.

The inclusion of TSC-features from the ShapeData shows decreased performance as expected. The behaviour is more significant with the more complex datasets. To further analyse the reasons for performance difference, we analysed the individual shape correctness. This study considers only the Original Data, FitShpae and FitShape, TSC included.

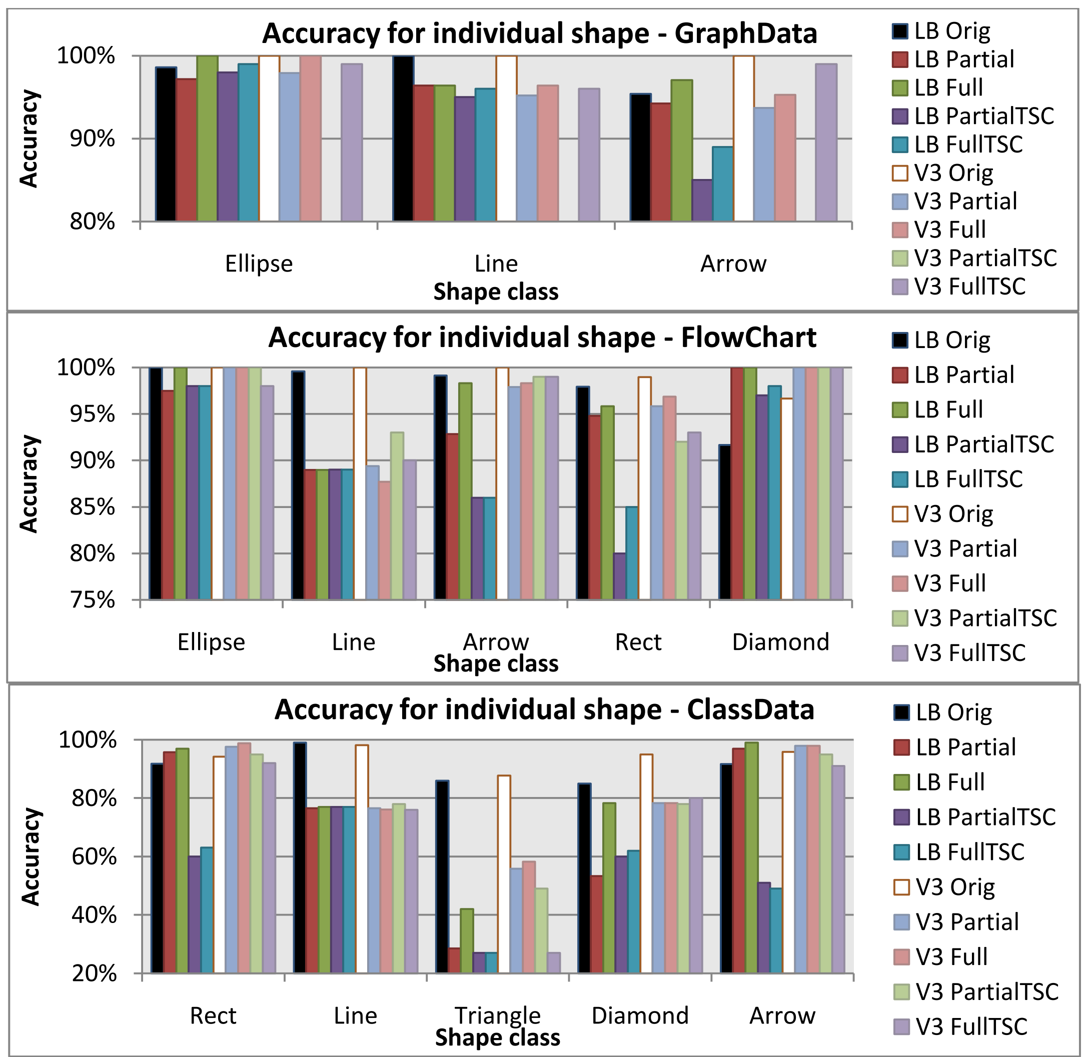


Figure 116. ShapeData comparison: Recognition accuracy for individual shape classes

While generally decrement in accuracy can be found in all situations, lines have showed consistent reduction across all datasets. This is expected behaviour because the lines drawn in ShapeData are better formed than the ones drawn in other datasets. Furthermore, connectors such as arrows and the connectors appearing in ClassData reveal similar levels of reduction, if not worse.

The inclusion of TSC-features reduces the accuracy, especially with the LogitBoost algorithm. We expect this is because LogitBoost considers more of the TSC-features than the other algorithms.

6.4. Summary

Five stroke recognisers, three generated with singular WEKA algorithms and two generated by combining algorithms with Voting, are generated. They are compared with the existing stroke recognisers. The result shows the data mining classifiers generated by Rata.SSR are significantly better than the existing classifiers. Experiment also shows Rubine algorithm is significantly improved with the attribute selection among a rich feature collection. We have also demonstrated that in diagram recognition data should be collected with in-situ-collection, which deliver much higher accuracy than isolated-selection.

Chapter 7
Discussion

The objective of this research is to explore how data mining can improve sketched diagram recognition. To simplify the problem we focused on single stroke on-line eager recognition. Four single stroke diagram sets were collected, each with different complexity, and WEKA was used to conduct the data mining process.

We choose to generate recognisers with data mining because, among the many approaches considered in the literature review, training based algorithms demonstrated both extensibility and the possibility of utilising the rich information contained in digital ink strokes. However, most studies utilised small quantities of ink information, and very few studies attempted to compare different algorithms to explain why one algorithm is selected over another. Through this study, we have found a set of good algorithms and further improved them, with a feature set consisting of 115 different features. The knowledge gained is included in Rata.SSR which can generate recognisers based on input training examples. These generated recognisers demonstrate higher accuracy than other recognisers in our evaluation.

The following discussions are based on the discoveries made in our experiments and implementation.

7.1. Data Mining

The application of a rich feature set proved effective. With the four datasets collected in this study, all data mining algorithms outperformed the existing recognisers, as demonstrated in Table 24. We believe this is not only due to the strength of algorithms, but is also caused by the capturing of relationships which were not considered previously. It may be hard to add these relationships into hard coded or template matching approaches; however, they can be easily encapsulated into features and used with unmodified training algorithms. With our implementation, new features can be easily added into DataManager, which can be immediately used with the WEKA algorithms.

On the other hand, the algorithms themselves certainly can be improved. In this study, the main emphasis of data mining is to optimise the algorithms, which includes modifying the

settings, selecting the attributes and combining the algorithms. The aim of optimisation is to allow the selected algorithms to generate best classifiers in the domain of sketched diagram recognition. For most settings, changing their values has the same effect toward all datasets. Modifying a Boolean setting causes the accuracy of all datasets to shift in the same direction, and similar trends are observed for numeric settings. For example, consider Figure 38, changing the setting brings similar behaviours to all datasets; the only difference between them is the exact value where each dataset starts to demonstrate these behaviours. Because the optimal value for each dataset is also slightly different, the average of the three datasets used is taken to decide the optimal value for each setting.

After the optimal settings for each algorithm are decided, FlowChart is used to verify the applicability of them, because it is not used in the optimisation study. Table 15 demonstrated that most algorithms agreed with other datasets. Irrefutably there are exceptions, because in this study the optimisations are only done with the average best configurations. With more data and diagrams, the nature of different datasets can be better modelled and allow Rata.SSR to automatically decide the optimised settings based on the nature of input dataset.

7.1.1. Attribute Selection

As good features can improve the recognition accuracy, we want to identify if applying only the better features among the 115 features can make further improvement. Table 14 demonstrated that for most WEKA algorithms used in this study it cannot. A possible explanation is that all the attributes used are effective, although on different levels. While attribute selection selected the better ones, the contributions of the lesser are removed. However, because experiments showed attribute selection can make improvement under certain situations, we believe there may be other reasons behind this phenomenon. Considering the algorithms applied, most are either based on tree structures or applied voting mechanisms. Tree structures natively filter out the worse features themselves; for voting algorithms, good variation between models can lead to better classifiers. Hence attribute selection does not have much benefit for these algorithms. In comparison, neither SMO nor MultilayerPerceptron applies voting technique. SMO does not explicitly filter off bad features; although MultilayerPerceptron can modify the weight to achieve a similar effect, the weightings are never adjusted to zero, which indicates the bad features can still affect the data. Hence by selecting good features to start with, better performance

can be achieved. This also improves the testing time, because the number of features to be considered is reduced. However, because their increment in accuracy is very limited, compared with the increased training time, we believe that with the experiment setups in this research there is no real advantage to apply attribute selection to any of the algorithms.

However, if more features are used, the situation can be different. Each feature requires a certain time to calculate; while each may only take a small amount of time, the combination can be large. Although with 115 features we managed to control the time within 0.1 seconds to support eager recognition, there are research applying more than a thousand features (Vogt & Andre, 2005). Under such a situation, even if accuracy is not improved, attribute selection would still be beneficial to reduce the number of features required to be calculated.

On the other hand, we believe simpler algorithms will be improved more with attribute selection, especially if it is not capable of removing the effect of bad features. Attribute selection was applied to select attributes from the original Rubine, and as shown in Table 21 the attribute selected Rubine performs significantly better. It is also very interesting that Rubine can perform well with only a really small amount of features. This also indicates the importance of finding quality features.

In addition, if we are to select a fixed number of attributes from many, it is certainly beneficial to apply data mining instead of picking them manually, which is demonstrated by the experiments conducted with Rubine.

7.1.2. Combined Algorithms

Although many algorithms are used, even after the optimisation none achieved perfect results. Because each algorithm is different, we attempt to combine them to boost the accuracy further. Figure 93 and Figure 95 show the optimisation depends on the selected combination algorithm and the algorithms to be combined. Two combination algorithms were selected, which are Voting and Stacking, and with the same selection of algorithms, Stacking is always slower and less accurate than Voting. Furthermore, usually it is even less accurate than the best performing singular algorithm.

While Stacking naturally should be slower because it requires an extra layer of training, the lack of accuracy does not match our expectation. Stacking theoretically should find the best application area for different algorithms based on their nature and strength.

Several reasons may be involved. Overfitting may occur because J48, a reasonably complex algorithm, is selected as the meta-classifier. In addition, the algorithms to be combined are all strong algorithms, and most of the error they generate is not caused by their inability in a whole area, but due to the noise in data. Thus even when assigning the best performing algorithm for a section, misclassification would still occur. In comparison, Voting is more robust because it considers the probability returned by different algorithms which may filter the noises. Because the algorithms we selected are all reasonably accurate, with the probability based Voting, the accuracy will likely be improved.

Settings were not modified for Voting and Stacking. With better configurations, such as changing the meta-classifier in Stacking or modifying the combination rule of Voting, accuracy may be further improved. In addition, only limited combinations of algorithms are tested, and according to the results it is possible to find better combinations. Overall, combination algorithms are very promising for diagram recognition and their values can be revealed with more study.

7.1.3. Difference between Algorithms

All singular algorithms are ranked according to their accuracy. If the ranking indicates one algorithm is always superior to another, we should only suggest the best performing one. However different algorithms perform differently toward different problems. The fact is supported by the result of Voting, as it can only make improvements if the other algorithms used can correctly classify shapes which the best algorithm cannot.

The accuracy reports of individual shapes are also considered. According to Figure 113 and Figure 114, even the worst performing algorithm, Bagging, can outperform the top two algorithms under certain situations. This shows that all algorithms have their potentials in different domains. More study may be able to find the correlation between different algorithms and different attributes of diagrams. Furthermore the experiment result also suggests that the Voting formed by combining the top three algorithms may not be the best configuration, as even the worst performing algorithm has the potential to further improve the top performing two algorithms. Figure 93 shows one case where the combination of three better algorithms is not performing as well as changing one of the participating algorithms to a weaker one.

Overall, the results suggest data mining is a successful approach for diagram recognition. The accuracy is improved, and with the support from WEKA it is possible to use different algorithms for different problems; however, due to the limited scale of our dataset, many observations require further investigation.

7.2. Nature of Diagrams

Experiments showed different algorithms perform differently toward different data. In fact, data is another important element in data mining apart from algorithms. If these correlations between the nature and algorithms can be identified, it will be easier to find suitable algorithms for a given diagram. In our data collection, two measures were used to divide the nature of the collected data – the complexity represented by the number of shape classes, and the way data is collected.

7.2.1. Complexity

Nothing reveals a diagram better than how it looks. The complexity is perhaps the most direct attribute one can observe. Such attributes can be broken down into many factors. Past research (Schmieder, 2009) showed that more shape classes lead to lower average accuracy. Table 24 shows that across all algorithms GraphData (3 classes) always has better performance than FlowChart (5 classes), which is always better than ClassData (5 classes). Although the behaviour of GraphData supports the hypothesis, the difference between FlowChart and ClassData cannot be explained with the same reasoning.

To analyse the reasons, we performed an experiment to find the percentage accuracy of each shape class, as shown in Figure 114. Although in ClassData all shapes have reduced accuracy compared with FlowChart, triangles, diamonds and arrows are reduced the most. The accuracy decrement for triangle and diamond may be reasonable as they are smaller hence harder to draw; however, arrow which has the same size as shown in FlowChart also showed significant reduction in accuracy.

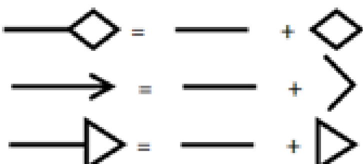


Figure 117. Connectors in ClassData

Figure 117 shows the three shape classes poorly recognised by ClassData. They are all connector ends. During the data collection, we have noticed that when drawing a connector, most people plot shapes immediately after lines. If such an observation applies to the data, TSC-features would be important; as for FlowChart, as long as lines are correctly recognised, there exists a great chance the next shape would be an arrow. In comparison, even if a line is correctly recognised, in ClassData algorithms still need to decide between three connector shapes. Such a situation may be improved by combining recognisers with multiple passes; for example, combining and calling all the connector shapes "Connector" and creating another classifier to perform classification on these connectors, such as is done by Lank et al. (2000).

The structure of the diagram can also contribute to the performance difference. During the data collection, one participant reported the design of ClassData collection is too complex, which could be the common feeling among the participants as suggested by Figure 25. More concentration would be required in the structure of the diagram rather than the shapes themselves, which may result in distorted shapes and more style variations.

This complexity problem is also related to another observation. According to Table 24, the results in ShapeData (6 classes) are similar to, or better than, FlowChart. Similar results can be found in section 4.3 where for some algorithms it even out performed GraphData. Comparing ShapeData with other datasets, the shapes contained appear to be better drawn than the shapes in other datasets. From observation in the data collection process, participants can focus on drawing the shapes, which is opposite to the ClassData case explained. The main reason is because ShapeData is collected with isolated-collection. Furthermore, they do not have to alter the shape to fit them into a graph (for example, compress a shape to avoid it touching a line). This shows that even using the same algorithm with the same number of shape classes, 2D gesture recogniser would achieve higher accuracy than graph recogniser, because samples would be drawn more tidily.

7.2.2. Collection Method in Diagram Recognition

As demonstrated, isolated-collection results in higher accuracy for isolated shapes. However would data collected this way be usable to train a diagram recogniser? This topic is still open for discussion; while some research suggests it can reduce the accuracy

(Schmieder et al., 2009), others believe there exist no significant difference (Field et al., 2009).

According to Table 27, using ShapeData to train diagram classifiers significantly decreases their accuracy. The results also suggest that the more complex the diagram is the less accurately it is recognised by ShapeData. Initially we thought the reason is that the lines are incorrectly recognised, because all lines plotted in ShapeData are straight while more turning points appear in other datasets. However Figure 116 shows that although lines do have reduced accuracy, other shape classes also display such reduction. The problem occurs not only within the lines which can be solved by applying semantics, but also affects all shape classes.

We believe this is because features in these datasets cannot be captured. The temporal and spatial information of those data certainly does not exist within ShapeData, neither do the geometric relationships, such as "the diamonds in ClassData are usually smaller, because they are connector shapes". Hence for datasets which contain more complex relationship, the accuracy would decrease.

On the other hand, if none of these temporal, spatial or geometric features are considered, the accuracy difference will be limited, which may explain why the study of Field et al. (2009) did not find significant difference. Their experiments were conducted with an image based recogniser (Kara & Stahovich, 2005), a template based recogniser (Wobbrock et al., 2007) and the original Rubine (1991). According to the report, none of them applied any TSC-features. However if data are to be collected with isolated collection, for example to be applied in a gesture system, TSC-feature should be excluded, to avoid the possible confusion they may bring. As shown in Table 27, their inclusion can cause reduction of accuracy when applied to recognise other datasets, especially for algorithms which rely heavily on these features such as LogitBoost.

7.2.3. Participant Style

The discussion also brings up another question: would individual drawing style affect the accuracy of recognisers? We hypothesis it would, for example, since temporal and spatial information are important, if a participant always draw arrows before lines, which is different from most other participants, accuracy would be affected. According to the splitting experiments done in section 3.2, RandomSplitting often has higher accuracy than

OrderedSplitting. This supports the hypothesis; the performance of RandomSplitting can be slightly optimistic due to the inclusion of the style information of participants. Certainly because RandomSplitting is the average result while OrderedSplitting considers only a single case study, such a claim is not very strong; however we believe this could prove a very interesting study.

If participant style does have an effect, and RandomSplitting is more optimistic, the standard cross validation would have a similar effect, because it too randomly chooses data for training. Under such a situation we believe that to validate the true performance of a recogniser, cross validation should not randomly pick strokes, but randomly pick participants.

The different results in RandomSplitting and OrderedSplitting also demonstrates the data collected in this research cannot represent the population, or at least with lower percentage of these data, because greater difference can be observed with lower percentages. This indicates the training data should come from the user or people who have the same knowledge as the targeting user, for example those who frequently use the targeting diagram. Although no strong correlation was found with our data in Table 25, we believe that is because the skills are self reported, and also because we started collecting diagrams from experienced groups. A study may be framed to collect data from those who have never seen or heard of the targeting diagrams to test the difference.

The best accuracy would be achieved if the recogniser is only used by one user who also provides the training data. As BayesianNetwork can achieve 96% accuracy with 10% RandomSplitting (Figure 68) which considers styles from different people, we believe higher accuracy can be achieved with a single user. On the other hand, although the population cannot be generalised, according to the trend we can see more participants brings higher accuracy. Hence we believe that although style differences exist, there are elements within drawings which can be generalised given enough examples. These elements can be found with data mining.

7.3. Implementation

We wanted to improve the extensibility and reduce the cost of the implementation in forming a diagram recogniser. As described, Rata.SSR naturally improves the extensibility – new features can be easily added, new algorithms can be used and as for

clients, they can choose between different algorithms. Furthermore, the designed architecture shown in Figure 99 allows new algorithms to be easily added into Rata.SSR. In addition, although the program restricts users only to use the provided algorithms, professional users can also customise recognisers through WEKA's interface and use them with Rata.SSR.

The restriction is for simplicity reasons, to reduce the cost of forming a diagram recogniser. A user is not required to be a professional in data mining to utilise the recogniser generator. Furthermore, when compared with direct application of WEKA, much of the process is hidden in the implementation, for example the translation of language, the merging of I/O and the process of generating and using a recogniser. The cost is certainly reduced.

It is more complex to compare the cost with existing recognisers. Different recognisers provide different functionalities, which reduce the applicability of lines of code. Hard coded approaches do not need the collection of training data, which is an advantage, however, Rata.SSR does not require many training examples if is used personally, as discussed previously. Furthermore, because Rata.SSR is focused on the problem domain, there is no requirement to convert the outputs and map the result, as were done with all hard coded recognisers in 6.1.3. In addition, compared with trainable methods, the application of a rich feature set, the ability of selecting different algorithms and opportunities for optimising them (in WEKA) make it more flexible which also reduces the cost of generating a good classifier.

7.4. Limitations

Although we believe the study successfully demonstrates that data mining can improve sketched diagram recognition and that Rata.SSR does improve the process of constructing recognisers, we are aware that many limitations exist in this study. Among them the most important limitation is the data collected.

Only twenty participants participated in the data collection process, where a total of 2252 strokes are collected across four datasets. This is not sufficient to express the population mean. Furthermore, because there are only four datasets each with different attributes, many comparison results are only observations from a single study which needs further study to validate.

Another limitation is that the optimisations are hard coded. Although there is an indication that many settings are correlated with the nature of input data, because the sample sizes are too small, we could not form a dynamic changing optimisation. This limits the success of Rata.SSR. Furthermore, many experiments are only conducted with the ordinary 10 fold cross validations which may produce overly optimistic results; the true performance can be better approached with the application of participant-based cross validations, which we did not implement. Although splitting experiments were conducted, they may be affected by noise.

In addition, due to the scope of this project, only single stroke data is used. Although the result can be directly applied in 2D gesture recognition, however, it is still unnatural in diagram recognition, because in the real world usage people draw with multiple strokes.

The next chapter will conclude the project, and provide possible directions for answering the unanswered questions.

Chapter 8
Conclusion and Future Work

This chapter summarises this research, and reviews the achievements made in improving sketched diagram recognition through the application of data mining, with suggestions for future work.

8.1. Conclusion

This thesis has explored how the current state of sketch recognition can be improved. It has focused on the application of data mining to automatically train recognisers which specialised in single stroke diagram recognition.

Our review of past studies shows there are three approaches in digital ink recognition: template comparison, hardcoded and training based. Among them the training based approach is the most promising, because it is both extendable and can utilise much information from digital ink strokes. However, the features used are still limited and only a small number of algorithms are explored.

To address these problems, data mining is applied. Four different datasets were collected from 20 participants, consisting of 2252 strokes. These were collected via DataManager, which is able to generate 115 features for each stroke. The calculated features were used to analyse and optimise the nine good performing algorithms implemented in WEKA, so they can generate accurate classifiers. Furthermore these classifiers are combined with Voting and Stacking which can further increase the accuracy.

As a result, we found that BayesianNetwork, LogitBoost, RandomForest and LADTree are the top algorithms in generating recognisers for sketched diagram recognition. The accuracy can be improved with the application of Voting algorithm. We also applied data mining to select better features for Rubine, which successfully improved its recognition rate. On the other hand, the application of attribute selection did not perform well in WEKA, which we assumed is due to the nature of algorithms used.

Both WEKA and Rubine are programmed into Rata.SSR which is our C# usable recogniser generator. To simplify the process of generating a recogniser, it provides a

minimal interface with a selection of recommended algorithms to use; however, these do not limit the extensibility of Rata.SSR. It can use classifiers generated via the interface provided by WEKA, which allows advanced customisation. Furthermore, users can select to use Rubine with automatic attribute selection, or to specify the features they want to use.

The evaluation shows that the algorithms built with data mining are significantly better than the existing recognisers. Among the evaluated recognisors, the Voting which united BayesianNetwork, LogitBoost and RandomForest demonstrated the best accuracy. Rubine algorithms were also evaluated, of which a significant difference can be observed between the attribute selected ones and the original Rubine.

We have also analysed the differences between isolated-collection and in-situ-collection. The result suggests that in-situ-collection is better for data mining if the recogniser is to be used for diagram drawing, especially for complex diagrams.

The contributions of this research are as follows:

- The application of data mining to sketched diagram recognition, with analysis of several algorithms
- The improvement in Rubine algorithm through the data mining of better features
- The implementation of a recogniser generator in C# which is simple to use, accurate and extensible
- Sketched diagram recognition is therefore improved with more accurate recognisers

8.2. Future Work

Several directions for future work can be considered based on the study described above.

Data mining with sketched diagram recognition is successful, and a major reason is due to the rich feature set provided by DataManager. Hence, it is possible to improve the performance further by introducing more quality features, such as applied by Willems et al. (2009). The combination approach worked well and is capable of generating the strongest recognisers; however, the settings of both Voting and Stacking were not analysed; besides evidence suggests different combinations of algorithms may further improve the accuracy; these are promising directions to explore. The same applies to

attribute selection, which may be improved if good algorithm or better settings are selected. Although Rubine is successfully improved with attribute selection, only one method is applied, and other attribute selection methods may provide better accuracy or ranked results. Furthermore there are many algorithms in WEKA, which although they demonstrated good performance as shown in Table 3, were not selected for study. Overall, there are still plenty of possibilities in WEKA which can be explored to improve the recognition accuracy.

Although algorithms are optimised by changing their settings, these optimisations were implemented in a hardcoded manner. As shown in Chapter 4 there are relationships between the nature of datasets and the optimal settings. It will be more appropriate to find these relationships and use them to dynamically decide the sittings. Such study requires data. In this study only four datasets were collected. While we believe they provide good indications of the performance of algorithms, they are not sufficient to reveal the relationship between the nature of datasets and the different algorithms. Each dataset was created with only twenty participants, which is also a small amount that may not represent the hypothesis space. If more data are obtained, the strength and data-preference of each algorithm can be further explored.

The optimisation and ranking of algorithms were done by combining 10 fold cross validation and different versions of splitting tests. While this may reveal the relative strength of recognisers, it is not predicting the true performance. Participant styles appeared in both training and testing examples for 10 fold cross validation, which make them overly optimistic; the splitting experiments can be easily affected by noises. The best method to test sketch data is to alter 10 fold cross validation by, instead of selecting strokes randomly, selecting participants randomly. On the other hand, the results suggest participant styles do affect the performance, and if training data were from the same user the accuracy would be increased. While Rata.SSR can create a recogniser easily, it may be beneficial if certain user adaption, such as retraining functionality, is provided.

The result of this study can be directly applied in 2D gesture recognition. However, it may be too restrictive for real world sketched diagram recognition. Many variables were removed from common sketches, such as the presence of multi-stroke and text, to allow us concentrate on applying data mining. In real world diagram drawing, these elements are unavoidable. There are dividers to divide text from diagrams as well as joiners which

can transfer multi-stroke information into single stroke information. Future work could involve the development of a multi-stroke recogniser utilising these mechanisms and data mining.

It is impossible to provide a perfect sketch recogniser, because even people can make mistakes when interpreting shapes. However, the recognisers and the way to develop them can always be improved. We believe that with more input into sketch recognition, eventually it will be as accurate as human interpretation, which would then reduce much of the work in this digital era.

Appendix A:

The Questionnaire

Questionnaire

Complete only This Section before the session

I have used directed graph	Frequently	Occasionally	A couple of times	Rarely	Never
I have used flowchart diagram	Frequently	Occasionally	A couple of times	Rarely	Never
I have used class diagram	Frequently	Occasionally	A couple of times	Rarely	Never
I have used a tablet input on a computer	Frequently	Occasionally	A couple of times	Rarely	Never

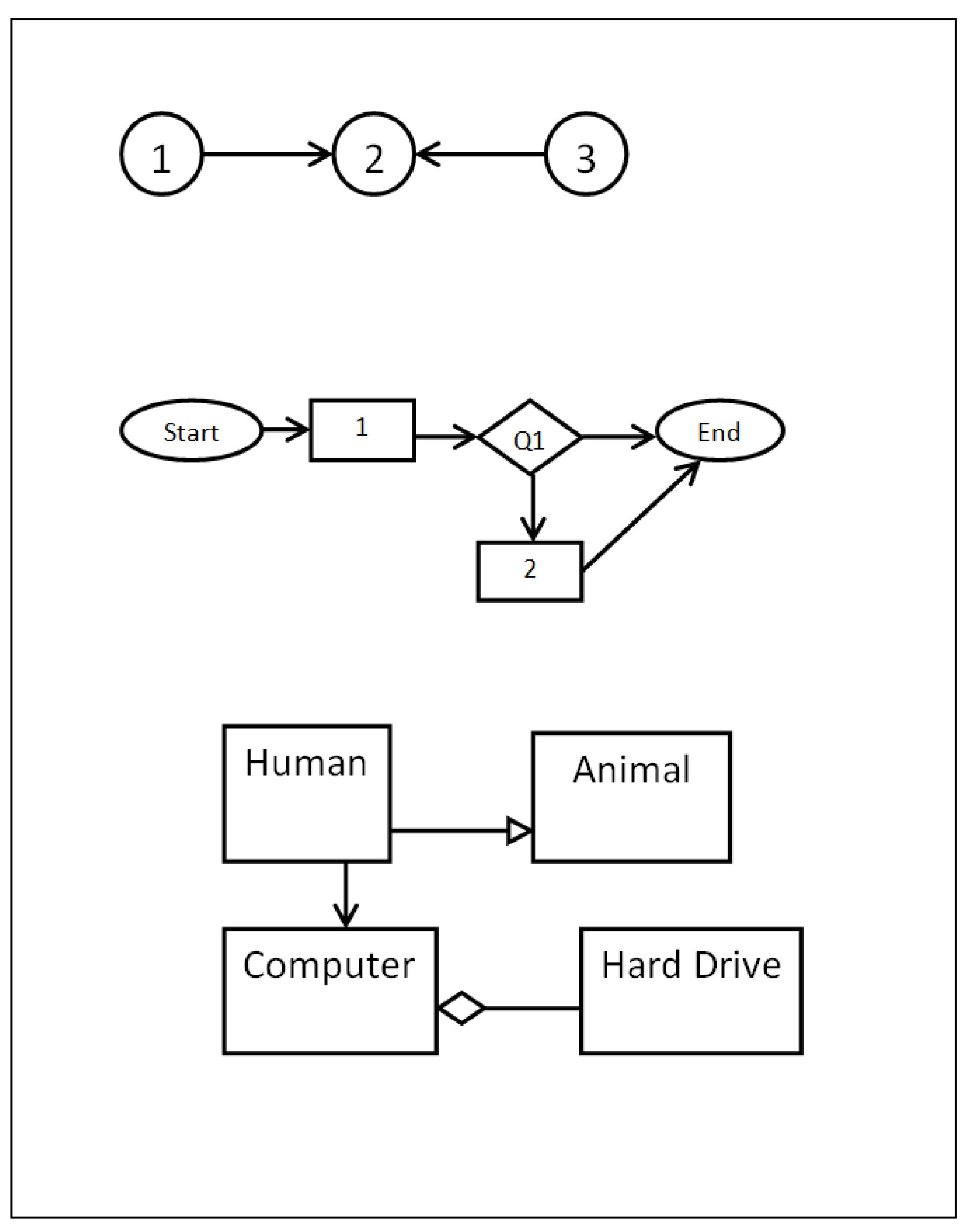
	Strongly Agree	Agree	Neutral	Disagree	Strongly Disagree
About the task					
I understand how to complete task 1					
I understand how to complete task 2					
I understand how to complete task 3					
I understand how to complete task 4					
About the environment					
Creating the diagram was easy					
Checking and editing the diagram was easy					

Comments/Recommendations:

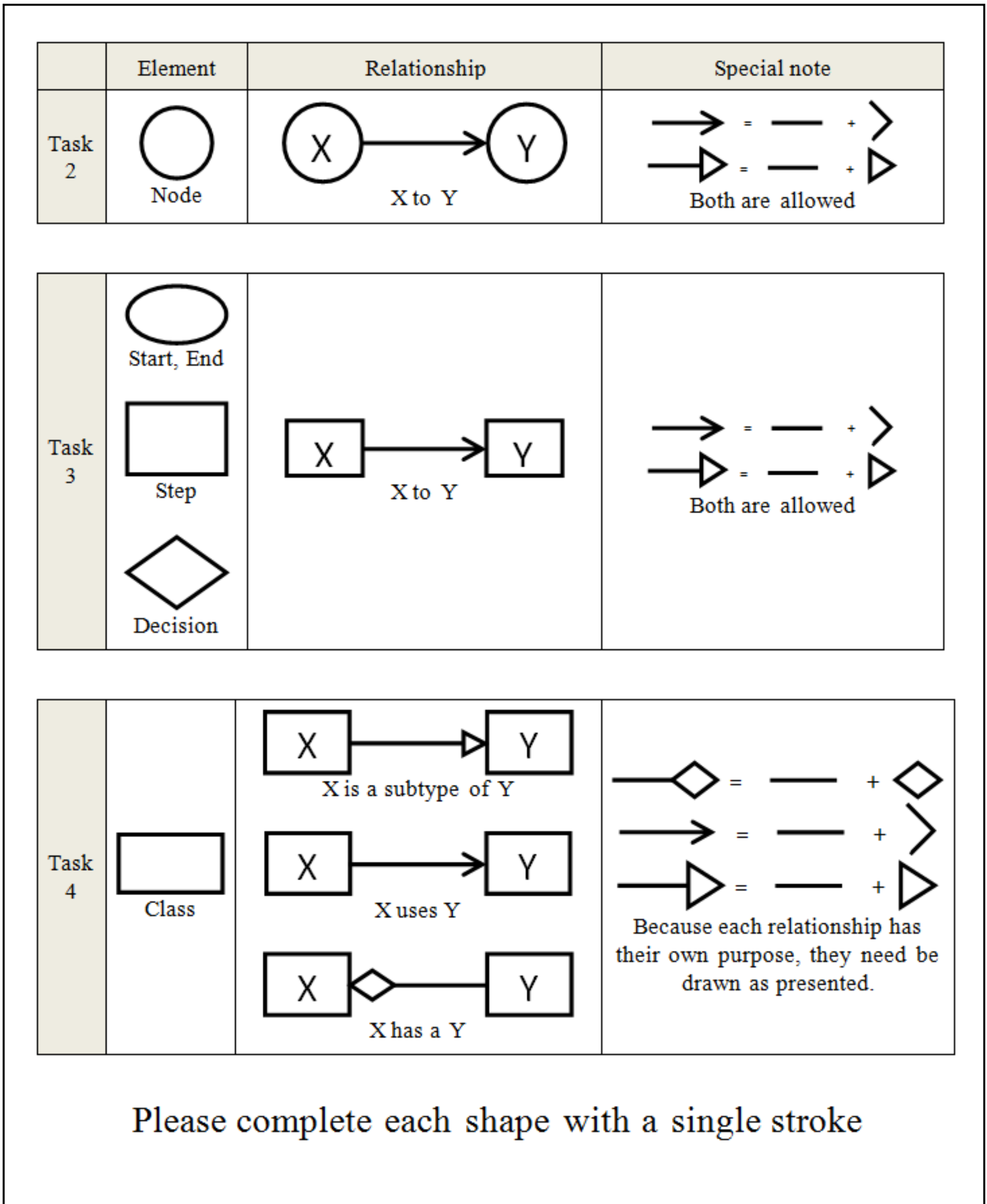
Appendix B:

The Information Sheets

- The sample graph sheet



- The dictionary sheet



Appendix C:

The Instructions

ShapeData	GraphData
Please draw shapes as described below: 4 Rectangles 4 Ovals 4 Triangles 4 Arrows 4 Diamonds	Please draw a directed graph diagram. You can create a diagram with 8 or more Nodes 8 or more Directed edges or you can follow the description below: 1 to 4, 6, 7. 2 to 3. 3 to 5. 4 to 7. 5 to 3. 6 to 1, 4.

FlowChart	ClassData
Please draw a flowchart diagram. You can create a diagram with 1 Start-node 1 End-node 4 or more Steps 3 or more Decisions 11 or more Directed edges or you can follow the description below: START to 1. 1 to Q1. 2 to Q3. 3 to Q2. 4 to 5. 5 to END. Q1 to 2, 3. Q2 to 2, END. Q3 to 4, Q1.	Please draw a class diagram. You can create a diagram with 8 or more Classes 3 or more Subtype 3 or more Has 3 or more Uses or you can follow the description below: 1 is a subtype of 2. 1 has a 4. 3 is a subtype of 2. 3 uses 5. 4 uses 4. 5 has a 4. 6 uses 2. 7 uses 5. 7 is a subtype of 6. 8 has a 7. 8 uses 8.

References

Alimoglu, F., & Alpaydin, E. (2001). Combining Multiple Representations and Classifiers for Pen-based Handwritten Digit Recognitio. *Turkish Journal of Electrical Engineering and Computer Sciences, 9*(1), 1-12.

Alvarado, C., & Davis, R. (2004). SketchREAD: a multi-domain sketch recognition engine. *Proceedings of the 17th annual ACM symposium on User interface software and technology*, 23-32.

Anderson, D., Bailey, C., & Skubic, M. (2004). Hidden Markov Model symbol recognition for sketch-based interfaces. *AAAI Fall Symposium*, 15-21.

Apte, A., Vo, V., & Kimura, T. D. (1993). Recognizing multistroke geometric shapes: an experimental evaluation. *Proceedings of the 6th annual ACM symposium on User interface software and technology*, 121-128.

Avola, D., Buono, A. D., Gianforme, G., Paolozzi, S., & Wang, R. (2009). *SketchML a representation language for novel sketch recognition approach*. Paper presented at the Proceedings of the 2nd International Conference on PErvsive Technologies Related to Assistive Environments.

Bahlmann, C., Haasdonk, B., & Burkhardt, H. (2002). On-line Handwriting Recognition with Support Vector Machines - A Kernel Approach. *In Proc. of the 8th IWFHR*, 49-54.

Bartolo, A., Farrugia, P. J., Camilleri, K. P., & Borg, J. C. (2008). A Profile-driven Sketching Interface for Pen-and-Paper Sketches. *workshop on Sketch Tools for diagramming*.

Basili, R., Serafini, A., & Stellato, A. (2004). Classification of musical genre: a machine learning approach. *in Proceedings of the 5th International Conference on Music Information Retrieval (ISMIR '04)*.

Ben-Gal, I. (2007). Bayesian Networks. In F. Ruggeri, F. Faltin & R. Kenett (Eds.), *Encyclopedia of Statistics in Quality and Reliability*: John Wiley & Sons.

Blagojevic, R. (2009). DataManager. Retrieved May 5, 2009, from http://www.cs.auckland.ac.nz/research/hci/downloads/index.shtml

Blagojevic, R., Plimmer, B., Grundy, J., & Wang, Y. (2008a). A data collection tool for sketched diagrams. *5th Eurographics Conference on Sketch Based Interfaces and Modelling*.

Blagojevic, R., Plimmer, B., Grundy, J., & Wang, Y. (2008b). Development of techniques for sketched diagram recognition. *Proceedings of the 2008 IEEE Symposium on Visual Languages and Human-Centric Computing*, 258-259.

Blagojevic, R., Plimmer, B., Grundy, J., & Wang, Y. (2010). Building Digital Ink Recognizers using Data Mining: Distinguishing Between Text and Shapes in Hand Drawn Diagrams. *IEA-AIE 2010*, in press.

Blagojevic, R., Schiemder, P., & Plimmer, B. (2009). Towards a Toolkit for the Development and Evaluation of Sketch Recognition Techniques. *Proceedings of Intelligent User Interfaces (IUI'09) Sketch Recognition Workshop, (accepted in press)*

Breiman, L. (1996a). Bagging predictors. *Mach. Learn., 24*(2), 123-140.

Breiman, L. (1996b). *Out-of-bag estimation* (Technical report): Statistics Department, Universityof California Berkeley.

Breiman, L. (2001). Random Forests. *Machine Learning, 45*(1), 5-32.

Brereton, M., & McGarry, B. (2000). An Observational Study of How Objects Support Engineering Design Thinking and Communication: Implications for the design of tangible media. *Proceedings of the SIGCHI conference on Human factors in computing systems*, 217-224.

Brieler, F., & Minas, M. (2008). *Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis*. Paper presented at the Proceedings of the working conference on Advanced visual interfaces.

Calhoun, C., Stahovich, T. F., Kurtoglu, T., & Kara, L. B. (2002). Recognizing Multi-Stroke Symbols. *AAAI Spring Symposium - Sketch Understanding*, 15-23.

Casella, G., Deufemia, V., Mascardi, V., Costagliola, G., & Martelli, M. (2008). An agent-based framework for sketched symbol interpretation. *J. Vis. Lang. Comput., 19*(2), 225-257.

Chin, K. K. (1999). *Support Vector Machines applied to Speech Pattern Classification.* Unpublished Master's thesis, Cambridge University.

Choi, H., Paulson, B., & Hammond, T. (2008). *Gesture Recognition Based on Manifold Learning*. Paper presented at the Proceedings of the 2008 Joint IAPR International Workshop on Structural, Syntactic, and Statistical Pattern Recognition.

Clarkson, P., & Moreno, P. J. (1999). On the use of support vector machines for phonetic classification. *ICASSP '99: Proceedings of the Acoustics, Speech, and Signal Processing, 1999. on 1999 IEEE International Conference, 2*, 585-588

Connell, S. D., & Jain, A. K. (2001). Template-based online character recognition. *Pattern Recognition., 34*(1), 1-14.

Connell, S. D., Sinha, R. M. K., & Jain, A. K. (2000). Recognition of unconstrained on-line Devanagari characters. *in Proc. 15th ICPR*, 368--371.

Do, T. M. T., & Artières, T. (2009). Learning mixture models with support vector machines for sequence classification and segmentation. *Pattern Recognition, 42*(12), 3224-3230

Dong, L., Frank, E., & Kramer, S. (2005). Ensembles of Balanced Nested Dichotomies for Multi-class Problems. *Knowledge Discovery in Databases: PKDD 2005*, 84-95.

Field, M., Gordon, S., Peterson, E., Robinson, R., Stahovich, T., & Alvarado, C. (2009). *The effect of task on classification accuracy: using gesture recognition techniques in free-sketch recognition*. Paper presented at the Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling.

Fonseca, M. J., & Jorge, J. A. (2000). Using Fuzzy Logic to Recognize Geometric Shapes Interactively. *FUZZ IEEE 2000., 1*, 291-296.

Fonseca, M. J., Pimentel, C., & Jorge, J. A. (2002). Cali: An online scribble recognizer for calligraphic interfaces. *AAAI Spring Symposium* 51-58.

Forbus, K. D., Usher, J., & Chapman, V. (2003). Sketching for Military Courses of Action Diagrams. *Proceedings of the 8th international conference on Intelligent user interfaces*, 61 - 68.

Frank, E., & Kramer, S. (2004). *Ensembles of nested dichotomies for multi-class problems*. Paper presented at the Proceedings of the twenty-first international conference on Machine learning.

Friedman, J., Hastie, T., & Tibshirani, R. (2000). Special Invited Paper. Additive Logistic Regression: A Statistical View of Boosting *The Annals of Statistics, 28*(2), 337-374.

Frijters, J. (2009). IKVM.NET (Version0.40.0.1) [Software]. Available from http://www.ikvm.net/.

Ganapathiraju, A. (1998). *Support vector machines for speech recognition.* Unpublished Ph. D. Thesis, Mississippi State University.

Ganapathiraju, A., Hamaker, J. E., & Picone, J. (2004). Applications of support vector machines to speech recognition. *Signal Processing, IEEE Transactions on, 52*(8), 2348-2355.

Gennari, L., Kara, L. B., & Stahovich, T. F. (2004). Combining geometry and domain knowledge to interpret hand-drawn diagrams. *AAAI Fall Symp., AAAI Press,*, 64-70.

Goel, V. (1991). *Sketches of Thought*.

Gross, M. D. (1994). Recognizing and interpreting diagrams in design. *Proceedings of the workshop on Advanced visual interfaces*, 88 - 94.

Gross, M. D., & Do, E. Y.-L. (1996). Ambiguous Intentions: A paper-like interface for creative design. *UIST 1996*, 183-192.

Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., & Witten, I. H. (2009). The WEKA Data Mining Software: An Update. *SIGKDD Explorations, 11*(1).

Hammond, T., & Davis, R. (2002). Tahuti: A Geometrical Sketch Recognition System for UML Class Diagrams. *In AAAI Spring Symposium on Sketch Understanding*, 59-68.

Hammond, T., Eoff, B., Paulson, B., Wolin, A., Dahmen, K., Johnston, J., et al. (2008). Free-sketch recognition. putting the chi in sketching. *CHI '08 extended abstracts on Human factors in computing systems*, 3027-3032.

Herot, C. F. (1976). Graphical input through machine recognition of sketches. *ACM SIGGRAPH Computer Graphics, 10*(2), 97-102.

Holmes, G., Pfahringer, B., Kirkby, R., Frank, E., & Hall, M. (2002). *Multiclass Alternating Decision Trees*. Paper presented at the Proceedings of the 13th European Conference on Machine Learning.

Hse, H. H., & Newton, A. R. (2004). Sketched Symbol Recognition using Zernike Moments. *Pattern Recognition, 2004. ICPR 2004. Proceedings of the 17th International Conference on, 1*, 367- 370.

Hse, H. H., & Newton, A. R. (2005). Recognition and beautification of multi-stroke symbols in digital ink *Computers & Graphics, 29*(4), 533-546.

Hutton, G., Cripps, M., Elliman, D. G., & Higgins, C. A. (1997). A strategy for on-line interpretation of sketched engineering drawings. *Fourth International Conference Document Analysis and Recognition (ICDAR'97)*, 771-775.

Jain, A. K., Griess, F. D., & Connell, S. D. (2002). On-line signature verification. *Pattern Recognition, 35*(12), 2963-2972.

Jiang, W., & Sun, Z.-X. (2005). HMM-based on-line multi-stroke sketch recognition. *Machine Learning and Cybernetics, 2005., 7*, 4564- 4570.

Johnson, G., Gross, M. D., Hong, J., & Do, E. Y.-L. (2009). Computational Support for Sketching in Design: A Review. *Foundations and Trends® in Human–Computer Interaction, 2*(1), 1-93.

Jorge, J. A., & Fonseca, M. J. (1999). A Simple Approach to Recognise Geometric Shapes Interactively. *Selected Papers from the Third International Workshop on Graphics Recognition, Recent Advances*, 266 - 276

Juang, B. H., & Rabiner, L. R. (1991). Hidden Markov models for speech recognition. *Technometrics, 33*(3), 251-272.

Kara, L. B., & Stahovich, T. F. (2004). Hierarchical Parsing and Recognition of Hand-Sketched Diagrams. *UIST '04*, 13-22.

Kara, L. B., & Stahovich, T. F. (2005). An image-based, trainable symbol recognizer for hand-drawn sketches. *Computers & Graphics, 9*(4), 501-517.

Keerthi, S. S., Shevade, S. K., Bhattacharyya, C., & Murthy, K. R. K. (2001). Improvements to Platt's SMO Algorithm for SVM Classifier Design. *Neural Computation, 13*(3), 637-649.

Klautau, A., Jevtic, N., & Orlitsky, A. (2002). Combined binary classifiers with applications to speech recognition. *In Seventh International Conference on Spoken Language Processing*, 2469-2472.

Kohavia, R., & John, G. H. (1997). *Wrappers for feature subset selection*. Paper presented at the Artificial Intelligence 97.

Kosina, K. (2002). *Music genre recognition.* Unpublished Diploma thesis, Technical College of Hagenberg.

Krishnapuram, B., Bishop, C. M., & Szummer, M. (2004). Generative models and Bayesian model comparison for shape recognition. *Frontiers in Handwriting Recognition, 2004. IWFHR-9 2004. Ninth International Workshop on*, 20-25.

Landay, J. A., & Myers, B. A. (1995). Interactive Sketching for the Early Stages of User Interface Design *Proceedings of the SIGCHI conference*, 43 - 50.

Landwehr, N., Hall, M., & Frank, E. (2005). Logistic Model Trees. *Mach. Learn., 59*(1-2), 161-205.

Lank, E., Thorley, J. S., & Chen, S. J.-S. (2000). An interactive system for recognizing hand drawn UML diagrams. *Proceedings of the 2000 conference of the Centre for Advanced Studies on Collaborative research*, 7.

LaViola, J., & Zeleznik, R. (2004). MathPad 2: A System for the Creation and Exploration of Mathematical Sketches. *ACM Transactions on Graphics (Proceedings of SIGGRAPH 2004), 23*(3), 432-440.

Lee, W., Kara, L. B., & Stahovich, T. F. (2007). An efficient graph-based recognizer for hand-drawn symbols. *Computers and Graphics, 31*(4), 554-567.

Manjunath, B. S., Salembier, P., & Sikora, T. (2002). *Introduction to MPEG-7: Multimedia Content Description Interface*. New York, NY, USA: John Wiley \& Sons, Inc.

Microsoft. (2009). Ink Analysis API Usage. Retrieved December 20, 2009, from http://msdn.microsoft.com/en-us/library/ms702484%28VS.85%29.aspx

Norowi, N. M., Doraisamy, S., & Wirza, R. (2005). Factors Affecting Automatic Genre Classification: An Investigation Incorporating Non-Western Musical Forms. *Proceedings of Sixth International Conference on Music Information Retrieval*, 13-20.

Oudeyer, P.-Y. (2003). The production and recognition of emotions in speech: features and algorithms. *International Journal of Human-Computer Studies, 59*(1-2), 157-183.

Ouyang, T. Y., & Davis, R. (2009). *A visual approach to sketched symbol recognition*. Paper presented at the Proceedings of the 21st international jont conference on Artifical intelligence.

Patel, R. V. (2007). *Exploring better techniques for diagram recognition.* The University of Auckland, Auckland.

Paulson, B., & Hammond, T. (2008). PaleoSketch: accurate primitive sketch recognition and beautification. *IUI '08*, 1-10.

Paulson, B., Rajan, P., Davalos, P., Gutierrez-Osuna, R., & Hammond, T. (2008). *What?! No Rubine Features?: Using Geometric-based Features to Produce Normalized Confidence Values for Sketch Recognition*. Paper presented at the Visual Languages and Human Centric Computing Workshop on Sketch Tools for Diagramming.

Platt, J. C. (1998). Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines *ADVANCES IN KERNEL METHODS - SUPPORT VECTOR LEARNING*.

Plimmer, B., & Apperley, M. (2004). INTERACTING with sketched interface designs: an evaluation study. *CHI '04 extended abstracts on Human factors in computing systems*.

Plimmer, B., & Freeman, I. (2007). A toolkit approach to sketched diagram recognition. *Proceedings of HCI 2007*

Purchase, H., Plimmer, B., Baker, R., & Pilcher, C. (2010). Graph drawing aesthetics in user-sketched graph layouts. *in proc AUIC, Crpit/ACM DL*, 80-88.

Rabiner, L. R. (1990). A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition. In *Readings in speech recognition* (pp. 267 - 296). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Rigoll, G., & Neukirchen, C. (1996). A New Approach to Hybrid HMM/ANN Speech Recognition Using Mutual Information Neural Networks. *Advances in Neural Information ProcessingSystems (NIPS-96)*, 772–778.

Rubine, D. (1991). Specifying gestures by example. *Proceedings of the 18th annual conference on Computer graphics and interactive techniques*, 329-337.

Sahoo, G., & Singh, B. K. (2008). A New Approach to Sketch Recognition using Heuristic. *International Journal of Computer Science and Network Security, 8*(2), 102-108.

Schmieder, P. (2009). *Comparing Basic Shape Classifiers: A Platform for Evaluating Sketch Recognition Algorithms.* The University of Auckland, Auckland.

Schmieder, P., Plimmer, B., & Blagojevic, R. (2009). *Automatic evaluation of sketch recognizers*. Paper presented at the Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling.

Schweikardt, E., & Gross, M. D. (2000). Digital clay: deriving digital models from freehand sketches. *Automation in Construction, 9*(1), 107-115.

Schwenk, H. (1999). Using boosting to improve a hybrid HMM/neural network speech recognizer. *ICASSP '99: Proceedings of the Acoustics, Speech, and Signal Processing, 1999. on 1999 IEEE International Conference, 2*, 1009-1012

Sezgin, T. M., & Davis, R. (2005). HMM-based efficient sketch recognition. *International Conference on Intelligent User Interfaces*, 281 - 283.

Sezgin, T. M., & Davis, R. (2006). Scale-space Based Feature Point Detection for Digital Ink. *ACM SIGGRAPH 2006 Courses*.

Sezgin, T. M., & Davis, R. (2007). Sketch Interpretation Using Multiscale Models of Temporal Patterns. *IEEE Computer Graphics and Applications, 27*(1), 28-37.

Sezgin, T. M., Stahovich, T., & Davis, R. (2001). *Sketch Based Interfaces: Early Processing for Sketch Understanding*. Paper presented at the International Conference on Computer Graphics and Interactive Techniques.

Shilman, M., Pasula, H., Russell, S., & Newton, R. (2002). Statistical visual language models for ink parsing. *In AAAI Spring Symposium on Sketch Understanding* 126-132.

Smithies, S., Novins, K., & Arvo, J. (1999). A Handwriting-Based Equation Editor. *Proceedings of Graphics Interface '99*, 84-91.

Smithies, S., Novins, K., & Arvo, J. (2001). Equation entry and editing via handwriting and gesture recognition. *Behaviour & information technology 20*, 53-67.

Stahovich, T. F. (2004). Segmentation of pen strokes using pen speed. *Proceedings of 2004 AAAI fall symposium on making pen-based interaction intelligent and natural*, 152-158.

Sun, Z. X., Zhang, B., Qiu, Q. H., & Zhang, L. S. (2003). A freehand sketchy graphic input system: SketchGIS. *Machine Learning and Cybernetics, 5*, 3232-3237.
Sutherland, I. E. (1963). *Sketchpad: a man-machine graphical communication system*. Paper presented at the Proceedings of the May 21-23, 1963, spring joint computer conference.
Tappert, C. C. (1982). Cursive Script Recognition by Elastic Matching. *IBM J. Res. Devel., 26*.
Tay, K. S. (2008). Improving digital ink interpretation through expected type prediction and dynamic dispatch. *Pattern Recognition (ICPR)*, 1-4.
Ting, K. M., & Witten, I. H. (1997). Stacked generalization: when does it work? *Procs. International Joint Conference on Artificial Intelligence*, 866--871.
Trentin, E., & Gori, M. (2001). A survey of hybrid ANN/HMM models for automatic speech recognition. *Neurocomputing, 37*, 91-126.
Vogt, T., & Andre, E. (2005). Comparing feature sets for acted and spontaneous speech in view of automatic emotion recognition. *Multimedia and Expo (ICME)*, 474-477.
Waranusast, R., Haddawy, P., & Dailey, M. (2009). Segmentation of Text and Non-text in On-Line Handwritten Patient Record Based on Spatio-Temporal Analysis. *Proceedings of the 12th Conference on Artificial Intelligence in Medicine: Artificial Intelligence in Medicine*, 345-354.
Willems, D., Niels, R., Gerven, M. v., & Vuurpijl, L. (2009). Iconic and multi-stroke gesture recognition. *Pattern Recogn., 42*(12), 3303-3312.
Witten, I. H., & Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques* (2 ed.): Morgan Kaufmann.
Wobbrock, J. O., Wilson, A. D., & Li, Y. (2007). Gestures without libraries, toolkits or training: a $1 recognizer for user interface prototypes. *UIST '07*, 159 - 168
Wolin, A., Eoff, B., & Hammond, T. (2008). Shortstraw: A simple and effective corner finder for polylines. *Eurographics 2008 - Sketch-Based Interfaces and Modeling*, 33-40.
Wolin, A., Paulson, B., & Hammond, T. (2009). *Sort, merge, repeat: an algorithm for effectively finding corners in hand-sketched strokes*. Paper presented at the Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling.
Yacoub, S., Simske, S., Lin, X., & Burns, J. (2003). Recognition of emotions in interactive voice response systems. *Proc. Eurospeech, Geneva*, 1-4.
Yang, J., & Byun, H. (2008). Feature extraction method based on cascade noise elimination for sketch recognition. *ICPR 2008*, 1-4.
Yiyan, X., & LaViola, J. (2009). *Revisiting ShortStraw: improving corner finding in sketch-based interfaces*. Paper presented at the Proceedings of the 6th Eurographics Symposium on Sketch-Based Interfaces and Modeling.
Young, S. (1996). A review of large-vocabulary continuous-speech recognition. *IEEE Signal Processing Magazine. , 13*(5), 45-57.
Yu, B., & Cai, S. (2003). A domain-independent system for sketch recognition. *GRAPHITE '03*, 141 - 146