A Multilevel Search Algorithm for Feature Selection in Biomedical Data

by

Idowu Olayinka Oduntan

A Thesis

Presented to the Faculty of Graduate Studies

in Partial Fulfillment of the Requirements for the Degree

Masters of Science

in

Computer Science

Department of Computer Science

University of Manitoba

Winnipeg, Manitoba, Canada 2005

A Multilevel Search Algorithm for Feature Selection in Biomedical Data

BY

Idowu Olayinka Oduntan

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirement of the degree

Of

Master of Science

Abstract

The automated analysis of patients' biomedical data can be used to derive diagnostic and prognostic inferences about the observed patients. Many noninvasive techniques for acquiring biomedical samples generate data that are characterized by a large number of distinct attributes (i.e. features) and a small number of observed patients (i.e. samples). Deriving reliable inferences, such as classifying a given patient as either cancerous or non-cancerous, using these biomedical data requires that the ratio r of the number of samples to the number of features be within the range $5 < r < 10$. To satisfy this requirement, the original set of features in the biomedical datasets can be reduced to an '*optimal*' subset of features that most discriminates the observed patients. Feature selection techniques strategically seek the '*optimal*' subset.

In this thesis, I present a new feature selection technique - *multilevel feature selection*. The technique seeks the '*optimal*' feature subset in biomedical datasets using a multilevel search algorithm. This algorithm combines a hierarchical search framework with a search method. The framework, which provides the capability to easily adapt the technique to different forms of biomedical datasets, consists of increasingly coarse forms of the original feature set that are strategically and progressively explored by the search method. Tabu search (a search meta-heuristics) is the search method used in the multilevel feature selection technique.

I evaluate the performance of the new technique, in terms of the solution quality, using experiments that compare the classification inferences derived from the result of the technique with those derived from the result of other feature selection techniques such as the basic tabu-search-based feature selection, sequential forward selection, and random feature selection. In the experiments, the same biomedical dataset is used and equivalent amount of computational resource is allocated to the evaluated techniques to provide a common basis for comparison. The empirical results show that the multilevel feature selection technique finds '*optimal*' subsets that enable more accurate and stable classification than those selected using the other feature selection techniques. Also, a similar comparison of the new technique with a genetic algorithm feature selection technique that selects highly discriminatory regions of consecutive features shows that the multilevel technique finds subsets that enable more stable classification.

Acknowledgements

I appreciate the support, assistance, and contributions that I received to successfully complete this thesis. Particularly, I thank my advisor, Dr. Michel Toulouse for his invaluable advice, guidance, and educative ideas. Besides, I thank him for supporting me to finish the thesis successfully and on time.

I am grateful to Dr. Richard Baumgartner, Dr. Christopher Bowman, and Dr. Ray Somorjai, and the other members of the Biomedical Informatics Group at the National Research Council's Institute for Biodiagnostics for their contributions to this research and for providing the materials required for the research. I am also thankful to Dr. Sylvanus Ehikioya and the other faculty members and staff of the Department of Computer Science, University of Manitoba, for providing a great opportunity to get trained as a computer science graduate student.

I specially thank Dr. and Mrs. Olumide Makinde for their invaluable support in all respects during this M.Sc. program. I sincerely appreciate the regular counsels of my brotherly pastor and his wife (Pastor and Mrs. Okunnu) and of my mentors, Ismail Odewale and Dr. Olanrewaju Oyewola. I am also thankful to my friends at The Redeemed Christian Church of God, Winnipeg for their support.

I greatly appreciate the support and assistance of my family. I thank my parents (Mr. and Mrs E. O. Oduntan) and siblings (Olufemi, Taiwo, Bamidele, Olanrewaju, Adebowale, and Ruka) for believing strongly in me; that has always motivated my courage. The emotional support and encouragement provided by my fiancée, Oluboade Aderinto, are invaluable. I thank you for being there for me at those challenging periods that I needed just a little push to continue.

Finally, I am grateful to God for His continual presence and loving guidance that have sustained me hitherto; I cannot be thankful enough.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The rate of occurrence of terminal diseases such as cancer is increasing globally and there is need to support the conventional clinical procedures used for the diagnosis and prognosis of these diseases with automated techniques. In recent times, identifying and monitoring the progress of these diseases can be automated by exploring the intrinsic discriminatory information that exist in biomedical data generated using noninvasive techniques such as magnetic resonance spectroscopy (MRS) [20] and gene microarrays [7]. Automating disease diagnosis and prognosis can enhance the successful treatment or management of diseases by enabling the early discovery of the diseases.

Biomedical datasets acquired using noninvasive techniques are usually characterized with high dimensionality and small sample size [40]. To process and analyze these datasets using appropriate pattern recognition (PR) techniques [21], a common requirement is to reduce the dimensionality of the datasets into a size that optimizes the cost of diagnosis and prognosis without compromising the reliability and generalization of the inferences derived from the reduced form of the datasets. The focus of this research is to design and develop a meta-heuristics that performs this form of reduction. This chapter presents the motivation of the research; a comprehensive description of the problem to be solved; a preview of the contributions of the research to knowledge; and the organization of the remainder of the thesis.

1.1 Motivation

Biomedical datasets acquired using noninvasive techniques have great potentials that can be developed to enable the automation of disease diagnosis and prognosis. Automated diagnosis and prognosis can be used to support and speed-up the conventional clinical practices in order to enable the early discovery of diseases, particularly terminal disease such as cancer that can possibly be managed if discovered early enough.

Deriving diagnostic and prognostic inferences by analytically processing biomedical datasets is an important but challenging task. An inherent challenge to this task involves finding a compact representation of the datasets that enables the generation of reliable inferences using feasible computational techniques. However, this challenge and other similar challenges can be resolved using existing or innovative computing techniques.

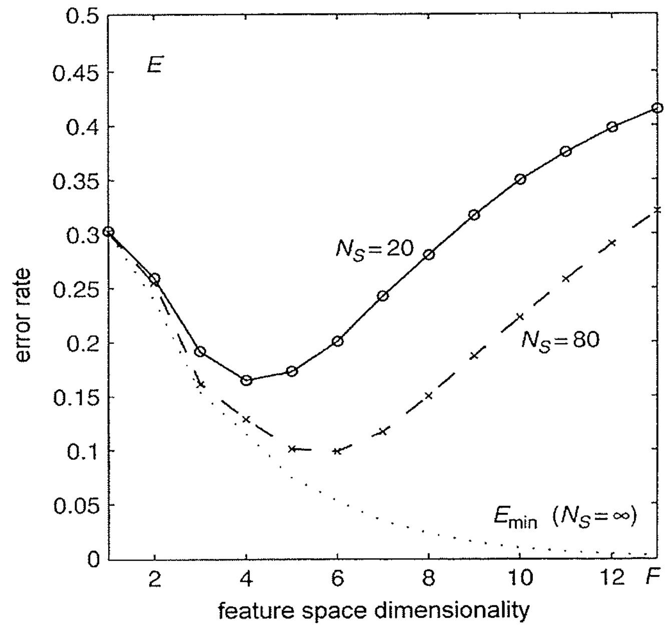
Finding a compact representation of biomedical datasets is an interesting research that can be used to drive the enhancement of existing computing techniques and possibly pioneer the innovation of new computing techniques. Besides, the results of such research can be adapted to other problem domains that relate to research areas such as machine learning in order to determine the variables that are most relevant in an inductive learning process; data mining to determine data attributes that mostly contribute to trends in huge databases; and astronomy to determine the prominent stars in the galaxies.

1.2 Problem definition and description

Biomedical datasets acquired from MRS and gene microarrays usually consist of a combination of distinctive characteristics having quantitative measures. Each distinctive characteristic is a feature. For instance, each chemical shift value in an MRS spectrum corresponds to a feature; and each expressible gene in gene microarrays also corresponds to a feature. A collection of L-ordered features that represents each given observed sample is a feature vector; a feature space is the L-dimensional space that contains the feature vector; and the dimensionality of the feature space is the number of features L that defines each feature vector in the space.

Useful and reliable information and inferences can be derived from datasets of biomedical origin by applying appropriate pattern recognition techniques. Classification (another term for pattern recognition [54]) involves separating a given set of input data or patterns into distinct classes denoted by class labels [21]. A classifier is an algorithm that performs classification. When the input data are represented using feature vectors, classifiers can predict the class label of a given feature vector by constructing implicit boundaries in the feature space to separate feature vectors belonging to different classes from one another, while optimizing a cost function [10]. The cost function is a quantitative measure of the cost of misclassification that is often estimated using the classification error rate (a measure of the number of wrongly classified samples in a given dataset).

In theory, when the class distribution densities of a classification problem are fully known, the performance of most classifiers, in terms of the classification accuracy (i.e. approximately the inverse of the classification error rate), improves with increasing number of features [18]. Figure 1.1 illustrates this idea by depicting the relationship between the classification error rate, feature space dimensionality, and sample size (i.e. the number of samples in a given dataset). As shown in Figure 1.1, the true minimum error rate $\boldsymbol{E}_{min}$ (i.e. the classification error rate when the probability densities of the feature values for each class in the dataset are fully known) of the classifier is a non-increasing function of the feature space dimensionality. That is, the discriminatory information that the classifier can use to derive the class label for a given input data accumulates with the addition of new features; hence the classification error rate decreases directly with increasing feature space dimensionality.



However, in practice the distribution densities of the feature values are rarely fully known, therefore an implied knowledge of the distribution densities that can be derived from a training dataset (i.e. a collection of samples with predefined class labels from which the parameters of a classifier can be estimated) is often used to design classifiers instead of the complete knowledge of the distribution densities. When a classifier is designed using a training dataset, the performance of the classifier is not only influenced by the dimensionality of the feature space, but also by the sample size of the training dataset. As the sample size increases, the classification error rate decreases to the minimal at a higher feature space dimensionality. For instance, Figure 1.1 shows that the classification error rate is minimal at a feature space dimensionality of 4 when the sample size N_S is 20; and similarly, the classification error rate is minimal at a feature space dimensionality of 6 when the sample size N_S is 80.

Generally, when a training dataset is used to design classifiers, the performance of most classifiers degrades as a result of the peaking phenomenon [37] when the sample per feature ratio r exceeds a certain range. The typical values of r that guarantee high classifier performance range between 5 and 10 (i.e. $5 < r < 10$); but for biomedical datasets, the value of r typically ranges between 1/500 and 1/20 [39]. The value of r can be increased to the required range either by increasing the number of observed patients (i.e. the number of samples) or by reducing the number of features. The former option is usually not practical because of the cost of the required resources and time. The more feasible option of reducing the original set of features into an '*optimal*' subset that most enhances classification performance is the *feature selection problem* [25]. This research focuses on addressing feature selection problem in biomedical dataset.

Feature selection problem can be formulated as a 0-1 integer programming problem [50], a class of combinatorial optimization problems wherein the decision variables can only be either 0 or 1. In applications to biomedical data, feature selection appears in two different combinatorial formulations. For the first formulation, feature selection involves finding a subset of features of fixed cardinality m (in the range $5 < r < 10$) that yields the lowest classification error rate for a given classifier. For the second formulation, feature selection involves seeking a subset of features with the smallest cardinality such that the classification error rate is below a given threshold. These formulations can be stated as follows:

Formulation 1:

Given an original set of L features $\boldsymbol{F} = \{f_1, f_2, f_3 \ldots f_L\}$, find a proper subset $\boldsymbol{f}$ of $\boldsymbol{F}$ such that $|\boldsymbol{f}| = m$ and $\boldsymbol{C}(f)$ minimizes the classification error rate $\boldsymbol{e}$ of a given classifier when presented with the feature subset $\boldsymbol{f}$. That is:

$$min\ \boldsymbol{C}(f) = \boldsymbol{e}(f) \qquad (1)$$

$$such\ that\ \boldsymbol{f} \subset \boldsymbol{F},\ |\boldsymbol{f}| = m,\ m < L$$

Formulation 2:

Given an original set of L features $\boldsymbol{F} = \{f_1, f_2, f_3 \ldots f_L\}$, find a subset $\boldsymbol{f}$ of $\boldsymbol{F}$ such that the classification error rate $\boldsymbol{e}$ of a given classifier when presented with $\boldsymbol{f}$ is less than a given error threshold t and $\boldsymbol{C}(f)$ minimizes the cardinality of $\boldsymbol{f}$. That is:

$$min\ \boldsymbol{C}(f) = |\boldsymbol{f}| \qquad (2)$$

$$such\ that\ \boldsymbol{f} \subseteq \boldsymbol{F},\ \boldsymbol{e}(f) < t$$

In both formulations, the optimal solution of the feature selection problem $\boldsymbol{f}$ is known as the *optimal subset* (i.e. the subset that most enhances the accurate classification of any given sample with similar feature vector dimensions as the optimal subset).

The problem formulations in (1) and (2) can be solved exactly by exhaustively enumerating the subsets in the feature search space. For the problem formulation in (1), the $\begin{pmatrix} L \\ m \end{pmatrix}$ different subsets having cardinality m can be enumerated and evaluated, and the subset having the 'best' evaluation can be regarded as the *optimal subset.* For the formulation in (2), the subsets in the 2^L different subsets having a classification error rate that is less than a given threshold t can be enumerated, and the subset having the smallest cardinality amongst the enumerated subsets can be regarded as the *optimal subset.* However, this approach is impractical for solving either formulation of the feature selection problem except for very small values of L and m (e.g. $20 < L$ and $m < 10$) [53]. In biomedical datasets, the typical value of L is in thousands and the desired value of m is usually in units. For instance, the biomedical dataset that is used in the experiments of this research consist of a feature space having dimensionality $L = 1500$. For the first formulation, if the desired cardinality of the optimal subset $m = 10$, the original solution space of the problem instance for this dataset consists of $\binom{1500}{10}$ (i.e. about 10^{26}) feature subsets having a cardinality of $m = 10$. For the second formulation, there are possibly 2^{1500} feature subsets that can be evaluated in order to find the *optimal subset.* Finding the optimal subset amongst the 10^{26} or 2^{1500} feature subsets in the solution space using implicit or explicit enumeration is intractable [12]. This thesis focuses on techniques that seek a *near-optimal subset* (i.e. the subset that is as close as possible to the *optimal subset* in terms of the discriminatory capability) within practicable computational time.

To solve the feature selection problem in a practical way, many feature selection techniques have been developed using search algorithms that enable the selection of *near-optimal subsets.* Usually, these techniques can be configured to handle the two formulations of the feature selection problem. Techniques based on heuristics such as the greedy-like sequential search algorithms [1], and meta-heuristics such as genetic algorithm (GA) [38] and tabu search [53] have been proposed. These techniques have been adapted or possibly enhanced appropriately to suit feature selection problem in particular types of biomedical data. For instance, Nikulin et al. [33] proposed a GA-based feature selection technique that is primarily aimed at biomedical MRS data wherein there is evident correlation amongst adjacent features; but this technique is inappropriate for some other forms of biomedical data such as microarrays wherein such correlation may not exist [39]. Also, feature selection techniques such as in [3, 11] that focus primarily on microarray data are usually not flexible enough to exploit the evident correlation that exist amongst features in MRS data. There is a need for a feature selection technique having an underlying search method that is flexible enough to effectively adapt to the different types of biomedical data, and strategically seeks an optimal subset that enhances the performance of a classifier.

In this thesis, I propose a new feature selection technique – *multilevel feature selection* - that addresses this need. Using the hierarchical structure that is inherent to multilevel methods [48], the technique creates a framework that can adapt easily to different forms of biomedical dataset. The technique is presently designed and configured to address the first formulation of the feature selection problem (i.e. formulation (1) above).

Given a biomedical dataset, the *optimal subset* f derived for an instance of formulation (1) above can be used to create a compact representation of the dataset such that a classifier that is designed using f easily, accurately and reliably separates the samples in the dataset into definite classes. The result of such separation can be used for disease identification purposes. The *optimal subset* derived for an instance of the second formulation can be used to identify biomarkers (i.e. the features or substances that primarily determine the presence or extent of a disease) in the observed samples

1.3 Preview of research contributions

This research contributes to knowledge in two folds. Firstly, the research furthers the evolving investigation on adapting the multilevel paradigm to solve combinatorial optimization problems. I provide empirical inferences that describe how configuration parameters influence the performance of search methods that are based on the multilevel paradigm. I also perform experiments that generate results that provide a basis to compare the performance of the multilevel-based search methods with greedy-like search methods and other meta-heuristics search methods such as tabu search in application to the feature selection problem in biomedical data; although, the inferences derived may not generically compare the performance of these search methods in application to other forms of combinatorial optimization problems. Secondly, this research creates a new feature selection technique that is based on multilevel search paradigm. The technique is presently designed to address the feature selection problem in biomedical data. The novelty of the new technique is shown in the potential capability of the technique to flexibly adapt to different forms of biomedical datasets.

1.4 Thesis organization

The remainder of this thesis is organized as follows. Chapter 2 provides background information and a review of related research work. Chapter 3 describes the concept of the search framework (i.e. multilevel search) underlying the proposed feature selection technique. In Chapter 4, I present the newly proposed multilevel search feature selection technique. Chapter 5 provides the details of the experiments, the biomedical datasets used in the experiments, and a discussion of the results generated from the experiments. Chapter 6 presents the conclusion derived from the research and the possible directions of future work.

Chapter 2

Background and Related Work

To provide a proper understanding of the feature selection problem in the context of this research, a basic knowledge of the characteristics of the common forms of biomedical datasets that are acquired using noninvasive techniques is imperative. A survey of the existing feature selection techniques that can be used to select the optimal or near-optimal subset from these datasets is also necessary. In this chapter, I describe the common forms of biomedical datasets that can be acquired using modern noninvasive techniques (magnetic resonance and microarray), the common characteristics of these datasets, the challenges of processing the datasets and the development trend of the existing feature selection techniques for processing the datasets.

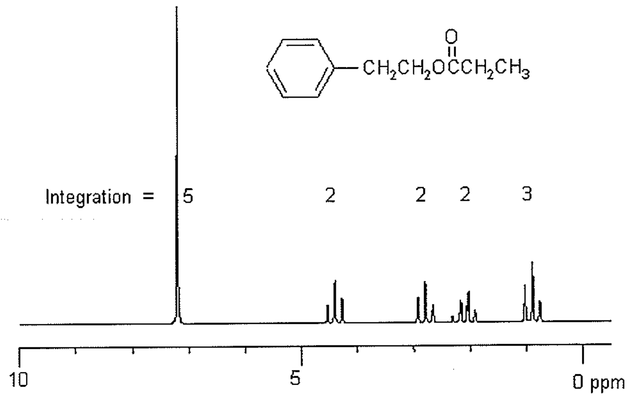
2.1 Acquiring biomedical datasets

Biomedical and clinical sciences benefit from advancements in noninvasive techniques for acquiring data to study the properties of the chemical and biological components of cellular organisms. Modern techniques such as magnetic resonance (MR) and microarrays provide noninvasive means of measuring the concentration of chemical components within tissues and the expression levels of genes respectively. An ordered collection of these measures for similar observations (i.e. biomedical dataset) can be used for diagnostic and prognostic purposes. This section briefly describes the concepts and techniques of MR and microarray and the characteristics of the biomedical datasets acquired using these techniques.

Magnetic resonance, originally known as nuclear magnetic resonance (NMR), involves interacting electromagnetic radiation (within the radio frequency bandwidth) with a collection of nuclei in a strong magnetic field. Subjecting the nuclear particles of some elements to a strong magnetic field causes the particles to precess at a characteristics angular frequency (lamour frequency) and generate a resultant magnetic moment directed along the magnetic field.

Applying electromagnetic radiation at a frequency that equals the oscillating frequency of the nuclear particles causes resonance, and some of the particles are excited from a lower to a higher energy level. Besides, the nuclear particles' magnetic moments coherently track the oscillating magnetic field of the radiation and form precessions around the field. Consequently, the resultant magnetic moment splits into components that oscillate along the plane of the radiation's magnetic field. These oscillating magnetic components are transformed into a radio frequency signal (MR signal) having a frequency that equals the precession frequency of the nuclei. This signal can be transformed (e.g. induced into a coil to generate alternating current) and amplified to create an MR image or MR spectrum.

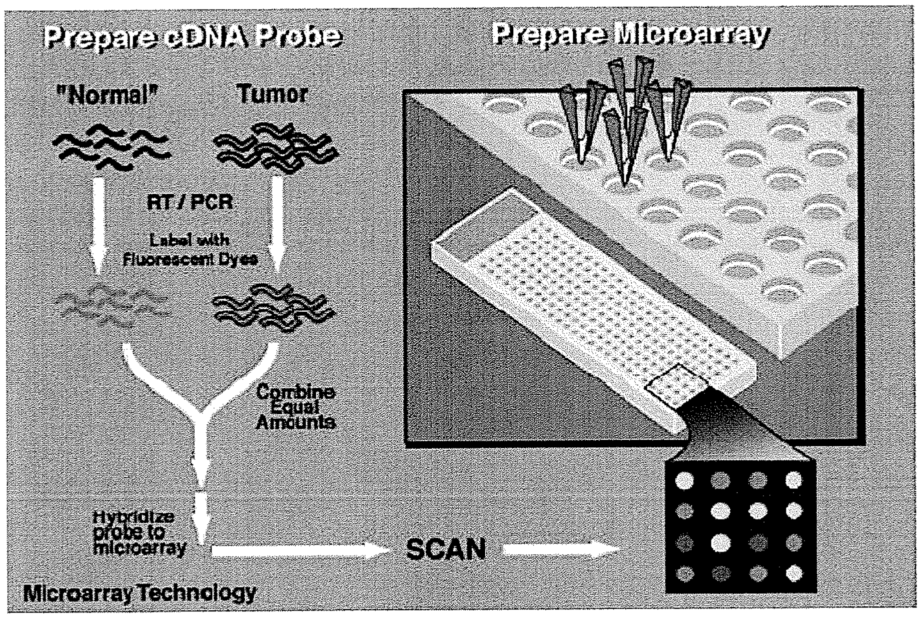
Two broad applications of MR technology are: magnetic resonance imaging (MRI) and magnetic resonance spectroscopy (MRS). Both techniques are based on the same physical principles of MR technology as explained above, i.e., detecting the energy exchanged between external magnetic fields and specific nuclei within atoms. The distinction between both applications is: in MRI, the emitted MR signals correspond to the spatial positions of the observed nuclei and are then translated into an atomic image by assigning different grey values according to the strength of the emitted signals; while in MRS, the emitted signals correspond to the chemical components of a scanned tissue, and are transformed into spectra with peaks that represent the concentration of the chemical components according to the strength of the emitted signal. The spectra generated by MRS can be used to determine the chemical structure of compounds and the concentration of chemical compounds during metabolic processes. In the former application of MRS, the chemical shift (i.e. a field-independent variation of the resonance frequency) of a nucleus can provide a clue on the nature of the chemical bonds surrounding the nucleus. Different chemical shift values in such MRS spectra represent different chemical bonds in the observed compound that contain the nucleus. In using MRS spectra to determine the chemical composition of compounds, the area of each peak value of chemical shift in the spectra is compared with the corresponding chemical shift value for a nucleus in a standard compound and the difference in the area of each peak value corresponds to the abundance of the given nucleus in the observation. Tetramethyl silane ($(CH_3)_4Si$) is a common standard compound used for this purpose. Figure 2.1 depicts an MRS spectrum that reveals the chemical structure of an organic compound. A comprehensive description of MR technology and its application in MRS and MRI is available in [44].



Microarrays is another noninvasive technique that can be used to acquire biomedical data useful for describing phenomena that relate to disease identification and progression, responses to stimuli, underlying differences between cells of different types, and the elucidation of gene function at the molecular level. The nucleus of the biological cell of cellular organisms contains substances such as DNA (Deoxyribonucleic Acid) that encode genetic information which describe the uniqueness of a cell, and RNA (Ribonucleic Acid) that enables the transmission of the genetic information. DNA molecules consist of long sequences of four different deoxyribonucleotides that define the genome (an encoding of the complete genetic information) of a cellular organism. Transcribing a DNA to mRNA (messenger RNA) and subsequently to proteins is referred to as gene expression [31].

All the cells of an organism, with a few specific exceptions, have the same genomic DNA representation; though not all the cells of an organism are the same [31]. The differences in cell types result from the different subset of genes they express. Also, the response of a cell in the form of gene expressions varies with stimuli. Therefore, gene expressions can also be used to determine cell types and to differentiate between normal and abnormal cells by monitoring the expression conditions of the cell in response to stimuli.

A microarray contains a glass or polymer slide onto which DNA molecules are attached at fixed locations. These locations are known as spots or features. An array can contain thousands of spots, each containing millions of identical DNA molecules or fragments. Each DNA molecule or fragment has length ranging from tens to hundreds of nucleotides. In a microarray, each DNA molecule identifies a single mRNA in a genome, and the identification of the mRNA transcript is printed on the microarray by a robot or jet to avoid ambiguity. Applying a laser beam makes the print fluoresce with varying intensity. The intensity at each slot in the microarray reflects the abundance of the DNA expressed from the slot. Figure 2.2 depicts the data acquisition process in microarray technology.



A comprehensive description of microarray technology and its applications is available in [31].

2.2 Processing biomedical datasets

Biomedical datasets acquired using most noninvasive techniques are characterized with high feature space dimensionality and small sample sizes. Typically, for each observable sample, the number of measurable attributes (i.e. features) ranges from thousands to hundreds of thousands. Conversely, the number of observable samples usually ranges between units to hundreds, since the cost (time and resources) of measuring the large number of attributes for each sample is usually huge. However, a collection of the measured attributes of the observable samples can be analysed to derive reliable inferences. A simple analysis of biomedical datasets using literal visual observation is difficult and can generate unreliable inferences. For instance, in the biomedical spectra shown in Figure 2.3, the regions of features (highlighted area) that discriminate between the normal specimen's spectrum and the infected specimen's spectrum are not visually evident.

Pattern recognition techniques can be applied to biomedical datasets to learn the intrinsic relationship within the datasets and to subsequently assign independent samples to distinct classes in order to derive reliable inferences. These inferences can be used for diagnostic and prognostic purposes. A pattern recognition system is also known as a classification process [54]. A pattern consists of an ordered set of observed measurements having an associated meaning. In the design of a classification process for biomedical datasets, each sample in the dataset, which typically consists of an ordered feature vector and an optional class label, corresponds to a pattern. The basic goal in the design of a classification process is to establish a mapping from the feature vector space into the space of class labels and thereby associate a meaning to each feature vector in a given dataset. Classification can be supervised or unsupervised. In supervised classification, the class label space is fully predefined while in unsupervised classification the class label space is defined during the classification process [21]. The scope of this thesis covers only supervised classification; therefore subsequent reference(s) to classification is only applicable to the supervised option.

Besides the feature vector space and the class label space, a classification process consists of basic components that are interconnected. These components are: pre-processing, feature selection and extraction, classifiers design, and optimization [54]. Figure 2.4 shows the connection amongst the basic components of a classification process. I describe these components in the following subsections.

2.2.1 Pre-processing component

The raw datasets that are acquired for classification purposes using most noninvasive techniques usually contain noisy background information that can impair an easy classification of the datasets and a meaningful interpretation of the inferences derived from the dataset. The pre-processing module primarily performs operations that segment the interesting portion of the raw datasets from the background noise and possibly create a compact representation of the raw datasets. Examples of operations that can be performed in this module are: noise filtering, smoothing, and normalization [54]. Some form of optimization may be required in this module to ensure that the input patterns (i.e. feature vectors) are represented using the best form.

2.2.2 Feature selection and extraction component

Most pre-processed real-life datasets of biomedical origin that are used in the design of a classification process are characterized with high feature space dimensionality. Using the high-dimensional datasets directly for classification is disadvantageous in two respects. Firstly, the computational complexity of the classification becomes too large. For instance, a simple linear classifier requires in the order of KL operations, where K is the number of classes and L is the dimensionality of the feature space; and similarly a quadratic classifier requires in the order of KL^2 operations [18]. Secondly, an increasing feature space dimensionality eventually degrades the classification performance when there is no corresponding increase in the sample size. To avert these disadvantages, there is need to reduce the dimensionality of the feature space by selecting a subset that has the highest discriminatory capability (i.e. perform feature selection) or by transforming the feature space into a projected space that eases the classification process (i.e. perform feature extraction).

Feature selection simply selects a subset of the original feature set with an ultimate goal of finding the (optimal) subset that enables the highest classification accuracy or the lowest classification error rate. The optimality of the subset is measured by an evaluation criterion created using the optimisation component, as shown in Figure 2.4 above. Feature selection is often modelled as a combinatorial optimisation problem and several feature selection techniques have been proposed to find the optimal subset in different problem domains. A detailed survey of existing feature selection techniques is presented in the later part of this chapter (section 2.3). Despite the existence of many feature selection techniques, new methods that minimize computational complexity requirements and maximize classification accuracy are still desirable.

Feature extraction methods are usually applied when a transformation of the original feature space into a new feature space can simplify the classification process. Feature extraction methods project the original feature space onto another feature space; that is, new features are extracted as functions of the features in the original feature space. The subsequent classification processes are performed using the new feature space. Based on whether the extraction function is linear or non-linear, there are two broad categories of feature extraction techniques: linear feature extraction techniques and non-linear feature extraction techniques [54]. An example of a linear feature extraction technique is the principal component analysis (PCA) [22] and an example of a non-linear feature extraction technique is the multi-dimensional scaling (MDS) [27]. Feature extraction techniques usually make classification easier when a suitable transformation function is used. However, finding a suitable transformation function for different problem domains is a challenging task.

2.2.3 Classifier design component

The core component of the classification process is the classifier (i.e. the algorithm that actually derives the mapping of a given input data or pattern onto the class label space). A classifier can predict the class label of a given input data by constructing implicit boundaries in the feature space to separate the samples that belong to different classes from one another while optimizing a cost function [10]. The cost function is usually an estimation of the rate of wrong classification. The cost function for the classifier design component is used as an evaluation criterion to tune the performance of the classifier.

The optimization component provides the evaluation criterion for the classifier component (Figure 2.4).

Classifiers can be designed using three different approaches [54]. The first approach is based on the identifiable similarities that can exist between a reference entity (i.e. a prototype) and the other entities to be classified. Template matching [21] is an example of a classifier that can be designed using the concept of similarity. The second approach uses probabilistic methods. Classifiers in this category are based on the Bayes decision rule, the maximum likelihood or density estimators [21]. Examples of classifiers in this category include k-nearest neighbour (KNN) classifiers and Parzen window classifiers [21]. The third approach uses statistical methods to construct a decision boundary directly in the feature space while optimizing the classification error rate. Examples of classifiers in this category are Fisher's linear discriminant analysis (LDA), multilayer perceptrons, and support vector machines (SVM) [21].

The classifier design component in Figure 2.4 can consist of a classifier that is designed using any of the approaches described above, or a hybrid of the approaches. Two or more individual classifiers that are designed using any of the approaches can also be combined in a collaborative manner to address complex classification problems.

2.2.4 Optimization component

The optimization component combines with the other components of a classification process by providing the evaluation criterion that determines the optimality of the results generated by those other components. As shown in Figure 2.4, the optimization component is a sub-component of the preprocessing, feature selection and extraction, and classifiers design components. In the preprocessing component, the optimization portion provides a basis to determine the quality of the feature vectors that represent each sample. In the feature selection and extraction module, the optimization sub-component determines the optimality of the examined subsets during the search for the optimal feature subset or evaluates the effect of transforming a feature space into another feature space on the complexity of the classification process. In the classifiers design component, the optimization sub-component provides an evaluation criterion that guides the classifier towards generating minimum classification error rate.

2.3 Feature selection

Given an original feature space defined by a given feature set, feature selection involves finding the '*optimal*' subset of the feature set that best represents the original feature space. Typically, the optimal subset is sought strategically using techniques (feature selection techniques) that are guided by an evaluation criterion. The following subsections describe existing feature selection techniques and the different evaluation criteria that can be used to guide these techniques.

2.3.1 Feature selection techniques

Decades of active research have been devoted to designing and developing feature selection techniques to address the feature selection problem. Feature selection techniques typically consist of an underlying search or ranking algorithm that explores the feature space and a cost function (e.g. a measure of the classification error rate) that guides the underlying algorithm. Considering the approach for evaluating the cost function of the feature selection techniques, Kohavi and John [26] identify two approaches to designing feature selection techniques: filter and wrapper approaches. Liu and Yi [29] further identify a third approach known as the hybrid approach, which combines the strength of the filter and wrapper approaches. The filter-based approach determines the fitness of an examined feature subset without any reference to or feedback from the target classifier. That is, the cost function evaluation is independent of the target classifier that uses the selected subset of features in the subsequent classification of independent datasets. Rather, a generic error estimation function can be used to compute the cost function value that guides the ranking of the individual features or the search for an '*optimal*' subset in the feature space. Examples of filter-based feature selection techniques are described in [16, 28, 52, 24]. On the other hand, the wrapper-based approach determines the fitness of an examined feature subset by referring the subset to the target classifier to get a feedback in the form of an estimation of the classification error rate that will result when the examined subset underlies the design of the target classifier. Examples of wrapper-based feature selection techniques are described in [8, 26]. The wrapper approach usually enables the selection of feature subsets that leads to higher classification accuracy than the filter approach. However, the computational requirement for evaluating the discriminatory capability (i.e. the cost function value) of each examined subset by the target classifier makes the wrapper-based methods more computationally intensive than the filter-based methods. The filter-based feature selection methods are typically faster than wrapper-based methods. Figure 2.5 depicts the concept of the filter-based and wrapper-based feature selection techniques. The hybrid-based feature selection techniques combine the strengths of the other two approaches. An example of an hybrid-based feature selection technique is described in [9].

In this thesis, I focus on wrapper-based approach for designing feature selection techniques because of the high classification accuracy requirement in the processing of biomedical datasets for diagnostic and prognostic purposes.

Guyon and Elisseeff [15] group feature selection techniques into two broad categories based on the underlying search or ranking algorithm: feature ranking techniques and feature subset selection techniques. Feature ranking techniques order the features in the feature space according to a relevance criterion such as covariance, and select a subset from the ordered features. Kira and Rendell [24] describe a simple feature ranking technique. The technique scores each feature in the original feature set using a ranking criterion and selects the first k features having the highest scores as the '*optimal*' feature subset, where k is the cardinality of the desired subset. A primary drawback of the feature ranking techniques with respect to classifier design is: a combination of the k

highest ranked features is not necessarily the optimal or near-optimal subset of features that can most enhance the classifier's performance.

Given sufficient computational time, feature subset selection techniques such as in [32, 42, 43, 45] implicitly examine all the feature subsets and select the subset having the 'best' cost function evaluation as '*optimal*'. These techniques guarantee finding the optimal feature subset with respect to the estimation of the target classifier performance. However, the computational requirements (time and resources) of the techniques are very intensive and may be impracticable when applied to large-scale feature selection problems. Besides, the techniques are usually based on assumptions that are not always true in practice. For instance, the branch and bound-based feature selection techniques [32, 42, 43] require that the cost function be monotonic on the subset of features; i.e. adding a new feature from the original feature set to a current feature subset must result in a better optimization of the cost function.

To solve the feature selection problem in a practical way, some other feature subset selection techniques intelligently examine some of the possible subset of features and select a subset having the 'best' cost function evaluation amongst all the examined subsets as the '*optimal*' subset. Although these techniques do not guarantee finding the optimal subset, they can find a subset that is almost as qualitative as the optimal subset in terms of the discriminatory capability (i.e. near-optimal subset). These feature subset selection techniques are described in the following.

Sequential forward selection (SFS) and sequential backward selection (SBS) [1] are based on a simple greedy-like deterministic heuristics. SFS starts by selecting an empty subset as the current subset and sequentially adds a new feature (from the original set) to the current subset. In each sequence, the added feature satisfies the condition of combining with the current subset to give the 'best' evaluation of the cost function. The selection process stops when a termination criterion is satisfied (e.g. when the addition of a new feature no longer improves the cost function or when the cardinality of the subset equals a set threshold) and the '*optimal*' subset of the selection is the current subset prior to termination. SBS is similar to SFS, but the selection process is reversed. SBS begins with the entire original set as the current subset and sequentially removes a feature from the current subset until a termination criterion is satisfied. SFS adds a single feature (and SBS removes a single feature) at each search sequence; hence the discriminatory dependencies that exist amongst some combinations of features are ignored during the search. Stearns [45] proposes the plus-*l*-take-away-*r* method to address the shortcoming of possible exclusion. At each sequence, the method adds *l* features to the current selection using SFS and removes *r* features using SBS. The challenging task of this method is: there is presently no theoretical means of choosing a predefined value for *l* and *r* that enables finding the '*optimal*' subset. Generalized sequential forward selection (GSFS), a generic form of SFS, provides a flexible means of finding the '*optimal*' subset by permitting the addition of *k* features to the current selection at every search sequence. Similarly, there are generic forms for SBS and plus-*l*-take-away-*r*: generalized sequential backward selection (GSBS) and generalized plus-*l*-take-away-*r*, respectively.

The aforementioned sequential search techniques do not permit backtracking; that is, a search step cannot be reversed even when subsequent steps reveal the step as impairing to finding the '*optimal*' subset. To resolve the backtracking drawback, Pudil et al. [36] propose the sequential floating forward selection (SFFS) and the sequential floating backward selection (SFBS). At each search sequence, the SFFS method adds to the current selection, using SFS and performs some SBS steps as long as the cost function evaluates to a better value. SFBS is similar to SFFS, but the progressive search sequence is based on SBS. Generally, other than the SFS and SBS, the sequential search techniques are computationally expensive for large-scale feature selection problems.

Siedlecki and Sklansky [38] propose a genetic algorithm (GA) approach to feature subset selection. Nikulin et al [33] develop a GA-based technique for selecting the '*optimal*' subset of block of features (regions) in biomedical MRS spectra. The technique does not generate a stable subset of features, because of the highly probabilistic property of genetic algorithms. Also, biomedical MRS spectra usually have evident correlation amongst consecutive features; therefore, this technique may be inadequate for biomedical data not having such correlation.

Zhang and Sun [53] develop a tabu search method for feature subset selection. Tabu search is a search meta-heuristics that intelligently explores a given solution space beyond local optimality by using an adaptive memory structure and a strategic responsive exploration [13]. The adaptive memory, known as the *tabu list*, keeps track of solutions that have been visited and should be avoided for a number of iterations; and the *tabu tenure* determines how long a solution remains in the *tabu list.* During a tabu search iteration, adjusting or varying a current solution can be used to derive a new set of solutions; a function that maps a current solution onto a set of solutions derived from the current solution is known as a *neighbourhood.* Zhang and Sun [53] performed a comparative analysis of the tabu-search-based technique and other feature selection techniques (SFS, GSFS, SBS, GSBS, plus-l-take-away-r, SFFS, SFBS and GA) using a synthetic dataset. Although the result of the performance analysis shows tabu search as a promising search heuristics for feature selection problem, the analysis is done using a synthetic dataset and there may be a need to verify the claims using real-life datasets. I examine the strength of the basic tabu search feature selection technique using biomedical data and propose a new feature selection technique that can be adapted to solve feature selection problem in most forms of biomedical data (e.g. MRS spectra, microarrays, mass spectra). The proposed technique is based on a multilevel search paradigm [48].

2.3.2 Evaluation criteria for feature selection techniques

To find the optimal subset, an imperative and challenging requirement is: determining an appropriate criterion for evaluating the discriminatory capability of each examined subset during the search process. The cost of misclassification when an examined subset underlies a target classifier is a proper criterion for evaluating the fitness of the examined subset. However, assessing the cost of misclassification is difficult and sometimes fully unknown; therefore, this cost is often replaced with the classification error rate [18]. The classification error rate can be derived mathematically or empirically. The exact mathematical representations of the classification error rate are complex and simple approximations are often used. Two of the common approximate mathematical methods for representing the classification error rate are: the inter-intra class distance and the Chernoff distance [18]. The inter-intra class distance is based on the Euclidean distances that separate the classes from one another and the proximity of the feature vectors of the samples in each class to the mean feature vector of the class. The inter-intra class distance can be expressed as a ratio of the between class separability to the within-class separability. The Chernoff distance (and other similar distance measures such as Bhattacharyya distance) is based on the probability densities of the classes. While the Chernoff distance and Bhattacharyya distance are mainly suitable for two-class classification problems, the inter-intra class distance can be used as an evaluation criterion for multi-class classification problems [18].

Empirical methods can also be used to estimate the classification error rate in order to evaluate the examined feature subsets. The usual practice is to partition the entire dataset into a training set and a test set (also known as a validation set or evaluation set). An examined feature subset f defines a feature space wherein a classifier is designed and validated. The classifier is designed using the training set and is subsequently validated by classifying the samples in the test set; and the estimated classification error rate $\boldsymbol{e}$ is expressed as the ratio of the number of the wrongly classified samples during validation n_e to the sample size of the test set N.

$$\boldsymbol{e}(\boldsymbol{f}) = n_e/N$$

For most biomedical datasets, the sample size of the available dataset is usually small and using the same dataset as the training set as well as the test set can be considered a viable option. However, this option poses the threat of overfitting the designed classifier to the training set and the resulting classifier can perform poorly when subsequently used to classify an independent test set. The possibility of overfitting can be averted when evaluating the fitness of the examined feature subsets from datasets with small sample size by using the cross-validation method and the leave-one-out method [46]. For the cross-validation method, the available dataset **D** is randomly partitioned into

x equally sized subsets of samples (i.e. $\mathbf{D} = \{D_1, D_2, D_3, \ldots D_x\}$). One of the subsets e.g. D_1 is withheld and the remaining x-1 subsets are used to train the classifier. The withheld subset is subsequently used to validate the classifier and an estimated error rate over the withheld subset $\boldsymbol{e}_{D1}$ is computed. Similarly, the estimated error rate $\boldsymbol{e}_{Di}$ over each subset D_i in $\mathbf{D}$ is computed and the average of the estimated error rates can be used as an evaluation criterion for the examined feature subsets. The leave-one-out method is a particular case of the cross-validation method wherein the partitioning of the available dataset is done such that the cardinality of the subset of samples is 1 (i.e. $|D_i| = 1$). Although the leave-one-out method is more computationally intensive than the cross-validation method, the leave-one-out method is often used in practice since the bias in the evaluation method is less for this method and even negligible when the sample size is sufficiently large.

Chapter 3

Search Heuristics

Search heuristics are algorithms that explore a given solution space using strategies that are primarily based on intuitive intelligence. A category of search heuristics, local search and its improved versions such as tabu search, explores a given solution space using strategies that are based on a neighbourhood function paradigm [35]. The formal proves of the search behaviours of these methods are presently not in existence, rather the behaviours are described using empirical analysis.

In this chapter, I describe the underlying search heuristics (i.e. multilevel search) for the newly presented feature selection technique. Multilevel search creates an intelligent search framework for solving optimization problems by combining the multilevel paradigm with other search methods such as local search, tabu search, and genetic algorithm. The following sections describe the tabu search (i.e. the search method used in the multilevel search presented in this research) and the multilevel paradigm. The choice of tabu search as the underlying search method for the multilevel feature selection technique is attributable to the comparative analysis in [53] that shows tabu search as a promising tool for solving feature selection problem.

3.1 Tabu search

Most real-life optimization problems (that are common in applied science, business and engineering) are difficult to solve and stiff challenges are often encountered when classical methods are used to solve these problems. The innovation of meta-heuristics such as tabu search is changing the approach to solve these problems. Tabu search is a meta-heuristic that guides a local heuristics search method to explore a given solution space beyond local optimality using an adaptive memory structure and a responsive exploration strategy [13].

The local heuristics search component of tabu search is similar to the generic local search method as described in [**35**]. The local heuristics search explores a solution space by *moving* from a given solution to another solution in the *neighbourhood* of the given solution according to some defined rules. Consider a combinatorial optimization problem that involves minimizing a cost function $Y(x)$ over a finite set of solutions X. For each solution $x \in X$, there is an associated subset $N(x) \subseteq X$ such that the elements of $N(x)$ can be derived by performing a *move* operation on x. $N(x)$ is known as the neighbourhood of x and a *move* is a simple operation that can transform x into any of the solution in its neighbourhood. To find a local optimal solution, the local heuristics search starts from an initial solution x_0 as the current solution and at each iterative step i, a new solution is chosen in the neighbourhood $N(x_i)$ of the current solution x_i. The choice of a new solution is usually guided by a selection strategy and the commonest is known as *steepest descent* (i.e. the 'best' solution in the neighbourhood of the current solution becomes the next current solution – the next current solution x_{i+1} satisfies $Y(x_{i+1}) \leq Y(x)\ \forall\ x \in N(x_i)$). The iteration continues until a termination criterion is satisfied and the current solution after the termination criterion is satisfied is the local optimal solution. For instance, the termination criterion can be when all the solutions in the neighbourhood of a current solution are worse than the current solution.

In most practical optimization problems, the trajectory of the cost function over the set of solutions usually consists of several local optimums (Figure 3.1). Using the basic local heuristics search to find the global optimum for such problems can be misleading since the local search can be easily trapped in a local optimum. As shown in Figure 3.2, if the search starts at point x'_0, the local optimum $x'$is presented as the optimum solution instead of x that cannot be reached if the search starts at x'_0. Tabu search guides the local heuristic search beyond local optimality by maintaining a selective history of the encountered solutions during the search process and using the same to modify the neighborhood of the current solution $N(x)$.

The encountered solutions can be tracked using a short term memory structure and/or a long term memory structure. For the short term tabu search, the modified neighborhood $N'(x)$ of a current solution is a subset of the original neighborhood $N(x)$. Consider a list $\boldsymbol{T}$ (tabu list) that contains n different solutions, that is $\boldsymbol{T} = \{x_1, x_2, \ldots, x_n\}$, if $\boldsymbol{T}$ explicitly contains solutions that have been encountered during the search process, then the relationship amongst the original neighborhood $N(x)$ and the modified neighborhood $N'(x)$ of a current solution and the tabu list $\boldsymbol{T}$ is as follows:

$$N'(x) = N(x)\backslash \boldsymbol{T}$$

That is, $N'(x)$ consists of solutions of $N(x)$ that are not elements of the tabu list. The solutions on the tabu list $\boldsymbol{T}$ are usually labeled as *tabu-active* and the duration of a solution on the tabu list is known as *tabu tenure t;* that is, a solution can remain tabu-active for a period of t iterations other than when the solution satisfies an aspiration criterion (i.e. rules that are designed to override the tabu list membership rule in order to enable tabu search achieve the best performance). For the long term tabu search, the modified neighborhood $N'(x)$ can contain solutions that are ordinarily not included in the original neighborhood $N(x)$. Tabu strategies such as intensification and diversification benefit from the long term memory structure. Intensification involves modifying the choice rule to favor moves or combination of solutions that are historically found good. Diversification drives the search process towards examining unvisited regions of the search space in order to generate solutions that differ from the encountered solutions. I refer the reader to [13] for a comprehensive description of the concepts of tabu search.

Tabu search has been successfully applied to solve several difficult real-life optimization problems such as scheduling, location and allocation, logic and artificial intelligence, telecommunications, routing, graph optimization, and general combinatorial optimization. A comprehensive list and explanation of some of the applications of tabu search to solve different optimization problems is available in [14, 17]. Recently, Zhang and Sun [53] designed a tabu search to solve feature selection problem and compared its performance with other feature selection techniques. The pseudocode of the tabu search feature selection techniques is as follows:

```
Start
Define variables
    $F$ = Solution space
    $f$ = initial solution
    $f$ = 'optimal' solution
    $N(f)$ = the neighbourhood of solution $f$ and $N(f) \subset F$
    $k$ = iteration counter
    $TL$ = tabu list
    $J(f)$ = cost function that evaluates the discriminatory capability of solution $f$

Initialize variables
    Generate a starting solution $f$
    Set the optimal solution as equal to initial solution i.e. $f = f$
    Set iteration counter as 1 i.e. $k = 1$
    Set tabu list as a null set; $TL = \Phi$

Begin iteration
    Generate a non-null set of neighbourhood solutions of $f'$ $N(f') = \{y_1, y_2, \ldots\}$
    // Neighbourhood search process
    For each element of $N(f')$
        compute the cost function value
        compare the values and find the 'best' element in $N(f')$
    End for
    Set the 'best' element as $y'$ i.e. $y' \in N(f')$ such that $J(y') \leq J(y)\ \forall\ y \in N(f')$).
    Check if $y'$ is tabu-active (i.e. if $y' \in TL$) and if $y'$ does not satisfy the aspiration criterion
    if yes,
        remove $y'$ from $N(f')$ i.e. $N(f') = N(f') - \{y\}$
        search for another solution in $N(f')$ next in optimality to current $y'$ and repeat the check
    if no,
        Set $y'$ as current solution i.e. $f = y'$
        if $y'$ is better that $f$ i.e. $J(y') < J(f')$
            set $y'$ as the current optimal solution i.e. $f = y'$

    Check if termination condition is satisfied
    if no,
        append the current solution to tabu list i.e. $TL = TL \cup \{f'\}$
        increment counter $k = k + 1$
        Iteration continues
    if yes,
    output $f$ as optimal solution
Stop
```

The comparative analysis in [53] shows tabu search as a promising tool for solving feature selection problem.

3.2 Multilevel paradigm

Multilevel paradigm or method (also known as multigrid) is an approach to solving computational problem that was originally proposed in the field of numerical approximation [5, 51]. The primary idea of this method is to strategically and gradually reduce the size of a computational problem instance such that the solution of a reduced form of the problem can be extended to derive a solution for the original problem instance. Typically, to apply this approach, an original problem domain discretization is coarsened recursively to generate coarser discretizations (levels) by increasing the grid spacing at a given discretization with respect to the spacing at the next less coarse discretization. Subsequently, a solution at a coarse discretization is improved upon (i.e. refined) in the less coarse discretization until a final solution is derived for the original discretization. The process of improving a solution along the coarse discretizations is based on two primary ideas: *nested iteration* and *coarse grid correction* [6]. The nested iteration involves a simple process: compute an approximate solution at a given coarse discretization, interpolate and use the approximation as the initial guess for an approximation at the next less coarse discretization. The process starts at the coarsest discretization and continues across the levels with decreasing coarseness. The final approximation is used as an improved initial guess for the fine-grain relaxation (i.e. the computation of a final approximate solution at the original problem domain discretization). The nested iteration scheme helps to improve convergence in the original problem domain discretization by providing a good initial guess to the relaxation method. In some domain problem discretization, the final approximate solution produced by the nested iteration still has significant errors; the application of the coarse grid correction is very useful in such case. In the coarse grid correction, an approximation is first computed in the original discretization. The residual of the approximation is then projected onto the next coarser discretization where an approximation of the associated error is computed by solving for the error in the system of linear equations that relates the error with the residual. The correction at the coarse discretization is interpolated to the fine discretization and a new approximation is computed. The *V-cycle scheme* of multilevel methods is based on the recursive application of the *coarse grid correction.*

The combination of the multilevel approach and search algorithms, often referred to as *multilevel search,* is an adaptation of the multilevel paradigm to combinatorial optimization problems. The multilevel search framework primarily consists of three phases: *coarsening*, *initial search*, and *refinement*. Most often, the three phases occur consecutively in the order stated above. The following describes the multilevel search in relation to these phases. Consider a combinatorial optimization problem $\boldsymbol{A}$ and an original problem instance of $\boldsymbol{A}$ denoted as A_0. As shown in Figure 3.3, during the coarsening phase, a succession $A_0, A_1, A_2, \ldots, A_{i-1}, A_i$ of increasingly smaller (or coarser) problem instances of $\boldsymbol{A}$ is generated by recursively reducing the number of decision variables in accordance with the definition of the original problem instance.

During the initial search phase, which usually follows the coarsening phase, a feasible solution s_i of the smallest problem instance A_i is computed. During the refinement phase, the feasible solution s_i of the coarsest problem instance A_i is interpolated onto and improved upon at the next less coarse problem instance A_{i-1}. The refinement phase continues the interpolation and refinement of the solution from a coarse problem instance to a less coarse problem instance until the values of the decision variables of s_0 (the desired solution in the original problem instance A_0) can be derived by interpolating a solution of the immediate less coarse problem instance A_1. We refer the reader to [23, 48, 49] for recent applications of multilevel search to different combinatorial optimization problems.

The design and implementation of the coarsening, search, and refinement phases of multilevel search algorithms vary with different combinatorial optimization problems. There are no generic designs and implementations of the multilevel algorithm phases; therefore, the need to specify the design and implementation of these phases for the proposed multilevel feature selection is imperative. The description of the design, implementation, and calibration of these phases is presented in the next chapter.

Chapter 4

Multilevel Search Algorithm for Feature Selection Problem

This chapter describes the design of a search framework for the feature selection problem using the multilevel search method. Particularly, I present the design requirements and the calibration parameters of the multilevel search feature selection technique.

Consider a problem instance of the formulation (1) in section 1.2 above and a biomedical dataset **D** = $\{S_1, S_2, S_3, \ldots S_k\}$ consisting of k samples. Each sample is represented as a feature vector such that the dimensions of the vector correspond to the features in the original feature set **F** = $\{f_1, f_2, f_3, \ldots f_L\}$provided in the given dataset. For each sample $S_i = (s_1, s_2, s_3, \ldots s_L)$ represented as a feature vector, the elements $s_1, s_2, s_3, \ldots s_L$ are the coefficients of the dimensions of the feature vector and the values of these coefficients are provided in the given dataset. The feature selection problem instance that is considered in this context involves finding the '*optimal*' feature subset f from **F** such that the cardinality of f is predefined as m.

For programming purposes, the elements of **F** are represented using unique positive integers that correspond to the position of each feature in the feature vectors provided in the given dataset. f being a subset of **F** is also represented as a set of unique positive integers such that each element represents a feature that is selected from **F** as a member of the optimal subset. For instance if $L = 10$ and $m = 3$, the possible elements of **F** and f can be represented as follows: **F** = $\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ and $f = \{3, 8, 10\}$. In this example, the features in each given feature vector are uniquely represented using the positive integer position 1, 2, 3, …, 10 and the elements of *f*, i.e. 3, 8, 10 imply that the third, eighth and tenth features in each feature vector constitute the optimal subset. This representation is appropriate since the feature vectors in a given biomedical dataset are ordered sets of feature values and the positions of the features are the same in each feature vector. In a given feature space, the decision to select a feature to be a member of a subset of the feature space is represented using an implicit Boolean variable *x*. Each element f_i of **F** has a corresponding Boolean variable x_i, such that when the value of x_i is set to '1' during a selection process, the corresponding integer value f_i in **F** is included in the desired subset; otherwise when set to '0', the corresponding integer value in **F** is not included in the desired subset. The value of the decision variables are set by a decision process and the underlying search metaheuristics (i.e. tabu search) performs the decision process in the multilevel feature selection technique.

The first step in the design of the multilevel feature selection technique involves identifying the coarsening strategy that can be used to recursively reduce the number of features for the original problem instance in context. The second step involves identifying an appropriate search method that finds a starting solution from the coarsest form of the problem instance. The third step is to refine the starting solution across the reduced problem instances from the coarsest to the least coarse instance. Figure 4.1 illustrates an overview of the multilevel feature selection technique.

As shown in Figure 4.1, the coarsening phase recursively generates coarser feature subspaces using a parameter (*reduction factor*) that determines the coarseness of the subspaces and a coarsening algorithm that determines the elements of the subspaces. The search phase generates a solution of the coarsest feature subspace and the solution is used to initialize the refinement phase of the technique. The refinement phase improves upon the solution generated by the search phase across the subspaces with decreasing coarseness. The following sections describe the details of these phases.

The multilevel search is a meta-heuristic method, therefore the calibration of parameters such as the *reduction factor*, the number of levels in the hierarchy, and the appropriate search method for the search and refinement phases, that define the design configurations of the multilevel feature selection technique can be determined using experimental results. I performed experiments for the different design phases of the multilevel feature selection technique. In each experiment, I used a 2-class MRS dataset of biomedical origin from the National Research Council's Institute for Biodiagnostics (NRC-IBD). The dataset consists of 337 samples and a feature space dimensionality of 1500; and the cardinality of the desired '*optimal subset*' is 10. For each complete experiment run, the dataset is randomly partitioned into training and test sets in the ratio 2:1 respectively. The experiments performed in the design phases use only the training set partitions and these are represented as *training dataset instances* in the tables and charts presented in the following sections.

4.1 Coarsening phase

The coarsening phase recursively creates a hierarchy of reduced form of the original problem instance. For the feature selection problem, this phase recursively generates a hierarchy of coarse feature subspaces. Across the hierarchy, a coarse feature subspace consists of features obtained from the immediate less coarse subspace and the dimensionality of the subspaces reduces with increasing coarseness. Given an original feature set $\mathbf{F_0}$, the coarsening phase combines a coarsening strategy with parameters such as the *reduction factor* r and the number of levels n to generate feature subspaces $\mathbf{F_1}$, $\mathbf{F_2}$, …, $\mathbf{F_{n-1}}$, $\mathbf{F_n}$ such that $|\mathbf{F_n}| < |\mathbf{F_{n-1}}| < \ldots < |\mathbf{F_2}| < |\mathbf{F_1}| < |\mathbf{F_0}|$, where n is an implicitly or explicitly defined parameter that determines the number of levels in the hierarchy; and r, the *reduction factor*, is the ratio of the dimensionality of a given feature subspace to the dimensionality of the immediate coarser feature subspace i.e. $r = |\mathbf{F}_i| / |\mathbf{F}_{i+1}|$.

The coarsening phase can reduce the dimensionality of the feature subspaces using different coarsening approaches. We identify two coarsening approaches for the feature selection problem: *feature clustering* and *feature pre-setting*. The first approach is similar to the coarsening strategy applied to the graph-partitioning problem [19]. It involves merging a collection of decision variables together and then representing the merged variables by a single variable that results from the mergence. This idea can be used to combine features and generate feature subspaces as follows: for a given level i, the feature subspace $\mathbf{F}_i$ can be coarsened by aggregating groups of features in $\mathbf{F}_i$ such that an approximated form of each group represents a new feature in the immediate next coarse feature subspace $\mathbf{F}_{i+1}$. For a given subspace, the groups of features that are approximated to constitute the immediate coarser subspace can be created using feature clustering algorithms that identify the possible correlations that may exist amongst the features in the given subspace. For instance, for biomedical MRS spectra wherein evident correlations exist amongst adjacent features, the groups can be created by selecting consecutive features within predefined window(s) and a statistics (e.g. median, average) of the features within the window can represent an approximation of each group. Typically, the coarsened feature subspaces generated using the clustering approach are synthetic, i.e. they consist of features that literally may not exist in the original feature space. Besides, the characteristics of the features can vary for each subspace in the multilevel hierarchy. Therefore, the task of relating the solutions generated from the synthetic feature subspaces to the desired solution in the original problem instance and to the subsequent interpretation of the desired solution can be quite challenging. This challenge may not be prominent for biomedical datasets wherein evident correlation exists amongst adjacent features, since the resulting clusters in the synthetic subspaces do not necessarily compromise the meaning of the original features that constitute the clusters. However, for datasets wherein such correlation may not exist, the clusters are usually not an approximate representation of the constituent original features with respect to meaningful interpretation; therefore, the final near-optimal solution selected for such dataset can impair diagnosis and prognosis. A means of addressing the challenge of retaining the originality of features in the feature clustering coarsening approach requires tracking the features in a subspace that are combined to form new features in the coarser subspaces.

The second coarsening approach (*feature pre-setting*) generates a coarse feature subspace $\mathbf{F}_{i+1}$ at level $i+1$ from the immediate less coarse subspace $\mathbf{F}_i$ by excluding some features from the feature subspace at level i. Once a feature is excluded at a given level, the feature cannot be included in the solution space at the coarser levels. Selecting the elements of the feature spaces at different levels in this manner recursively reduces the dimensionality of the feature subspaces in the multilevel hierarchy and therefore reduces the size of the solution space of the original optimization problem progressively. In the present implementation, the hierarchical framework is encoded by maintaining an n x m dimensional array such that m defines the number of levels in the hierarchy and n defines the varying dimensionality of the feature spaces at each level in the hierarchy.

In order to create the coarse feature subspaces using the pre-setting approach, there is need for a means of determining which features belong to each level. I identify and investigate two strategies for this purpose: *biased feature pre-setting* and *random feature pre-setting*. For a given feature subspace, the biased pre-setting strategy determines the feature that belongs to the immediate coarser subspace by examining the discriminatory capability of the features in the given subspace. The discriminatory capability of the features can be determined by applying a feature selection technique to explore the given feature subspace. Any appropriate feature selection technique can be used for this purpose; a simple feature ranking technique is used in the present implementation. For a given feature subspace, the ranking technique sorts the features in descending order of discriminatory capability and the first *k* features are selected, where *k* is the cardinality of the next coarser subspace. In the random feature pre-setting strategy, the features that belong to a coarser subspace that is next to a given subspace in the hierarchy are selected randomly and recursively. A simple Gaussian random number generator is used to guide the selection in the present implementation. I performed experiments to investigate the effect of the two coarsening strategies on the performance of the multilevel feature selection technique.

In the experiment, I use three instances of the multilevel algorithm that have the same configurations except for the coarsening strategy to find the near-optimal subset for 10 *training dataset instances*. The coarsening phase of the multilevel algorithm instances differs only in the coarsening strategy that underlies the algorithm; the first instance is based on the feature clustering strategy, the second and third instances are based on the two versions of the feature pre-setting strategy (i.e. random feature pre-setting and biased feature pre-setting, respectively). For the three multilevel algorithm instances, I compare the estimated values of the classification error rate - CER derived using leave-one-out cross validation. Table 4.1 (a, b, and c) and Figure 4.2 show the results of the experiment.

Table 4.1a The classification error rate values for the multilevel feature selection technique with a feature clustering coarsening strategy

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0591133	0.0394089	0.0738916	0.0738916	0.0295567	0.0689655	0.0492611	0.0591133	0.08867	0.0689655
2	0.0492611	0.0492611	0.0738916	0.0788177	0.0394089	0.044335	0.0541872	0.0492611	0.0591133	0.0788177
3	0.0541872	0.0492611	0.0640394	0.0738916	0.0394089	0.0591133	0.044335	0.0541872	0.0541872	0.0689655
4	0.0640394	0.0591133	0.0640394	0.08867	0.0394089	0.0394089	0.0492611	0.0738916	0.0640394	0.0837438
5	0.0591133	0.0591133	0.0541872	0.0935961	0.0394089	0.0541872	0.0492611	0.0591133	0.0492611	0.0689655
Average	**0.05714**	**0.05123**	**0.06601**	**0.08177**	**0.03744**	**0.0532**	**0.04926**	**0.05911**	**0.06305**	**0.07389**

Average objective function value = **0.05921**　Standard Deviation = ± **0.01275**

Table 4.1b The classification error rate values for the multilevel feature selection technique with a random feature pre-setting coarsening strategy

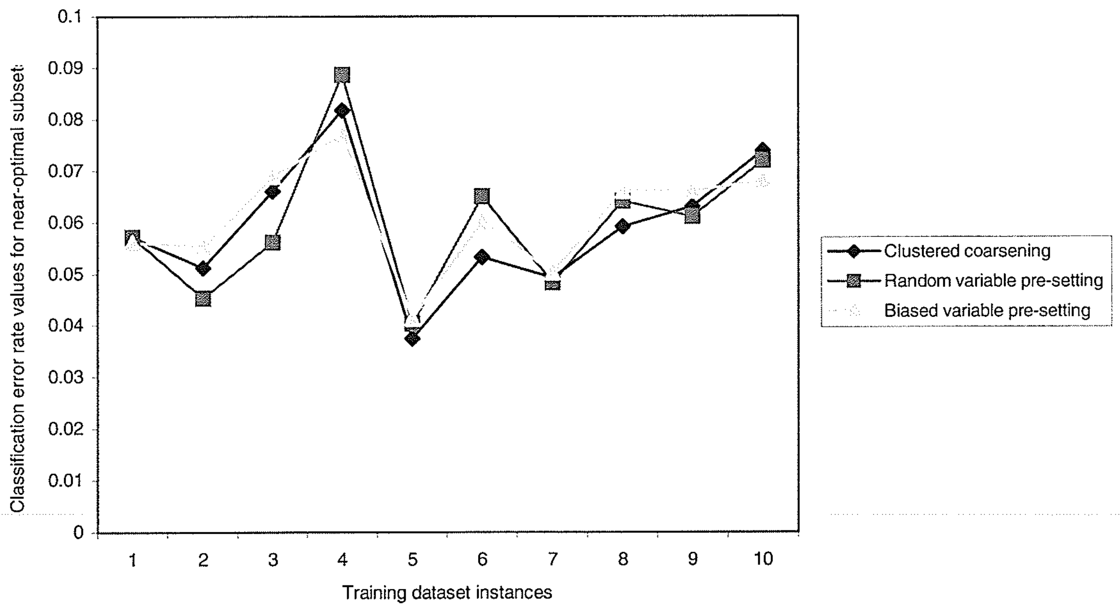
Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0541872	0.0492611	0.0689655	0.0738916	0.0344828	0.0689655	0.0492611	0.0492611	0.044335	0.0837438
2	0.0591133	0.0394089	0.0344828	0.0837438	0.0344828	0.0591133	0.0394089	0.0640394	0.0541872	0.0738916
3	0.0492611	0.0394089	0.0689655	0.0935961	0.044335	0.0591133	0.0541872	0.0689655	0.0689655	0.0738916
4	0.0738916	0.0492611	0.0541872	0.0985222	0.0394089	0.0738916	0.0541872	0.0738916	0.0591133	0.0640394
5	0.0492611	0.0492611	0.0541872	0.0935961	0.0492611	0.0640394	0.044335	0.0640394	0.0788177	0.0640394
Average	**0.05714**	**0.04532**	**0.05616**	**0.08867**	**0.04039**	**0.06502**	**0.04828**	**0.06404**	**0.06108**	**0.07192**

Average objective function value = **0.05980** Standard Deviation = ± **0.01401**

Table 4.1c The classification error rate values for the multilevel feature selection technique with a biased variable pre-setting coarsening strategy

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0591133	0.0541872	0.0640394	0.0837438	0.0541872	0.0640394	0.0689655	0.0837438	0.0689655	0.0640394
2	0.0541872	0.0640394	0.0837438	0.0640394	0.044335	0.0640394	0.0295567	0.0541872	0.0689655	0.0837438
3	0.0591133	0.044335	0.0541872	0.0788177	0.044335	0.0541872	0.0541872	0.0738916	0.0591133	0.0689655
4	0.0541872	0.0689655	0.0689655	0.0738916	0.0295567	0.0591133	0.0591133	0.0591133	0.0738916	0.0640394
5	0.0541872	0.044335	0.0738916	0.0837438	0.0344828	0.0591133	0.0394089	0.0591133	0.0591133	0.0591133
Average	**0.05616**	**0.05517**	**0.06897**	**0.07685**	**0.04138**	**0.0601**	**0.05025**	**0.06601**	**0.06601**	**0.06798**

Average objective function value = **0.06089** Standard Deviation = ± **0.01039**



As shown in Figure 4.2, varying the coarsening strategies using the aforementioned strategies does not have obvious influences on the overall performance of the multilevel feature selection technique. However, Table 4.1(a, b and c) show that the average estimated classification error rate (0.05921) and the standard deviation (± 0.01275) over the 10 dataset instances are least for the feature clustering strategy. Also, the random pre-setting strategy has a lower average estimated error rate (0.05980) but a higher standard deviation (± 0.01401) than the biased pre-setting strategy (0.06089 ± 0.01039). The clustering strategy and the biased pre-setting strategy require more computational resource than the random pre-setting strategy. The clustering strategy requires additional computing resource to track the features that are combined to constitute a coarser subspace at each level; and the biased pre-setting strategy requires additional computing resource to determine the features that are selected from a coarse subspace to a coarser subspace. Therefore, the computational cost of the multilevel feature selection technique is higher when based on the clustering coarsening or the biased pre-setting approach than when based on the random pre-setting coarsening approach. I use the random pre-setting coarsening strategy in present implementations of the multilevel feature selection technique, since this strategy requires the least computation cost and its influence on the multilevel feature selection technique is highly competitive when compared with the other strategies.

Besides the coarsening strategy, another important parameter in the coarsening phase is the *reduction factor* (i.e. $r = |\mathbf{F}_i| / |\mathbf{F}_{i+1}|$). The reduction factor can be predefined explicitly as a constant value when the coarsening is done using feature pre-setting strategy. The reduction factor can also be defined implicitly as the average window size when the coarsening is done using a clustering strategy. Presently, the appropriate value of the reduction factor for a given problem instance cannot be determined theoretically. I perform experiments to determine an appropriate value for the reduction factor for a given problem instance. In the experiment, I use similar configurations (except for the value of the reduction factor that varies with each instance) of the multilevel feature selection algorithm to find the near-optimal subset for the same biomedical datasets. The dimensionality of the original feature space of the biomedical datasets is 1500 and the number of levels in the multilevel algorithm instances is set to 3. I compare the estimated values of the classification error rate – CER using leave-one-out cross validation for the multilevel algorithm instances with the reduction factor value set to 2, 3, 4, 5, and 6. Table 4.2a, b, c and Figure 4.3 show the results of the experiment. In Figure 4.3, the objective function value for the near-optimal subset selected using the different multilevel algorithm instances is plotted against 10 training set instances.

Table 4.2a classification error rate values for the multilevel feature selection technique with a reduction factor of 2

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0591133	0.0689655	0.0689655	0.0492611	0.0492611	0.0394089	0.0541872	0.1034483	0.0837438	0.0640394
2	0.0541872	0.0541872	0.0394089	0.0591133	0.0541872	0.0394089	0.0689655	0.0935961	0.0689655	0.0591133
3	0.0738916	0.0591133	0.0344828	0.0541872	0.0344828	0.0394089	0.0738916	0.0689655	0.0837438	0.0580186
4	0.0541872	0.0541872	0.0541872	0.0591133	0.0541872	0.0492611	0.0591133	0.0738916	0.0738916	0.0591133
5	0.0640394	0.0492611	0.0640394	0.0591133	0.0394089	0.044335	0.0591133	0.1034483	0.0689655	0.0613027
Average	**0.0610837**	**0.0571429**	**0.0522168**	**0.0561576**	**0.0463054**	**0.0423645**	**0.0630542**	**0.08867**	**0.0758621**	**0.0603175**

Average objective function value = **0.06032** Standard Deviation = ± **0.01360**

Table 4.2b The classification error rate values for the multilevel feature selection technique with a reduction factor of 3

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0541872	0.0541872	0.0541872	0.0591133	0.0591133	0.0344828	0.0640394	0.08867	0.0788177	0.0607553
2	0.0492611	0.0591133	0.0344828	0.044335	0.0738916	0.0394089	0.0591133	0.0689655	0.0640394	0.0547345
3	0.0541872	0.0640394	0.044335	0.0640394	0.0344828	0.0344828	0.0492611	0.0788177	0.0738916	0.0552819
4	0.0492611	0.0640394	0.0394089	0.0492611	0.0394089	0.0492611	0.0689655	0.0738916	0.08867	0.0580186
5	0.0689655	0.0640394	0.0591133	0.0689655	0.0344828	0.044335	0.0689655	0.08867	0.0640394	0.0623974
Average	**0.0551724**	**0.0610837**	**0.0463054**	**0.0571429**	**0.0482759**	**0.0403941**	**0.062069**	**0.079803**	**0.0738916**	**0.0582375**

Average objective function value = **0.05824** Standard Deviation = ± **0.01203**

Table 4.2c The classification error rate values for the multilevel feature selection technique with a reduction factor of 4

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0640394	0.08867	0.0738916	0.0640394	0.0541872	0.0591133	0.0837438	0.0935961	0.08867	0.074439
2	0.0640394	0.0738916	0.0689655	0.0689655	0.0640394	0.044335	0.0738916	0.0935961	0.0738916	0.0695129
3	0.0689655	0.0738916	0.0640394	0.0640394	0.0738916	0.0492611	0.0689655	0.0788177	0.08867	0.0700602
4	0.0689655	0.0738916	0.0591133	0.0788177	0.044335	0.044335	0.0689655	0.0837438	0.0935961	0.0684182
5	0.0689655	0.0788177	0.0591133	0.0738916	0.0541872	0.0541872	0.0788177	0.0985222	0.1133005	0.0755337
Average	**0.0669951**	**0.0778325**	**0.0650246**	**0.0699507**	**0.0581281**	**0.0502463**	**0.0748768**	**0.0896552**	**0.0916256**	**0.0715928**

Average objective function value = **0.07159** Standard Deviation = ± **0.01283**

Table 4.2d The classification error rate values for the multilevel feature selection technique with a reduction factor of 5

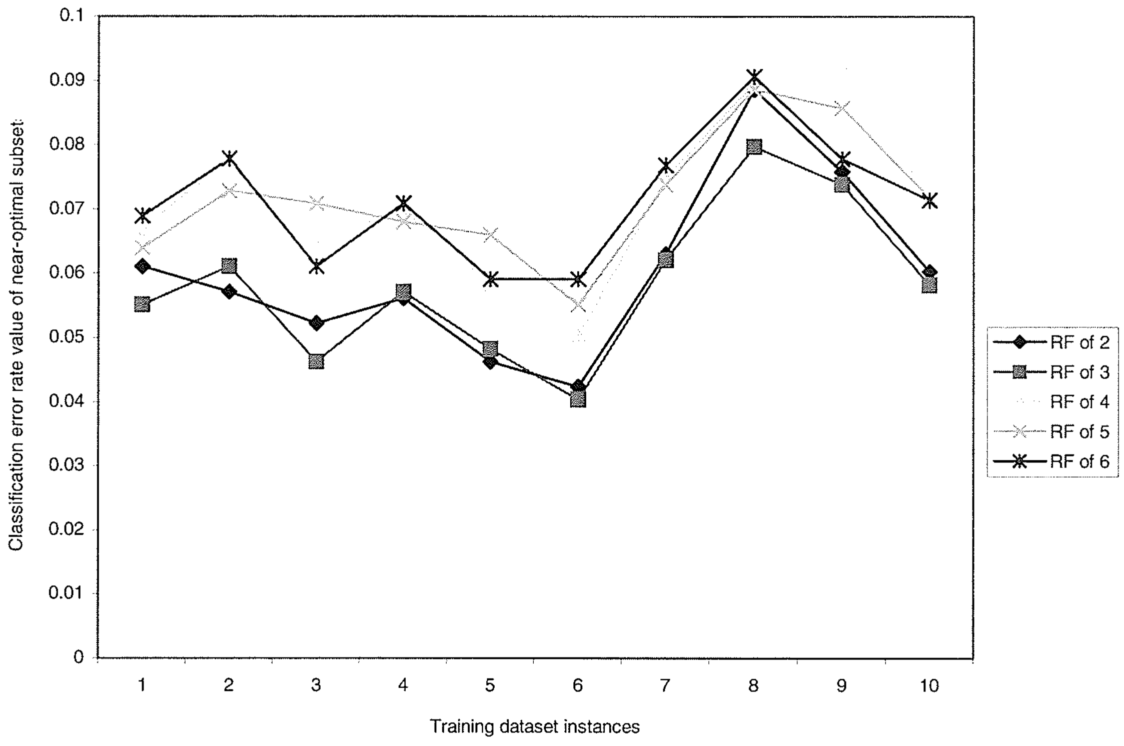
Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0640394	0.0738916	0.0837438	0.0788177	0.0541872	0.0492611	0.0591133	0.08867	0.0788177	0.0700602
2	0.0640394	0.0788177	0.0689655	0.0689655	0.0738916	0.0541872	0.0788177	0.0985222	0.08867	0.0749863
3	0.0591133	0.0640394	0.0738916	0.0689655	0.0788177	0.0591133	0.0788177	0.0935961	0.0738916	0.0722496
4	0.0689655	0.0689655	0.0640394	0.0591133	0.0541872	0.0541872	0.0689655	0.0837438	0.0837438	0.0673235
5	0.0640394	0.0788177	0.0640394	0.0640394	0.0689655	0.0591133	0.0837438	0.0788177	0.1034483	0.0738916
Average	**0.0640394**	**0.0729064**	**0.070936**	**0.0679803**	**0.0660099**	**0.0551724**	**0.0738916**	**0.0886699**	**0.0857143**	**0.0717022**

Average objective function value = **0.07170** Standard Deviation = ± **0.00983**

Table 4.2e The classification error rate values for the multilevel feature selection technique with a reduction factor of 6

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0689655	0.0738916	0.0394089	0.0788177	0.0591133	0.0591133	0.0788177	0.0788177	0.0788177	0.0684182
2	0.0689655	0.0689655	0.0640394	0.08867	0.0591133	0.0689655	0.0788177	0.0985222	0.0837438	0.0755337
3	0.0640394	0.0738916	0.0689655	0.0541872	0.0492611	0.0640394	0.0788177	0.0985222	0.0788177	0.0700602
4	0.0689655	0.0837438	0.0640394	0.0591133	0.044335	0.044335	0.0640394	0.0837438	0.0689655	0.0645868
5	0.0738916	0.08867	0.0689655	0.0738916	0.0837438	0.0591133	0.0837438	0.0935961	0.0788177	0.0782704
Average	**0.0689655**	**0.0778325**	**0.0610837**	**0.070936**	**0.0591133**	**0.0591133**	**0.0768473**	**0.0906404**	**0.0778325**	**0.0713738**

Average objective function value = **0.07137** Standard Deviation = ± **0.00998**



As shown in Figure 4.3, with reduction factor values of 2 and 3, the multilevel feature selection algorithm consistently finds near-optimal subsets having lower average error rate than for the other reduction factor values (i.e. 4, 5 and 6). Using the empirical results shown above, a reduction factor that coarsens a subspace by 30–50% can be recommended for similar problem domain instances. This recommendation agrees with the reduction factor of 2 that is used in most configurations of the multilevel search algorithm for solving the graph partitioning problem. A reduction factor of 3 is used in the present implementations of the multilevel feature selection algorithm.

The number of levels in the multilevel hierarchy is another important parameter that can influence the performance of the multilevel feature selection technique. This parameter can be explicitly predefined based on empirical inferences or implicitly defined as a function of other parameters such as the reduction factor and the dimensionality of the original problem instance. I investigate the effect of the number of hierarchical levels on the performance of the multilevel feature selection technique using experiments that compare different instances of the technique having varying number of levels while the other parameters of the technique are constant. Similar caliberation experiments are performed using different instances of the multilevel feature selection algorithm having the number of levels set to 2, 3, 4, 5, and 6 while the other configurations remains constant. Table 4.3 (a, b, c, d, and e) and Figure 4.4 show the results of the experiment.

Table 4.3a The classification error rate values for the multilevel feature selection technique with 2 levels

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0837438	0.0738916	0.0689655	0.0837438	0.0591133	0.0689655	0.0738916	0.0640394	0.0738916	0.0788177
2	0.0492611	0.0640394	0.0837438	0.0591133	0.0541872	0.0738916	0.0689655	0.0689655	0.0591133	0.0738916
3	0.0591133	0.0788177	0.0788177	0.0492611	0.0640394	0.0640394	0.044335	0.0640394	0.0738916	0.0689655
4	0.0738916	0.0788177	0.0738916	0.0738916	0.0492611	0.0591133	0.0591133	0.0541872	0.0640394	0.0788177
5	0.0738916	0.0738916	0.0689655	0.0788177	0.0640394	0.0492611	0.0689655	0.0640394	0.0541872	0.0738916
Average	**0.06798**	**0.07389**	**0.07488**	**0.06897**	**0.05813**	**0.06305**	**0.06305**	**0.06305**	**0.06502**	**0.07488**

Average objective function value = **0.06729** Standard Deviation = ± **0.00582**

Table 4.3b The classification error rate values for the multilevel feature selection technique with 3 levels

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0640394	0.0640394	0.0738916	0.0689655	0.0738916	0.0541872	0.0640394	0.0689655	0.0541872	0.0788177
2	0.0689655	0.0738916	0.0788177	0.0689655	0.0689655	0.0591133	0.0738916	0.0689655	0.0640394	0.0738916
3	0.0541872	0.0689655	0.0837438	0.0640394	0.0640394	0.0492611	0.0591133	0.0492611	0.0541872	0.0788177
4	0.0689655	0.0738916	0.0738916	0.0541872	0.044335	0.0591133	0.0738916	0.0591133	0.0689655	0.0738916
5	0.0541872	0.0738916	0.0689655	0.0591133	0.0591133	0.0591133	0.0541872	0.0541872	0.0738916	0.0837438
Average	**0.06207**	**0.07094**	**0.07586**	**0.06305**	**0.06207**	**0.05616**	**0.06502**	**0.0601**	**0.06305**	**0.07783**

Average objective function value = **0.06562** Standard Deviation = ± **0.00700**

Table 4.3c The classification error rate values for the multilevel feature selection technique with 4 levels

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0689655	0.0591133	0.0738916	0.0837438	0.0689655	0.0541872	0.0591133	0.0541872	0.0738916	0.0788177
2	0.0541872	0.0541872	0.0837438	0.0738916	0.0541872	0.0492611	0.0689655	0.0689655	0.0788177	0.0738916
3	0.0591133	0.08867	0.0837438	0.0492611	0.0541872	0.0541872	0.0689655	0.08867	0.0591133	0.0837438
4	0.0738916	0.0640394	0.0788177	0.0689655	0.0591133	0.0738916	0.0788177	0.0591133	0.0640394	0.0837438
5	0.0492611	0.0541872	0.0689655	0.0640394	0.0541872	0.0689655	0.0541872	0.0689655	0.0640394	0.0738916
Average	**0.06108**	**0.06404**	**0.07783**	**0.06798**	**0.05813**	**0.0601**	**0.06601**	**0.06798**	**0.06798**	**0.07882**

Average objective function value = **0.06700** Standard Deviation = ± **0.00692**

Table 4.3d The classification error rate values for the multilevel feature selection technique with 5 levels

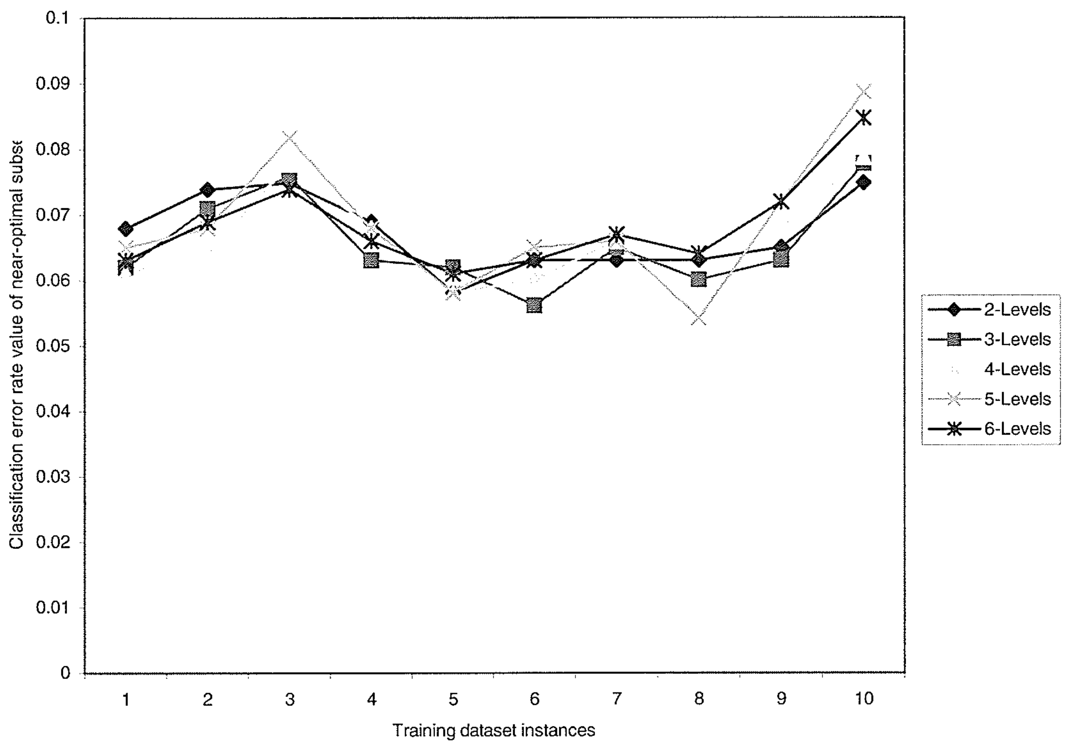
Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0591133	0.0640394	0.08867	0.0640394	0.0689655	0.0689655	0.0689655	0.0640394	0.0689655	0.08867
2	0.0541872	0.0738916	0.0738916	0.08867	0.044335	0.0591133	0.0788177	0.0689655	0.0541872	0.08867
3	0.0640394	0.0640394	0.0788177	0.0492611	0.0541872	0.0689655	0.0492611	0.0492611	0.0837438	0.0935961
4	0.0689655	0.0788177	0.0837438	0.0738916	0.0738916	0.0591133	0.0591133	0.0394089	0.08867	0.08867
5	0.0788177	0.0591133	0.0837438	0.0640394	0.0492611	0.0689655	0.0738916	0.0492611	0.0640394	0.0837438
Average	**0.06502**	**0.06798**	**0.08177**	**0.06798**	**0.05813**	**0.06502**	**0.06601**	**0.05419**	**0.07192**	**0.08867**

Average objective function value = **0.06867** Standard Deviation = ± **0.01021**

Table 4.3e The classification error rate values for the multilevel feature selection technique with 6 levels

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0788177	0.0788177	0.0738916	0.0640394	0.0591133	0.0640394	0.0640394	0.0689655	0.0788177	0.0738916
2	0.0492611	0.0689655	0.0738916	0.0738916	0.0640394	0.0640394	0.0689655	0.0492611	0.0738916	0.0837438
3	0.0492611	0.0591133	0.0738916	0.0640394	0.0738916	0.0591133	0.0640394	0.0788177	0.0689655	0.08867
4	0.0541872	0.0591133	0.0738916	0.0689655	0.0640394	0.0591133	0.0591133	0.0640394	0.0591133	0.0837438
5	0.0837438	0.0788177	0.0738916	0.0591133	0.044335	0.0689655	0.0788177	0.0591133	0.0788177	0.0935961
Average	**0.06305**	**0.06897**	**0.07389**	**0.06601**	**0.06108**	**0.06305**	**0.067**	**0.06404**	**0.07192**	**0.08473**

Average objective function value = **0.06837** Standard Deviation = ± **0.007047**



From Figure 4.4, varying the number of levels in the multilevel hierarchy seem not to show an obvious trend in the behaviour of the multilevel feature selection algorithm. Moreso, the number of levels that is possible in the hierarchy is implicitly restricted by the size of the feature selection problem instance and the value of the reduction factor, since the dimensionality of the feature subspaces at any level in the hierarchy cannot be less than the cardinality of the desired optimal subset. However, Table 4.3 (a, b, c, d, and e) shows that setting the number of levels to 3 or 4 can be appropriate for the given problem instance since the algorithm maintains a competitive classification error rate and stability for these values.

4.2 Search phase

The search phase involves finding a solution of the smallest problem instance. For the feature selection problem, the search phase finds a solution for the coarsest feature subspace. The solution generated by the search phase is presented to the refinement phase to be improved upon. This solution can be obtained using either an exact or an approximate search method. Exact searches yield optimal solutions for the given feature subspace, but there are strong limitations on the dimensionality of feature subspace that can be solved feasibly. To apply an exact search, the coarsening process has to be performed until a feature subspace with small dimensionality is obtained. In the context of the feature selection problem, there is need to consider the relevance of finding the optimal solution for the coarsest feature subspace to the quality of the desired near-optimal subset for the original feature space. Using a wrapper-based feature selection technique, a solution is usually optimal with respect to a target classifier. That is, the optimal solution can vary for different classifiers. Therefore, using an approximate solution at the coarsest feature subspace may not necessarily impair the quality of the near-optimal subset desired at the original feature space. Heuristic methods provide approximate solutions that are adequate irrespective of the dimensionality of the original feature space.

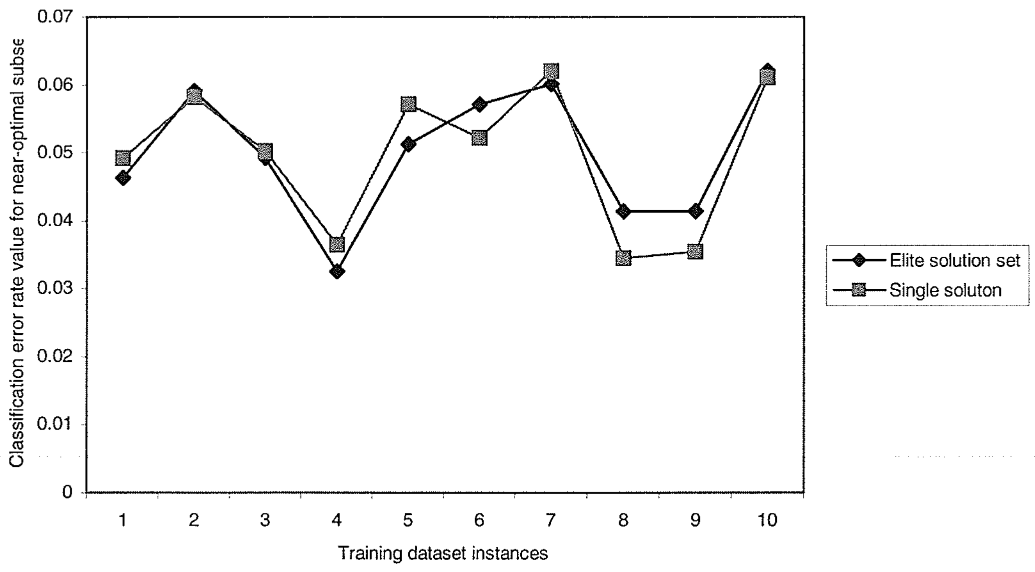
The search phase can be designed such that a single 'best' solution or a set of elite solutions is produced at the end of the phase, and the refinement phase can be initiated from the single 'best' solution or from the set of elite solutions. When a set of elite solutions is found, the refinement phase can be performed over the elite set by propagating and improving on the set of solutions across the levels. The set of elite solutions can be obtained in different ways. An approach is to search the coarsest level using different search methods and keeping the best solution obtained by each of the search methods. Another approach is to generate the set of elite solutions by randomly selecting k different initial solutions that are used to initiate different instances of the same search method. Also, the elite set can be generated using k best subsets found by a single search method. However, there is a potential constraint on working with a set of elite solutions: the additional computational cost required to compute the objective function more often may not worth the effort in terms of the quality of the final near-optimal solution. I perform experiments to investigate any influence on the performance of the multilevel feature selection technique when a single solution or an elite solution set is generated by the search phase. In the experiment, I configure two instances of the multilevel feature selection such that the search phase for one instance generates a single solution while the other instance generates a set of elite solutions at the coarsest feature subspace. The search phase generates the elite solution set by randomly selecting k initial solution that are used to initialize k instances of a search method (i.e. tabu search) and the solution from the instances constitute the elite set. The search phase uses the same search method to find the single solution and the solutions in the elite set. Any other search method applicable to feature selection problem can be used to find the solution(s) in the search phase. Allocating equivalent amount of computation resource to both search phase configurations, I compare the estimated values of the classification error rate using the leave-one-out cross validation for the multilevel algorithm instances having the two search phase configurations: using a single solution and an elite solution set. Table 4.4a, b and Figure 4.5 show the results of the experiments.

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0591133	0.0541872	0.0492611	0.0344828	0.0591133	0.044335	0.0591133	0.0295567	0.0394089	0.0591133
2	0.0394089	0.0640394	0.0541872	0.0394089	0.0591133	0.0640394	0.0591133	0.0295567	0.0295567	0.0689655
3	0.044335	0.0591133	0.0492611	0.0394089	0.0640394	0.0541872	0.0591133	0.044335	0.0344828	0.0591133
4	0.0541872	0.0541872	0.0492611	0.0394089	0.0492611	0.044335	0.0738916	0.0295567	0.0344828	0.0541872
5	0.0492611	0.0591133	0.0492611	0.0295567	0.0541872	0.0541872	0.0591133	0.0394089	0.0394089	0.0640394
Average	**0.04926**	**0.05813**	**0.05025**	**0.03645**	**0.05714**	**0.05222**	**0.06207**	**0.03448**	**0.03547**	**0.06108**

Average objective function value = 0.04966 Standard deviation = ± **0.01067**

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0492611	0.0591133	0.0541872	0.0344828	0.044335	0.0541872	0.0591133	0.0246305	0.0394089	0.0541872
2	0.0344828	0.0591133	0.044335	0.0344828	0.0541872	0.0591133	0.0591133	0.044335	0.044335	0.0640394
3	0.0492611	0.0591133	0.0492611	0.0295567	0.0541872	0.0591133	0.0689655	0.044335	0.0394089	0.0738916
4	0.044335	0.0591133	0.0492611	0.0344828	0.0492611	0.0541872	0.0640394	0.044335	0.044335	0.0640394
5	0.0541872	0.0591133	0.0492611	0.0295567	0.0541872	0.0591133	0.0492611	0.0492611	0.0394089	0.0541872
Average	**0.04631**	**0.05911**	**0.04926**	**0.03251**	**0.05123**	**0.05714**	**0.0601**	**0.04138**	**0.04138**	**0.06207**

Average objective function value = 0.05005 Standard Deviation = ± **0.00973**



As shown in Figure 4.5, there is no apparent performance difference in the multilevel feature selection technique when the search phase generates a single solution or a set of elite solution that is used in the subsequent phases of the technique. However, Table 4.3a, b show that the average estimated error rate over the 10 dataset instances is lower for the search phase configuration that generates a single solution than for the search phase configuration that generates an elite solution set. Also, the latter configuration generates fairly more stable estimated error rate since the standard deviation over the 10 dataset instances is lower. In the present implementations of the multilevel feature selection technique, I use the search phase configuration that generates a single solution for simplicity purposes.

4.3 Refinement phase

For the multilevel feature selection technique, the coarsening phase produces a hierarchy of coarse feature subspaces such that the subspace at level *i* is (explicitly or implicitly) a subset of the subspace at the next less coarse level *i*–1; the search phase produces the starting solution(s), i.e. a solution(s) of the coarsest feature subspace; and the refinement phase improves upon the starting solution(s) across the feature subspaces in the hierarchy with decreasing coarseness. The refinement phase improves upon the solutions(s) by interpolating the solutions(s) at a coarse level *i* onto the immediate less coarse level *i*-1, and refining the projected solution in the less coarse subspace. The interpolation and refining processes depend on how the features in the feature spaces are generated in the coarsening phase. When the coarse feature subspace at level *i* in the hierarchy consists of synthetic features generated from clusters of features from the next less coarse subspace at level *i*-1, interpolating the solution at level *i* onto level *i*-1 can involve decomposing each synthetic feature that belongs to the solution at level *i* into the constituent features at level *i*-1. To refine the solution, the features that result from the projection can be combined literally to form a subset of features wherein an initial solution can be selected and used to initiate the search heuristics over the subspace at level *i*-1. I refer to the search heuristics that are used for the refinement processes as *refinement heuristics*. When the coarse feature subspaces consist of original features that are selected using the feature pre-setting startegies, the interpolation can simply consist of using the 'best' solution (or solutions from the elite set) at level *i* as an initial solution for the refinement heuristics at level *i*-1. Then once a solution has been interpolated from level *i*, it can be improved upon by the refinement heuristics at level *i*–1. In the present implementation of the multilevel feature selection technique, the later form of interpolation is used since the coarsening is done using the *random feature pre-setting* strategy.

Unlike the search phase, the set of heuristics that can be used in the refinement phase is quite restricted. Search heuristics such as the greedy-like SFS, SBS, and their variants cannot be used as refinement heuristics since these methods usually create the '*optimal*' feature subset from a sequence of addition or elimination of features from a starting subset having a cardinality of 0 or L – the dimensionality of the original feature space. In the present implementation of the multilevel feature selection technique, the tabu search (as implemented in [53]) is used as the refinement heuristics.

To design the refinement phase of the multilevel feature selection technique, I consider the following decisions: whether to interpolate and refine a single best solution or a set of elite solutions across the levels; and whether to allocate a constant or varying amount of refinement resources. The first decision is guided by the results of the experiments in the search phase (Table 4.4a, b and Figure 4.5 above) that investigate the effect of using either a single solution or a set of elite solutions on the feature selection algorithm. The results show that there is no apparent performance difference in the multilevel feature selection technique when a single solution or an elite solution set is generated by the search phase and improved upon accordingly by the refinement phase. However, for the second decision, I investigate the effect of varying the allocation of the refinement resource across the levels on the performance of the multilevel feature selection algorithm. The refinement resource in this context refers to the cost of computing the objective function values in order to determine the discriminatory capability of an examined feature subset; and the allocation of the resource is based on the number of times the objective function evaluates the examined feature subset at each level during the refinement phase. Therefore, the allocation of resources directly relates to the number of iterations of the refinement heuristics at each level. I performed experiments to investigate three allocation possibilities: allocating equal amount of resource to refine the solution(s) at each level (i.e. constant resource allocation); increasing the amount of allocated resources with decreasing coarseness of the feature subspaces across the levels; and decreasing the amount of allocated resource with decreasing coarseness of the feature subspaces across the levels. The decrement or increment of the resource allocation across the levels is based on a simple arithmetic progression. The number of iterations for the level having the least amount of allocation is set at a value (i.e. the *basic number of iterations*) and the subsequent levels are increased progressively by a factor of the *basic number of iterations*. In the experiment, I create three instances of the multilevel feature selection algorithm such that each instance is based on one of the allocation possibilities. I compare the classification error rate derived using the leave-one-out cross validation for the three instances of the multilevel feature selection algorithm. Table 4.5a, b, c and Figure 4.6 show the effect of the three refinement scenarios on the performance of the multilevel feature selection technique.

Table 4.5a The classification error rate values for the multilevel feature selection technique with increasing number of iterations as coarseness decreases

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0640394	0.0837438	0.0689655	0.0837438	0.0738916	0.0985222	0.0640394	0.0837438	0.0738916	0.0985222
2	0.0541872	0.0689655	0.0492611	0.0640394	0.0689655	0.0689655	0.0689655	0.0640394	0.0689655	0.0689655
3	0.0738916	0.0640394	0.0837438	0.0541872	0.0640394	0.0738916	0.0640394	0.0541872	0.0640394	0.0738916
4	0.0591133	0.0689655	0.0492611	0.0738916	0.0738916	0.0689655	0.0591133	0.0738916	0.0738916	0.0689655
5	0.0492611	0.0837438	0.0640394	0.0591133	0.0738916	0.0689655	0.0689655	0.0591133	0.0738916	0.0689655
Average	**0.0600985**	**0.0738916**	**0.0630542**	**0.0669951**	**0.070936**	**0.0758621**	**0.0650246**	**0.0669951**	**0.070936**	**0.0758621**

Average objective function value = **0.06897** Standard Deviation = ± **0.00542**

Table 4.5b The classification error rate values for the multilevel feature selection technique with decreasing number of iterations as coarseness decreases

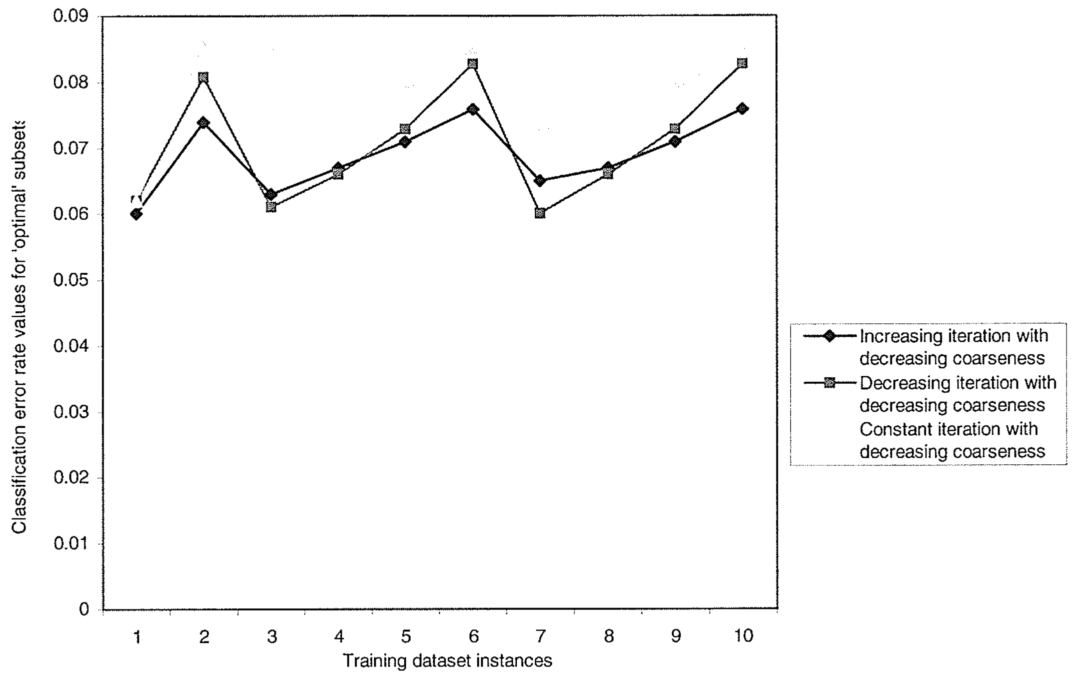
Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0492611	0.0935961	0.0738916	0.0689655	0.0689655	0.0738916	0.0640394	0.0689655	0.0689655	0.0738916
2	0.0640394	0.1034483	0.044335	0.0640394	0.0738916	0.0837438	0.0591133	0.0640394	0.0738916	0.0837438
3	0.0640394	0.0591133	0.0689655	0.0640394	0.0591133	0.08867	0.0541872	0.0640394	0.0591133	0.08867
4	0.0837438	0.0788177	0.0738916	0.0591133	0.0935961	0.0788177	0.0640394	0.0591133	0.0935961	0.0788177
5	0.0492611	0.0689655	0.044335	0.0738916	0.0689655	0.08867	0.0591133	0.0738916	0.0689655	0.08867
Average	**0.062069**	**0.0807882**	**0.0610837**	**0.0660099**	**0.0729064**	**0.0827586**	**0.0600985**	**0.0660099**	**0.0729064**	**0.0827586**

Average objective function value = **0.07074** Standard Deviation = ± **0.00898**

Table 4.5c The classification error rate values for the multilevel feature selection technique with constant number of iterations as coarseness decreases

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.0738916	0.0788177	0.0788177	0.0788177	0.0689655	0.0788177	0.0738916	0.0788177	0.0689655	0.0788177
2	0.0492611	0.0837438	0.0788177	0.08867	0.0837438	0.0837438	0.0591133	0.08867	0.0837438	0.0837438
3	0.0738916	0.0935961	0.0788177	0.0689655	0.08867	0.0935961	0.0689655	0.0689655	0.08867	0.0935961
4	0.0541872	0.0985222	0.1034483	0.0837438	0.0738916	0.0935961	0.0788177	0.0837438	0.0738916	0.0935961
5	0.0591133	0.0738916	0.0837438	0.0738916	0.0788177	0.0738916	0.0788177	0.0738916	0.0788177	0.0738916
Average	**0.062069**	**0.0857143**	**0.0847291**	**0.0788177**	**0.0788177**	**0.0847291**	**0.0719212**	**0.0788177**	**0.0788177**	**0.0847291**

Average objective function value = **0.07892** Standard Deviation = ± **0.00731**



As shown in Figure 4.6, when the number of iterations across the levels is varied (i.e. by decreasing or increasing), the multilevel feature selection algorithm finds subsets with consistently less classification error than when a constant number of iteration is maintained across the levels. This can be attributed to a more flexible exploration of the hierarchical search framework that is derivable by implicitly allocating more resources to the search method at levels where highly discriminatory subsets can be found. Besides, the configuration of the multilevel feature selection technique wherein the number of iteration increases as the coarseness decreases across the levels enables an intensive search in the subspaces with smaller dimensionality and a more diversified search in subspaces with larger dimensionality. This can explain the reason for the relatively more stable near-optimal subset generated by this configuration of the multilevel technique. In the present implementation of the multilevel feature selection technique, the allocation of refinement resources increases as the coarseness of the subspaces decreases across the levels.

Chapter 5

Experiments and Results

Generally, the evaluation of heuristics or meta-heuristics is based on empirical analysis since there are presently no known theoretical methods for evaluating or analysing these techniques. This chapter presents the empirical comparisons of the newly presented multilevel feature selection technique with other feature selection techniques. The following sections provide descriptions of the evaluation experiments and the biomedical datasets used in the experiments, and a discussion of the results of the experiments.

5.1 Experimental dataset

The dataset used in the experiments is a MRS dataset of biomedical origin from the National Research Council's Institute for Biodiagnostics (NRC-IBD). The dataset consists of 337 labelled samples (175 in the class with label '1' and 129 in the class with label '2') with a feature space dimensionality of 1500 and the cardinality of the desired optimal subset is set at 10. For each complete experiment run, the datasets is randomly partitioned into training and test sets in the ratio 2:1 and with corresponding amount of proportion from each class. That is the sample size of the training set is 203 (117 from class '1' and 86 from class '2') and the sample size of the test set is 101 (58 from class '1' and 43 from class '2'). The a priori class labels are used as the basis for the computation of the estimated classification error rates for the minimization objective function values, and the classification accuracies. To minimize over-fitting, for each training/test set dataset partition, the test set is independent of the training set and the test set is used for the external cross-validation purposes.

5.2 Evaluation experiments

Using a synthetic dataset, Zhang and Sun [53] empirically compare the performance of the basic tabu-search-based feature selection technique with other feature selection techniques such as SFS, SBS, GSFS, GSBS, plus-*l*-take-away-*r*, SFFS, SBFS, and GA. The result of the comparison shows that the SFS and the SBS require the least computational cost to obtain a solution (near-optimal feature subset), but these techniques obtain solution with the worst quality in terms of the estimated error rate. On the other hand, the tabu-search-based feature selection technique obtains better solutions than all the other techniques using competitive amount of computational cost. These inferences are used to set performance thresholds for the comparison experiments in this thesis.

Using a practical biomedical dataset (described in Section 5.1 above), I compare the performance of the newly presented multilevel feature selection technique with the tabu-search-based feature selection technique, the SFS technique, and a random feature selection technique. The implementation of the multilevel feature selection technique is based on the empirical recommendations for the calibration of the technique as presented in Chapter 4 above. The coarsening phase of the technique is based on the random variable pre-setting strategy with a reduction factor of 3 and the number of levels is also set at 3; the search phase generates a single solution to be improved upon during the refinement phase; and in the refinement phase, the allocation of the refinement resources is increased as the coarseness of the subspaces decreases across the hierarchical levels. The configuration of the tabu search method underlying the multilevel feature selection technique and the tabu-search-based feature selection technique is based on the recommendations in [53]. For the feature selection problem instance created by the biomedical dataset in context, the tabu list size is set at 30, the neighbourhood candidate list size is set at 100, and the initial solution is randomly selected based on a Gaussian random number generator. The SFS technique is implemented as a deterministic greedy-like method. The computational cost of this method is used as the upper limit of the computational requirement for the other techniques. The random feature selection technique simply evaluates the fitness of randomly selected feature subsets and the subset having the best evaluation is considered as the near-optimal subset. This technique is implemented to appraise any claims that the other feature selection techniques select features purely based on probabilistic chances.

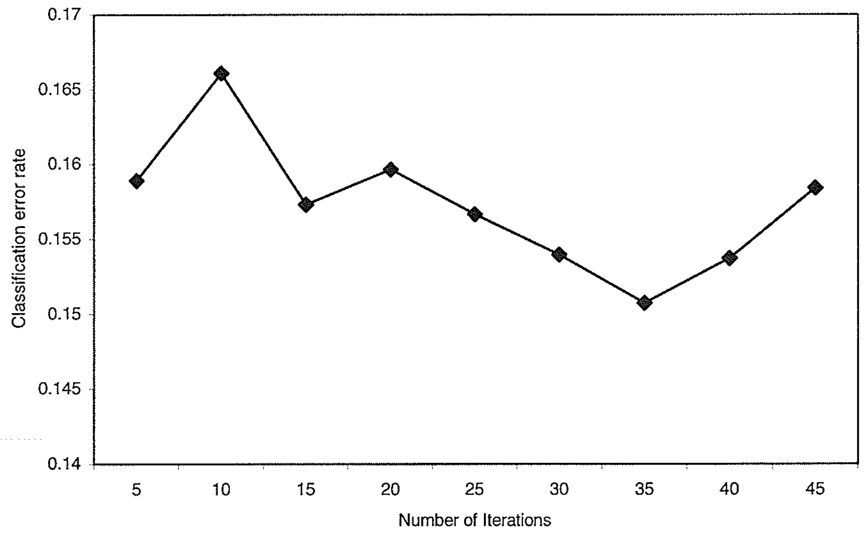
To provide a common basis for evaluating the examined feature selection techniques, an equivalent amount of computational cost is assigned to each technique. The computational cost is based on the number of times the objective function value is computed in a complete run of each technique. For instance, the computational cost of the SFS can be determined as follows: Given an original feature set **F** of cardinality L and the cardinality of the desired near-optimal subset is m; the computational cost **C** of the SFS technique is given as:

$$\mathbf{C} = \binom{L}{1} + \binom{L-1}{1} + \ldots + \binom{L-m}{1}$$

For the feature selection problem instance in context, $L = 1500$ and $m = 10$, therefore, $\mathbf{C} = 16445$. A fairly less amount of computational cost ($\mathbf{C} = 15000$) is allocated to the other feature selection techniques. For the multilevel feature selection technique, the computational cost is shared amongst the refinement heuristics according to the implemented refinement option (i.e. increasing computational cost with decreasing coarseness across the multilevel hierarchy). For the 3-level configuration of the multilevel technique, 25 basic iterations is assigned to the coarsest subspace; the next less coarse subspace is assigned 50 basic iterations; and the least subspace is assigned 75 basic iterations. For each coarse subspace, the neighbourhood size of the refinement heuristics (tabu search) is set at 100 and this implies that the computational cost per basic iteration is 100. Therefore the total computational cost **C** for the multilevel feature selection technique is given as:

$$\mathbf{C} = 25*100 + 50*100 + 75*100 = 15000.$$

The computational cost that is derived by assigning 25 basic iterations to the coarsest subspace is within the range of refinement resource allocation wherein over-fitting is less likely to occur using the multilevel feature selection technique. As shown in figure 5.1 below, the classification error rate on independent test set begins to increase continually when the number of basic iterations at the coarsest level is set at 35 and beyond.



That is, using the near-optimal feature subset selected by the multilevel technique as the underlying feature space in the design of a classifier, the classification performance degrades due to over-fitting when the computational cost assigned to the multilevel technique is 21000 (i.e. $\mathbf{C} = 35*100 + 70*100 + 105*100 = 21000$) and above.

For the tabu-search-based feature selection technique, the number of basic iterations is set at 150; therefore the total computational cost **C** is also given as:

$$\mathbf{C} = 150*100 = 15000.$$

For the random feature selection technique, the discriminatory capability of 15000 randomly selected subsets is examined and the optimal subset is the subset having the 'best' fitness or evaluation. Therefore, the computational cost for this technique is also 15000.

Besides, some of the examined techniques (multilevel feature selection, tabu-search-based feature selection, and random feature selection) have random components. Examples of the random components are: the random selection of subsets in the random feature selection; the random generation of the initial solution and the random selection of candidate set of solutions from the neighbourhood in the tabu search method underlying the tabu-search-based and multilevel feature selection; and the random coarsening of the feature subspaces in the multilevel feature selection. To normalize the randomness in these techniques, the complete run of these techniques are repeated on the same dataset instance for a number of times that is predefined by a *randomness factor*. The average of the evaluation parameters (i.e. the minimization objective function values, the classification accuracies on the training dataset instances, and the classification accuracies on the independent test dataset instances) is obtained and analysed for the evaluated feature selection techniques. The value of the randomness factor is set at 5 for the experiments implemented in this thesis.

To establish a trend in the comparison of the evaluated feature selection techniques, I perform the experiments on 10 randomly partitioned pairs of training and test sets from the biomedical dataset in order to establish a trend in the evaluation results. The evaluated techniques are implemented using Java 2 SDK Standard Edition version 1.4.2 on Microsoft Windows platform and the experiments are executed on Dell high performance desktop (Pentium 4 CPU 3.00GHz, 1.00GB of RAM) and IBM servers. The code listing of the implementation is presented in Appendix A.

5.3 Experimental results and discussions

The results of the experiments are shown in the following tables and figures (Table 5.1, 5.2, and 5.3; and Figure 5.1, 5.2, and 5.3). Table 5.1a, b, c, d and Figure 5.1 compare the classification error rate values of the near-optimal subset selected by the evaluated techniques; Table 5.2a, b, c, d and Figure 5.2 compare the training set classification accuracies of a simple LDA classifier that is designed using the near-optimal subsets selected by the evaluated techniques; and Table 5.3a, b, c, d and Figure 5.3 compare the test set classification accuracies of a simple LDA classifier that is designed using the near-optimal subsets selected by the evaluated techniques.

Table 5.1a The classification error rate values for the multilevel feature selection technique

Different dataset instances / same dataset instances	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	CER	CER	CER	CER	CER	CER	CER	CER	CER	CER
1	0.044335	0.0591133	0.0640394	0.0689655	0.0541872	0.0492611	0.0689655	0.0591133	0.0689655	0.0640394
2	0.0492611	0.0541872	0.0492611	0.0788177	0.0492611	0.0591133	0.0591133	0.0492611	0.0541872	0.0492611
3	0.0591133	0.0591133	0.0541872	0.0640394	0.0689655	0.0492611	0.0591133	0.0640394	0.044335	0.0640394
4	0.044335	0.0492611	0.0788177	0.0689655	0.0837438	0.0541872	0.0541872	0.0492611	0.0640394	0.0492611
5	0.0295567	0.0394089	0.0788177	0.0689655	0.0492611	0.0492611	0.0591133	0.0640394	0.0689655	0.0640394
Average	**0.0453202**	**0.0522167**	**0.0650246**	**0.0699507**	**0.0610837**	**0.0522167**	**0.0600985**	**0.0571429**	**0.0600985**	**0.0581281**

Average objective function value = **0.05813** Standard Deviation = ± **0.00698**

Table 5.1b The classification error rate values for the tabu-search-based feature selection technique

Different dataset instances / same dataset instances	**1** CER	**2** CER	**3** CER	**4** CER	**5** CER	**6** CER	**7** CER	**8** CER	**9** CER	**10** CER
1	0.0591133	0.0492611	0.0541872	0.0541872	0.0541872	0.044335	0.0591133	0.0689655	0.0640394	0.0591133
2	0.044335	0.0591133	0.0394089	0.0935961	0.0541872	0.0492611	0.0591133	0.0492611	0.0689655	0.0640394
3	0.0394089	0.0640394	0.0541872	0.0640394	0.0788177	0.0541872	0.0738916	0.0591133	0.0640394	0.044335
4	0.0591133	0.0541872	0.0492611	0.0837438	0.0788177	0.0541872	0.0591133	0.0541872	0.0541872	0.0541872
5	0.044335	0.0541872	0.0492611	0.0738916	0.0640394	0.0689655	0.0788177	0.0492611	0.0541872	0.0541872
Average	**0.0492611**	**0.0561576**	**0.0492611**	**0.0738916**	**0.0660099**	**0.0541872**	**0.0660099**	**0.0561576**	**0.0610837**	**0.0551724**

Average objective function value = **0.05872** Standard Deviation = ± **0.00794**

Table 5.1c The classification error rate values for the sequential forward selection technique

Different dataset instances / same dataset instances	**1** CER	**2** CER	**3** CER	**4** CER	**5** CER	**6** CER	**7** CER	**8** CER	**9** CER	**10** CER
1	0.0837438	0.0837438	0.0689655	0.1083744	0.0738916	0.0640394	0.08867	0.0985222	0.0788177	0.0837438
2	0.0837438	0.0837438	0.0689655	0.1083744	0.0738916	0.0640394	0.08867	0.0985222	0.0788177	0.0837438
3	0.0837438	0.0837438	0.0689655	0.1083744	0.0738916	0.0640394	0.08867	0.0985222	0.0788177	0.0837438
4	0.0837438	0.0837438	0.0689655	0.1083744	0.0738916	0.0640394	0.08867	0.0985222	0.0788177	0.0837438
5	0.0837438	0.0837438	0.0689655	0.1083744	0.0738916	0.0640394	0.08867	0.0985222	0.0788177	0.0837438
Average	**0.0837438**	**0.0837438**	**0.0689655**	**0.1083744**	**0.0738916**	**0.0640394**	**0.08867**	**0.0985222**	**0.0788177**	**0.0837438**

Average objective function value = **0.08325** Standard Deviation = ± **0.01323**

Table 5.1d The classification error rate values for the random selection feature selection technique

Different dataset instances / same dataset instances	**1** CER	**2** CER	**3** CER	**4** CER	**5** CER	**6** CER	**7** CER	**8** CER	**9** CER	**10** CER
1	0.08867	0.0985222	0.0985222	0.1083744	0.1034483	0.08867	0.1182266	0.0985222	0.1083744	0.1034483
2	0.0788177	0.0985222	0.08867	0.1083744	0.1083744	0.08867	0.1133005	0.0985222	0.1083744	0.1083744
3	0.08867	0.0935961	0.1034483	0.1182266	0.1083744	0.0837438	0.0985222	0.0985222	0.1034483	0.1034483
4	0.08867	0.0788177	0.0935961	0.1182266	0.0985222	0.0935961	0.1083744	0.0788177	0.0985222	0.1083744
5	0.0738916	0.0738916	0.08867	0.0985222	0.1133005	0.08867	0.1083744	0.1034483	0.1083744	0.1083744
Average	**0.0837438**	**0.0886699**	**0.0945813**	**0.1103448**	**0.1064039**	**0.0886699**	**0.1093596**	**0.0955665**	**0.1054187**	**0.1064039**

Average objective function value = **0.09892** Standard Deviation = ± **0.00980**

Table 5.2a The classification accuracies for the multilevel feature selection technique on the training set

Different dataset instances / same dataset instances	**1** CA	**2** CA	**3** CA	**4** CA	**5** CA	**6** CA	**7** CA	**8** CA	**9** CA	**10** CA
1	95.57%	94.09%	93.60%	93.10%	94.58%	95.07%	93.10%	94.09%	93.10%	94.09%
2	95.57%	95.07%	95.07%	92.12%	95.07%	94.09%	94.58%	95.07%	94.58%	95.07%
3	94.09%	94.09%	95.07%	93.60%	93.10%	95.07%	94.09%	93.60%	95.57%	93.60%
4	95.57%	95.07%	92.61%	93.10%	91.63%	94.58%	94.58%	95.07%	93.60%	95.07%
5	97.04%	96.55%	92.12%	93.10%	95.07%	95.07%	94.09%	93.60%	93.10%	93.60%
Average	95.57%	94.98%	93.69%	93.00%	93.89%	94.78%	94.09%	94.29%	93.99%	94.29%

Average classification accuracy = **94.26%** Standard Deviation = ± **0.72%** NB. **CA = Classification accuracy**

Table 5.2b The classification accuracies for the basic tabu search feature selection technique on the training set

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CA	CA	CA	CA	CA	CA	CA	CA	CA	CA
1	94.09%	95.07%	95.07%	94.58%	94.58%	95.57%	94.58%	93.10%	93.60%	94.09%
2	95.57%	94.09%	96.55%	90.64%	95.57%	95.07%	94.09%	95.07%	93.10%	93.60%
3	96.06%	93.60%	94.58%	93.60%	92.12%	94.58%	92.61%	94.09%	93.60%	95.57%
4	94.09%	94.58%	95.07%	91.63%	92.12%	94.58%	94.58%	94.58%	94.58%	94.58%
5	95.57%	94.58%	95.07%	92.61%	93.60%	93.10%	92.12%	95.07%	94.58%	94.58%
Average	**95.07%**	**94.38%**	**95.27%**	**92.61%**	**93.60%**	**94.58%**	**93.60%**	**94.38%**	**93.89%**	**94.48%**

Average classification accuracy = **94.19%** Standard Deviation = ± **0.79%**

Table 5.2c The classification accuracies for the sequential forward feature selection technique on the training set

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CA	CA	CA	CA	CA	CA	CA	CA	CA	CA
1	91.63%	91.63%	93.10%	90.64%	92.61%	93.60%	91.13%	91.13%	92.12%	92.12%
2	91.63%	91.63%	93.10%	90.64%	92.61%	93.60%	91.13%	91.13%	92.12%	92.12%
3	91.63%	91.63%	93.10%	90.64%	92.61%	93.60%	91.13%	91.13%	92.12%	92.12%
4	91.63%	91.63%	93.10%	90.64%	92.61%	93.60%	91.13%	91.13%	92.12%	92.12%
5	91.63%	91.63%	93.10%	90.64%	92.61%	93.60%	91.13%	91.13%	92.12%	92.12%
Average	**91.63%**	**91.63%**	**93.10%**	**90.64%**	**92.61%**	**93.60%**	**91.13%**	**91.13%**	**92.12%**	**92.12%**

Average classification accuracy = **91.97%** Standard Deviation = ± **0.93%**

Table 5.2d The classification accuracies for the random feature selection technique on the training set

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CA	CA	CA	CA	CA	CA	CA	CA	CA	CA
1	91.63%	90.64%	90.64%	89.16%	90.15%	91.63%	89.66%	92.12%	89.66%	90.15%
2	92.61%	90.64%	91.13%	89.66%	90.15%	91.63%	89.16%	91.13%	89.66%	89.16%
3	91.63%	90.64%	90.15%	88.18%	90.64%	92.12%	90.64%	90.64%	89.66%	89.66%
4	92.12%	92.61%	91.13%	88.67%	90.64%	91.13%	89.66%	92.12%	91.63%	89.16%
5	93.60%	92.61%	91.63%	91.13%	89.16%	92.12%	89.66%	90.15%	89.16%	89.16%
Average	**92.32%**	**91.43%**	**90.94%**	**89.36%**	**90.15%**	**91.72%**	**90.15%**	**90.15%**	**90.15%**	**89.46%**

Average classification accuracy = **90.58%** Standard Deviation = ± **0.98%**

Table 5.3a The classification accuracies for the multilevel feature selection technique on the test set

Different dataset instances / same dataset instances	1	2	3	4	5	6	7	8	9	10
	CA	CA	CA	CA	CA	CA	CA	CA	CA	CA
1	83.17%	80.20%	84.16%	88.12%	87.13%	83.17%	88.12%	84.16%	89.11%	83.17%
2	82.18%	85.15%	84.16%	83.17%	89.11%	84.16%	83.17%	80.20%	85.15%	85.15%
3	81.19%	85.15%	86.14%	82.18%	90.10%	84.16%	84.16%	89.11%	82.18%	86.14%
4	77.23%	84.16%	84.16%	93.07%	87.13%	83.17%	89.11%	82.18%	87.13%	84.16%
5	86.14%	87.13%	83.17%	87.13%	83.17%	85.15%	82.18%	82.18%	84.16%	87.13%
Average	**81.98%**	**84.36%**	**84.36%**	**86.73%**	**87.33%**	**83.96%**	**85.35%**	**83.56%**	**85.54%**	**85.15%**

Average Classification Accuracy = **84.83%** Standard Deviation = **±1.55%**

Table 5.3b The classification accuracies for the basic tabu search feature selection technique on the test set

Different dataset instances / same dataset instances	1 CA	2 CA	3 CA	4 CA	5 CA	6 CA	7 CA	8 CA	9 CA	10 CA
1	88.12%	83.17%	87.13%	88.12%	89.11%	83.17%	84.16%	77.23%	89.11%	85.15%
2	82.18%	88.12%	86.14%	78.22%	84.16%	84.16%	85.15%	78.22%	86.14%	86.14%
3	82.18%	83.17%	85.15%	81.19%	90.10%	86.14%	87.13%	82.18%	85.15%	83.17%
4	81.19%	81.19%	85.15%	77.23%	83.17%	83.17%	87.13%	89.11%	81.19%	88.12%
5	83.17%	83.17%	83.17%	85.15%	86.14%	83.17%	83.17%	81.19%	89.11%	88.12%
Average	**83.37%**	**83.76%**	**85.35%**	**81.98%**	**86.53%**	**83.96%**	**85.35%**	**81.58%**	**86.14%**	**86.14%**

Average classification accuracy = **84.42%** Standard Deviation = ± **1.76%**

Table 5.3c The classification accuracies for the sequential forward feature selection technique on the test set

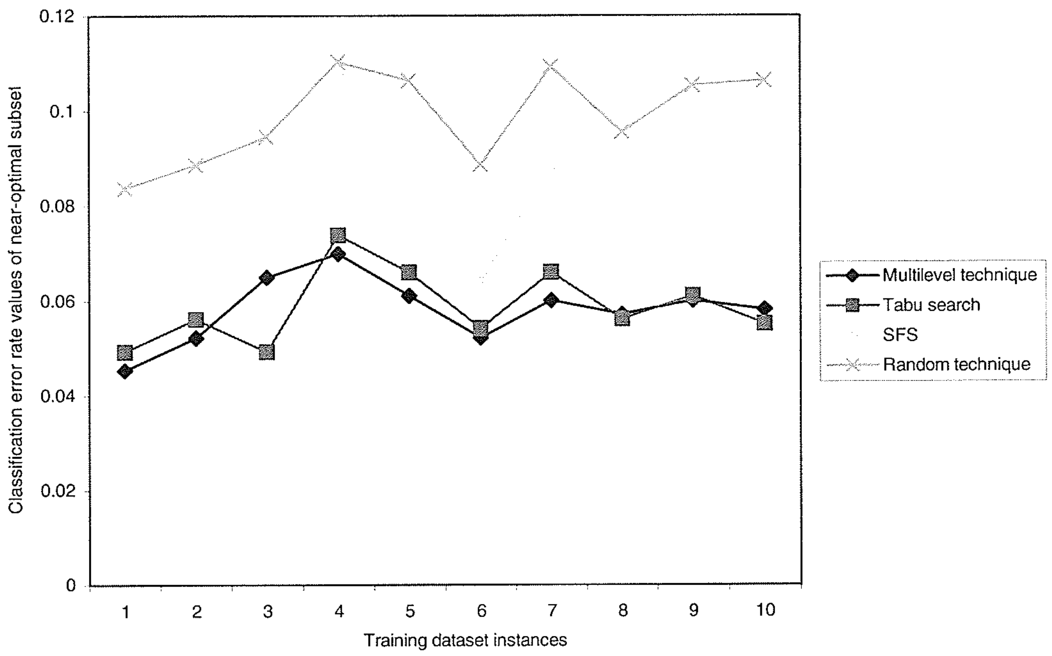
Different dataset instances / same dataset instances	1 CA	2 CA	3 CA	4 CA	5 CA	6 CA	7 CA	8 CA	9 CA	10 CA
1	83.17%	83.17%	84.16%	77.23%	86.14%	80.20%	85.15%	83.17%	86.14%	86.14%
2	83.17%	83.17%	84.16%	77.23%	86.14%	80.20%	85.15%	83.17%	86.14%	86.14%
3	83.17%	83.17%	84.16%	77.23%	86.14%	80.20%	85.15%	83.17%	86.14%	86.14%
4	83.17%	83.17%	84.16%	77.23%	86.14%	80.20%	85.15%	83.17%	86.14%	86.14%
5	83.17%	83.17%	84.16%	77.23%	86.14%	80.20%	85.15%	83.17%	86.14%	86.14%
Average	**83.17%**	**83.17%**	**84.16%**	**77.23%**	**86.14%**	**80.20%**	**85.15%**	**83.17%**	**86.14%**	**86.14%**

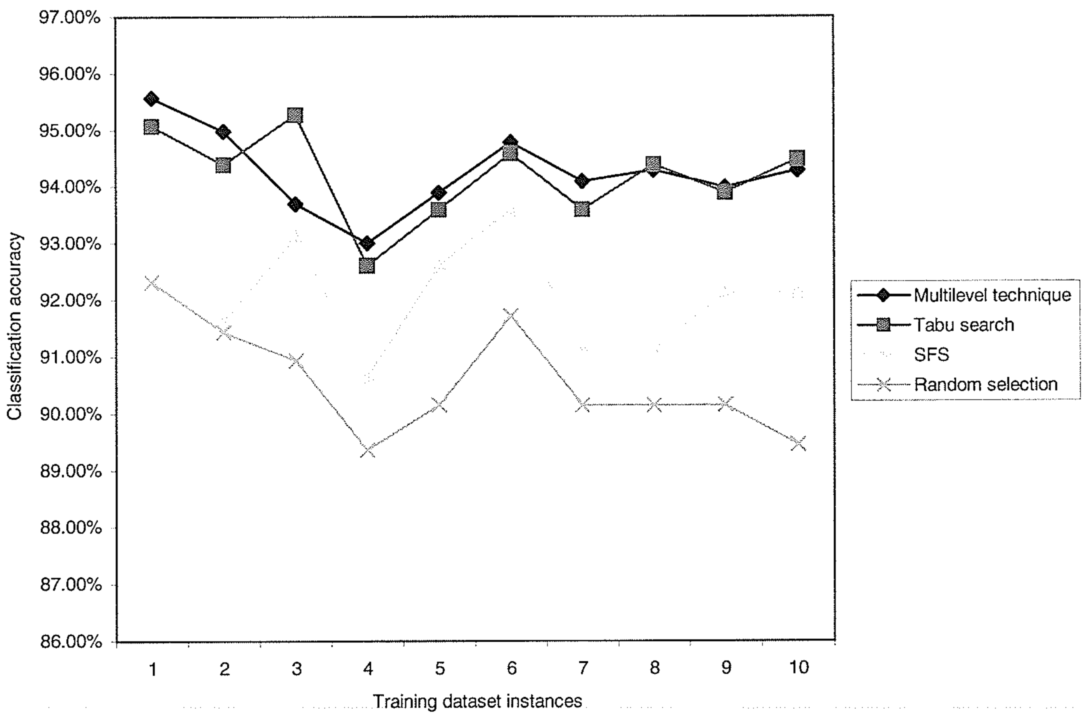
Average classification accuracy = **83.47%** Standard Deviation = ± **2.88%**

Table 5.3d The classification accuracies for the random feature selection technique on the test set

Different dataset instances / same dataset instances	1 CA	2 CA	3 CA	4 CA	5 CA	6 CA	7 CA	8 CA	9 CA	10 CA
1	80.20%	81.19%	89.11%	84.16%	83.17%	86.14%	87.13%	80.20%	81.19%	87.13%
2	78.22%	84.16%	84.16%	81.19%	87.13%	81.19%	89.11%	74.26%	83.17%	85.15%
3	84.16%	83.17%	85.15%	85.15%	92.08%	85.15%	84.16%	79.21%	84.16%	83.17%
4	76.24%	87.13%	84.16%	87.13%	87.13%	81.19%	83.17%	81.19%	83.17%	84.16%
5	83.17%	84.16%	86.14%	86.14%	87.13%	86.14%	86.14%	79.21%	83.17%	86.14%
Average	**80.40%**	**83.96%**	**85.74%**	**84.75%**	**87.33%**	**83.96%**	**85.94%**	**78.81%**	**82.97%**	**85.15%**

Average classification accuracy = **83.90%** Standard Deviation = ± **2.60%**


0.12
0.1
0.08
0.06
0.04
0.02
0
1
2
3
4
5
6
7
8
9
10
Training dataset instances
Classification error rate values of near-optimal subset
Multilevel technique
Tabu search
SFS
Random technique


97.00%
96.00%
95.00%
94.00%
93.00%
92.00%
91.00%
90.00%
89.00%
88.00%
87.00%
86.00%
1
2
3
4
5
6
7
8
9
10
Training dataset instances
Classification accuracy
Multilevel technique
Tabu search
SFS
Random selection

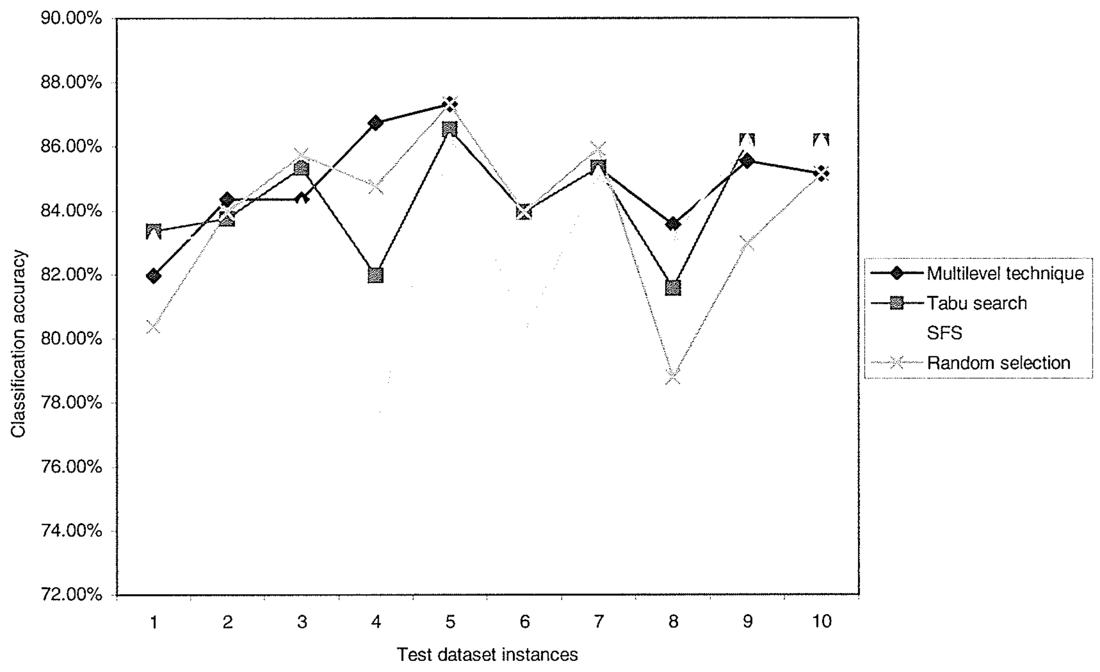


Figure 5.2 shows that the multilevel feature selection technique and the tabu-search-based feature selection techniques consistently selects near-optimal subsets that are better in terms of the minimization objective function values than the SFS and the random feature selection techniques. Besides, Table 5.1a and b show that the average minimization objective function value over the 10 dataset instances is better and more stable for the multilevel feature selection technique (0.05813 ± 0.00698) than for the tabu-search-based technique (0.05872 ± 0.00794). Moreover, by comparing the classification accuracies on the training set (Table 5.2 and Figure 5.3), the evaluated techniques exhibit a performance trend that is similar to the comparison of the minimization objective function values described above.

Comparing the classification accuracies of the evaluated techniques on the independent test set instances (Table 5.3 and Figure 5.4), the following enumerates the evaluated techniques in the order of increasing average classification accuracy and the stability of the selected solution: SFS (83.47% ± 2.88%), random feature selection technique (83.90% ± 2.60%), tabu-search-based feature selection technique (84.42% ± 1.76%), and multilevel feature selection technique (84.83% ± 1.55%).

Overall, the multilevel feature selection technique and the tabu-search-based feature selection technique are outstandingly better techniques than the other evaluated feature selection techniques, particularly for problem domains that are characterized with high dimensionality and small sample size. Moreover, the multilevel technique demonstrates better performance than the tabu-search-based technique in terms of the three evaluation parameters (i.e. minimization objective function values, classification accuracy on training set, and classification accuracy on independent test set). The outstanding performance of the better techniques can be attributed to the exploration strategy of the underlying search method for the techniques.

Moreover, using the GA-based feature selection technique presented in [33], regions of features (i.e. ranges of consecutive features) are selected and used to classify 25 randomly partitioned pair of training set and independent test set instances from the biomedical dataset used in the experiments described above. The average classification accuracy on the training and test set that is derived using the GA technique is 89.60% ± 2.12% and 84.90% ± 3.22% respectively. As presented above, the average classification accuracies of the multilevel feature selection technique (94.26% ± 0.72% and 84.83% ± 1.55%) on the same biomedical dataset is higher (for the training set) and more stable than the accuracies obtained using the GA technique.

Chapter 6

Conclusions

Feature selection problem is an interesting and challenging optimization problem that can be formulated as a 0-1 integer-programming problem. This research reviews existing and presents a new technique for resolving feature selection problem in application to biomedical datasets. This chapter presents a summary of the research and a description of the future direction of the research as presented in this thesis.

6.1 Summary of the research

The inherent characteristics of biomedical datasets (i.e. small sample size and high dimensionality) and the existence of different forms of these datasets pose challenges that make the feature selection problem in biomedical datasets particularly difficult. This research reviews existing feature selection techniques and investigates the application of some of these techniques (SFS, random feature selection, and tabu search feature selection) on practical biomedical datasets. More importantly, this research presents a new feature selection technique (i.e. multilevel feature selection) that is based on the multilevel search method. The performance of the new technique is influenced by calibration parameters such as the number of levels in the multilevel hierarchy, the reduction factor, the coarsening, search, and refinement strategies. The 'best' values of these calibration parameters are determined empirically.

The performance of the multilevel feature selection technique is evaluated in terms of the classification accuracies on training and test dataset instances of a simple LDA classifier that is designed based on the result of the feature selection technique.

Similarly, the performance of some existing feature selection techniques (SFS, random feature selection, and tabu search feature selection) is evaluated using equivalent amount of computational cost and the same dataset instances. The performance of the multilevel feature selection technique is compared with the existing feature selection techniques. The multilevel technique generates higher and more stable average classification accuracies on the training and test dataset instances than the other evaluated feature selection techniques.

6.2 Future work

In the present design of the multilevel feature selection technique, all the possible coarsening strategies have not been investigated. This research can be extended by exhaustively investigating the other coarsening options, since the capability of adapting the technique to different forms of biomedical datasets depends on the coarsening phase.

Besides, the newly presented multilevel feature selection technique is based on the simple *V-cycle* multilevel paradigm. An enhanced version of this technique can be developed using the multilevel cooperative search method [34, 47] that combines parallel multiple instances of the *V-cycle* multilevel methods that exchange search space exploration information in order to derive an overall enhanced search result. In future research work, the enhanced version of the multilevel feature selection technique will be designed and developed. The enhanced version can be used to resolve the feature selection problem formulation in (2) above. The result of the enhanced version can be used to identify biomarkers in biomedical datasets.

Appendix A. Code Listing of the Multilevel Festure Selection Algorithm

The following is the code listing for the implementation of the newly presented multilevel feature selection technique and the other existing feature selection techniques that are evaluated in this thesis.

```
/**.........................................................
 * basicTabuSearch.java
 * Implements the basic tabu search module
 * @author          Idowu Olayinka Oduntan
 * @version         August, 2005
 */
import java.lang.*;

public class basicTabuSearch
{
      public int[] tabuFinalSolution;
      public float tabuFinalValue;
      public int basicTabuCost = 0;

      // Declare object variables
      parameter localParamObj;
      localOptimum localOptimumObj;
      initialSolutionManager initialSolutionObj;

      // Declare other variables
      int[] initialSolution;
      int solutionSize, startDimension, endDimension;
      int localNoOfIterations, localNoOfTabuIteration, localNoOfLevels,
      localReductionFactor;
      int localNeighborhoodSize;
      int neighborhoodIncrement;
      String datasetType = "train";

      public basicTabuSearch(parameter paramObj)
      {
            // create object instance of required classes
            localParamObj = paramObj;
            startDimension = localParamObj.startDimension;
            endDimension = localParamObj.endDimension;
            solutionSize = localParamObj.finalSolutionSize;
            localNoOfIterations = localParamObj.noOfIterations;
            localNoOfLevels = localParamObj.noOfLevels;
            localNeighborhoodSize = localParamObj.neighborhoodSize;
            localReductionFactor = localParamObj.reductionFactor;
            initialSolutionObj = new initialSolutionManager(startDimension,
            endDimension, solutionSize);

            //Derive the equivalent number of iterations for basic tabu search
            localNoOfTabuIteration = localNoOfIterations *
            (localNoOfLevels*(localNoOfLevels+1)/2);

      }
```

```
        public void findTabuSearchSolution()
        {
                //Set the number of iterations of the local optimum search for the
                basic tabu search.
                localParamObj.setNoOfIterations(localNoOfTabuIteration);
                localParamObj.setNeighborhoodSize(localNeighborhoodSize);
                localOptimumObj = new localOptimum(localParamObj);
                //Generate initial solution for the basic tabu search.
                initialSolution =
                (int[])initialSolutionObj.starterInitialSolution().clone();

                //Find the optimal solution for the initial solution by calling
                localOptimum
                tabuFinalSolution =
                (int[])localOptimumObj.localOptimumSearch(initialSolution).clone()
                ;
                tabuFinalValue = localOptimumObj.localOptimumValue;
                basicTabuCost = localOptimumObj.computationCounter;
                localParamObj.resetNoOfIterations();
                localParamObj.resetNeighborhoodSize();

        }
}
----------------------End of basicTabuSearch.java-------------------------


/**.........................................................
 * Classifier.java
 * Implements the simple LDA classifier.
 * @author  Idowu Olayinka Oduntan
 * @version August 2005
 */

public class Classifier
{
        // Declare the variables
        int localSolutionSize, localNoOfItems;
        int[] localDataIndexA, localDataIndexB, localOptimalSubset,
        actualClassLabels, trainActualClassLabels, classifiedClassLabels,
        trainClassifiedClassLabels, localClassLabels;
        float[] localMeanVectorA, localMeanVectorB, localDataVector,
        trainLocalDataVector, constantVector, constantVector2;
        float[] diffMeanVector;
        float[] sumMeanVector;
        float[][] localDatasetA, localDatasetB, localCovarianceA,
        localCovarianceB, localInverseCovariance, localTestDataset,
        localTrainDataset;
        float tempMahalanobis, referenceValue, classificationValue,
        trainClassificationValue, classificationAccuracy,
        trainClassificationAccuracy;
        dataClass dataObject;
        matrix matrixObj;
        parameter paramObj;
        printClass printObj;
        objectiveFunction functionObj;

        public Classifier(int[] optimalSubset, parameter localParamObj)
        {
                // create instances of required class objects.
                paramObj = localParamObj;
                matrixObj = new matrix();
                functionObj = new objectiveFunction(paramObj);
        printObj = new printClass();
        // initialize generic variables.
        localOptimalSubset = (int[])optimalSubset.clone();
        localSolutionSize = localOptimalSubset.length;
        localClassLabels = paramObj.globalClassLabel;
        trainClassifier();
        testClassifier();

}

public void trainClassifier()
{
        // Assign data indexes for the training set.
        localDataIndexA = (int[])paramObj.globalTrainDataIndexA.clone();
        localDataIndexB = (int[])paramObj.globalTrainDataIndexB.clone();

        // Assign dataset values for the training set for current optimal
        solution
        localDatasetA = new
        float[localDataIndexA.length][localSolutionSize];
        localDatasetB = new
        float[localDataIndexB.length][localSolutionSize];

        int tempDataCounter = 0;
        localTrainDataset = new float[localDataIndexA.length +
        localDataIndexB.length][localSolutionSize];

        // Class labels for the classification of train set
        trainActualClassLabels = new int[localDataIndexA.length +
        localDataIndexB.length];
        trainClassifiedClassLabels = new int[localDataIndexA.length +
        localDataIndexB.length];

        for(int i=0; i<localDataIndexA.length; i++)
        {
                trainActualClassLabels[tempDataCounter] =
                localClassLabels[localDataIndexA[i]];
                for(int j=0; j<localSolutionSize; j++)
                {
                        localDatasetA[i][j] =
                        paramObj.globalDataSet[localDataIndexA[i]][localOptim
                        alSubset[j]-1];
                        localTrainDataset[tempDataCounter][j] =
                        paramObj.globalDataSet[localDataIndexA[i]][localOptim
                        alSubset[j]-1];
                }
                tempDataCounter++;
        }

        for(int i=0; i<localDataIndexB.length; i++)
        {
                trainActualClassLabels[tempDataCounter] =
                localClassLabels[localDataIndexB[i]];
                for(int j=0; j<localSolutionSize; j++)
                {
                        localDatasetB[i][j] =
                        paramObj.globalDataSet[localDataIndexB[i]][localOptim
                        alSubset[j]-1];
                        localTrainDataset[tempDataCounter][j] =
                        paramObj.globalDataSet[localDataIndexB[i]][localOptim
                        alSubset[j]-1];
                }
                tempDataCounter++;
        }

        localMeanVectorA =
        (float[])(functionObj.computeMeanVector(localDatasetA)).clone();
        localMeanVectorB =
        (float[])(functionObj.computeMeanVector(localDatasetB)).clone();

        localCovarianceA =
        (float[][])(functionObj.computeCovariance(localDatasetA,
        localMeanVectorA)).clone();
        localCovarianceB =
        (float[][])(functionObj.computeCovariance(localDatasetB,
        localMeanVectorB)).clone();

        localInverseCovariance =
        (float[][])functionObj.computeInverseSpooledCovariance(localCovari
        anceA, localDatasetA.length, localCovarianceB,
        localDatasetB.length).clone();

        sumMeanVector = (float[])(matrixObj.addVector(localMeanVectorA,
        localMeanVectorB)).clone();
        diffMeanVector =
        (float[])(matrixObj.subtractVector(localMeanVectorA,
        localMeanVectorB)).clone();

        constantVector =
        (float[])matrixObj.vectorMatrixMultiply(diffMeanVector,
        localInverseCovariance).clone();

        referenceValue = 0.5f *
        matrixObj.vectorVectorMultiply(constantVector, sumMeanVector);

        // Classify the train set
        for(int i=0; i<tempDataCounter; i++)
        {
                trainLocalDataVector =
                (float[])localTrainDataset[i].clone();
                trainClassificationValue = matrixObj.vectorVectorMultiply
                (constantVector, trainLocalDataVector);
                if(trainClassificationValue >= referenceValue)
                        trainClassifiedClassLabels[i] = 1;
                else if(trainClassificationValue < referenceValue)
                        trainClassifiedClassLabels[i] = 2;
        }
        trainClassificationAccuracy =
        classifierAccuracy(trainActualClassLabels,
        trainClassifiedClassLabels);

}

public void testClassifier()
{
        int dataCounter = 0;

        // Assign the test dataset indexes and declare class label entries
        for the test set.
        localDataIndexA = (int[])paramObj.globalTestDataIndexA.clone();
        localDataIndexB = (int[])paramObj.globalTestDataIndexB.clone();

        // For the classification of the test set
        actualClassLabels = new int[localDataIndexA.length +
        localDataIndexB.length];
```

```
            classifiedClassLabels = new int[localDataIndexA.length +
            localDataIndexB.length];

            localTestDataset = new float[localDataIndexA.length +
            localDataIndexB.length][localSolutionSize];

            for(int i=0; i<localDataIndexA.length; i++)
            {
                    actualClassLabels[dataCounter] =
                    localClassLabels[localDataIndexA[i]];
                    for(int j=0; j<localSolutionSize; j++)
                            localTestDataset[dataCounter][j] =
                            paramObj.globalDataSet[localDataIndexA[i]][localOptim
                            alSubset[j]-1];
                            dataCounter++;
            }

            for(int i=0; i<localDataIndexB.length; i++)
            {
                    actualClassLabels[dataCounter] =
                    localClassLabels[localDataIndexB[i]];
                    for(int j=0; j<localSolutionSize; j++)
                            localTestDataset[dataCounter][j] =
                            paramObj.globalDataSet[localDataIndexB[i]][localOptim
                            alSubset[j]-1];
                    dataCounter++;
            }

            for(int i=0; i<dataCounter; i++)
            {
                    localDataVector = (float[])localTestDataset[i].clone();
                    classificationValue = matrixObj.vectorVectorMultiply
                    (constantVector, localDataVector);
                    if(classificationValue >= referenceValue)
                            classifiedClassLabels[i] = 1;
                    else if(classificationValue < referenceValue)
                            classifiedClassLabels[i] = 2;
            }

            classificationAccuracy = classifierAccuracy(actualClassLabels,
            classifiedClassLabels);

    }

    public float classifierAccuracy(int[] actualSetLabels, int[]
    classifiedSetLabels)
    {
            int accuracyCounter = 0;
            for(int i=0; i<actualSetLabels.length; i++)
            {
                    if(actualSetLabels[i]==classifiedSetLabels[i])
                            accuracyCounter++;
            }

            return (accuracyCounter * 100.0f)/actualSetLabels.length;

    }
}
----------------------End of Classifier.java----------------------------

/**.....................................................................
 * coarser.java
 * This class generates a less coarse feature space from a given feature space
 * @author Idowu Olayinka Oduntan
 * @version August 2005
 */
import java.lang.Math;
public class coarser
{
       int[] levelSubSpaces;
       int[] coarseSubSpace;
       int[][] subSpacePartitions;
       int[][] featureSubspacePartitions;
       int[][] newFeatureSpacesWithComponents;
       int[][] coarseSubspaceElement;
       int[] remainderPartition;
       int[] partialSolution;
       int[] localGivenSolutionSpace;
       int localNoOfLevels, localNoOfPartition, localDimension,
       localReductionFactor, localPartitionSolutionSize,
       localNoOfPartialSolutions, tempLocalSolutionSize;
       int localClusterWindow, localTempDatasetSize;
       String localPartitionType;
       parameter paramObj;
       initialSolutionManager initialSolutionObj;

       coarser(parameter givenParamObj)
       {
              //Initializes local input parameters: givenFeatureSpcace,
              noOfPartitions, withinPartitionSearchParameters,
              paramObj = givenParamObj;
              localReductionFactor = paramObj.reductionFactor;
              tempLocalSolutionSize = paramObj.solutionSize;
              localNoOfLevels = paramObj.noOfLevels;
              localNoOfPartition = paramObj.noOfPartitions;
              localPartitionType = paramObj.partitionType;
              localPartitionSolutionSize = paramObj.solutionSize *
              localReductionFactor;
              localClusterWindow = paramObj.clusterWindow;
              featureSubspacePartitions = new int[localNoOfLevels][];
       }

       public void randomCoarsen(int[] givenSolutionSpace)
       {
              featureSubspacePartitions[0] = (int[])givenSolutionSpace.clone();
              int[] localSolutionSpace = (int[])givenSolutionSpace.clone();
              int[] tempSolutionList, tempSolutionList2;
              int levelSolutionSize;

              for(int j=1; j<localNoOfLevels; j++)
              {
                     levelSolutionSize =
                     localSolutionSpace.length/localReductionFactor;
                     tempSolutionList = (int[])localSolutionSpace.clone();
                     tempSolutionList2 = new int[levelSolutionSize];

                     for(int i=0; i<localSolutionSpace.length; i++)
                     {
                            int random1 = (int)(((localSolutionSpace.length - i)*
                            Math.random()) + i);
                            int temp1 = tempSolutionList[random1];
                            tempSolutionList[random1]=tempSolutionList[i];
                            tempSolutionList[i] = temp1;
                     }

                     for(int k=0; k<levelSolutionSize; k++)
```

```
                {
                        tempSolutionList2[k] = tempSolutionList[k];
                }
                featureSubspacePartitions[j] =
                (int[])tempSolutionList2.clone();
                localSolutionSpace = (int[])tempSolutionList2.clone();
        }
}

public int[] partitionFeatureSpace(int[] givenSolutionSpace)
{
        //For block or fixed partitioning
        localDimension = givenSolutionSpace.length;
        localGivenSolutionSpace = (int[])givenSolutionSpace.clone();
        localNoOfPartialSolutions = localDimension/(localReductionFactor *
        localNoOfPartition * localPartitionSolutionSize);

        int featureCounter=0; //Assign value feature counter
        int[] newSolutionSpace;
        int featureSubspaceSize = localDimension/localNoOfPartition;
        int featureSubspaceRemainder = localDimension%localNoOfPartition;
        featureSubspacePartitions = new int [localNoOfPartition-
        1][featureSubspaceSize];
        remainderPartition = new int [featureSubspaceSize +
        featureSubspaceRemainder];

        // /*Set parameter object to reflect partition values*/
        paramObj.setSolutionSize(localPartitionSolutionSize);
        paramObj.setTabuListSize(featureSubspaceSize/2);
        paramObj.setNeighborhoodSize(featureSubspaceSize);

        // Initialize the solution holder variable
        int[] tempDimension = new int [localDimension];
        for(int i=0; i<localDimension; i++)
        {
                tempDimension[i] = localGivenSolutionSpace[i];
        }

        if(localPartitionType.equals("random"))
        {
                for(int i=0; i<localDimension;i++)
                {
                        int r = (int)(((localDimension-i)* Math.random()) +
                        i);
                        int temp = tempDimension[r];
                        tempDimension[r]=tempDimension[i];
                        tempDimension[i] = temp;
                }
        }

        for(int i=0; i<localNoOfPartition-1;i++)
        {
                for(int j=0; j<featureSubspaceSize; j++)
                {
                        featureSubspacePartitions[i][j] =
                        tempDimension[featureCounter];
                        featureCounter++;

                }
        }

        for(int j=0; j<featureSubspaceSize + featureSubspaceRemainder;
        j++)
```

```
		{
			remainderPartition[j] = tempDimension[featureCounter];
			featureCounter++;
		}

		coarseSubSpace = (int[])coarsenSubSpaces().clone();
		paramObj.resetSolutionSize();
		paramObj.resetTabuListSize();
		paramObj.resetNeighborhoodSize();
		return coarseSubSpace;
	}

	public int[] coarsenSubSpaces()
	{
		//Assigns value to partitionSubSpaces[][];

		int[] remainderPartialSolution = new
		int[localPartitionSolutionSize];
		int[] tempPartialSolution;
		int[][] featurePartitionSolutions = new int[localNoOfPartition][];
		int partialSolutionCounter = 0;
		int solutionCounter = 0;
		boolean partialSolutionTester = false;
		int[] localCoarseSubSpace = new
		int[localNoOfPartition*localPartitionSolutionSize];
		for(int i =0; i<localNoOfPartition; i++)
		{
			if(i==localNoOfPartition-1)
			{
			for(int j=0; j<localNoOfPartialSolutions; j++)
			{
			initialSolutionObj = new
			initialSolutionManager(remainderPartition,
			localPartitionSolutionSize);
			localOptimum localOptimumObj = new localOptimum(paramObj,
			remainderPartition);
			int[] initialSolution =
			(int[])initialSolutionObj.starterInitialSolution().clone();
			tempPartialSolution =
			(int[])localOptimumObj.localOptimumSearch(initialSolution).
			clone();
			if(j>0)
			{
				for(int m=0; m<tempPartialSolution.length; m++)
				{
				for(int k=0; k<featurePartitionSolutions[i].length;
				k++)
				{
			if(tempPartialSolution[m]==featurePartitionSolutions[i][k])
			{
				partialSolutionTester = true;
			}
			}
			if(partialSolutionTester==false)
			featurePartitionSolutions[i][featurePartitionSolutions[i].l
			ength] = tempPartialSolution[m];
			partialSolutionTester = false;
			}
			}

			else
			{
        featurePartitionSolutions[i] =
        (int[])tempPartialSolution.clone();
        }

        }
        }

        else
        {
        for(int j=0; j<localNoOfPartialSolutions; j++)
        {
        initialSolutionObj = new
        initialSolutionManager(featureSubspacePartitions[i],
        localPartitionSolutionSize);
        localOptimum localOptimumObj = new localOptimum(paramObj,
        featureSubspacePartitions[i]);
        int[] initialSolution =
        (int[])initialSolutionObj.starterInitialSolution().clone();
        tempPartialSolution =
        (int[])localOptimumObj.localOptimumSearch(initialSolution).
        clone();
        if(j>0)
        {
        for(int m=0; m<tempPartialSolution.length; m++)
        {
        for(int k=0; k<featurePartitionSolutions[i].length; k++)
        {
        if(tempPartialSolution[m]==featurePartitionSolutions[i][k])
        {
        partialSolutionTester = true;
        }
        }
        if(partialSolutionTester==false)
        featurePartitionSolutions[i][featurePartitionSolutions[i].l
        ength] = tempPartialSolution[m];
        partialSolutionTester = false;
        }
        }
        else
        {
        featurePartitionSolutions[i] =
        (int[])tempPartialSolution.clone();
        }
        }

        }
}

for(int i =0; i<localNoOfPartition-1; i++)
{
for(int j=0; j<localPartitionSolutionSize; j++)
{
localCoarseSubSpace[solutionCounter] =
featurePartitionSolutions[i][j];
solutionCounter++;
}
}

for(int j=0; j<localPartitionSolutionSize; j++)
{
        localCoarseSubSpace[solutionCounter] =
        remainderPartialSolution[j];
        solutionCounter++;
        }
        return localCoarseSubSpace;

    }
    public void clusterContiguousCoarse(int[] givenFeatureSpace)
    {
        newFeatureSpacesWithComponents = new int[localNoOfLevels][];
        newFeatureSpacesWithComponents[0] =
        (int[])givenFeatureSpace.clone();
        featureSubspacePartitions[0] = (int[])givenFeatureSpace.clone();
        int[] tempGivenFeatureSpace = (int[])givenFeatureSpace.clone();

        for (int i=1; i<localNoOfLevels; i++)
        {
        int featureCounter = 0;
        int featureComponentCounter = 0;
        int newFeatureSpaceDimensionality;
        if(tempGivenFeatureSpace.length%localClusterWindow>0)
        {
        newFeatureSpaceDimensionality =
        tempGivenFeatureSpace.length/localClusterWindow+1;
        }
        else
        {
        newFeatureSpaceDimensionality =
        tempGivenFeatureSpace.length/localClusterWindow;
        }
        int[] newFeatureSpace = new int[newFeatureSpaceDimensionality];
        int[] tempNewFeatureSpacesWithComponents = new
        int[newFeatureSpaceDimensionality+tempGivenFeatureSpace.length];
        System.out.println("Length of new featurespace with components
        "+tempNewFeatureSpacesWithComponents.length);
        for(int j=0; j<newFeatureSpaceDimensionality; j++)
        {
        int featureCounter_2 = 0;
        int tempNewFeature = 0;
        for(int k=0; k<localClusterWindow; k++)
        {
        if(featureCounter<tempGivenFeatureSpace.length)
        {
        featureCounter_2++;
        tempNewFeature = tempNewFeature +
        tempGivenFeatureSpace[featureCounter];
        featureCounter++;
        }
        else
        break;
        }
        newFeatureSpace[j] = tempNewFeature/featureCounter_2;
        tempNewFeatureSpacesWithComponents[featureComponentCounter] =
        newFeatureSpace[j];
        featureComponentCounter++;
        int tempCounter = featureCounter_2;
        for(int m=featureComponentCounter;
        m<featureComponentCounter+featureCounter_2; m++)
        {
        tempNewFeatureSpacesWithComponents[m] =
        tempGivenFeatureSpace[featureCounter - tempCounter];
        tempCounter--;
        }
        featureComponentCounter = featureComponentCounter +
        featureCounter_2;
        }
```

```
                tempGivenFeatureSpace = (int[])newFeatureSpace.clone();
                featureSubspacePartitions[i] = (int[])newFeatureSpace.clone();
                newFeatureSpacesWithComponents[i] =
                (int[])tempNewFeatureSpacesWithComponents.clone();
        }
        }

        public void clusterUncontiguousCoarse(int[] givenFeatureSpace)
        {
                featureSubspacePartitions[0] = (int[])givenFeatureSpace.clone();
                int[] tempGivenFeatureSpace = (int[])givenFeatureSpace.clone();
        }

        public void biasedCoarser(int[] givenFeatureSpace)
        {
                featureSubspacePartitions[0] = (int[])givenFeatureSpace.clone();
                int[] tempGivenFeatureSpace = (int[])givenFeatureSpace.clone();
                int[] tempSolution = new int[tempLocalSolutionSize];

                for(int i=1; i<localNoOfLevels; i++)
                {
                int newTempFeatureSpaceSize =
                tempGivenFeatureSpace.length/localReductionFactor;
                int localWithinNoOfIteration =
                newTempFeatureSpaceSize/tempLocalSolutionSize;
                int[] newTempFeatureSpace = new int[newTempFeatureSpaceSize];
                int[] newTempFeatureSpace2;
                int[] rankedFeatureSpace1;
                int[] rankedFeatureSpace;
                int newSpaceSolutionCounter =0;
                int sameSolutionCounter = 0;
                boolean solutionItemFlag = false;
                rankFeatureSubset rankObj = new rankFeatureSubset(paramObj,
                tempGivenFeatureSpace);
                rankedFeatureSpace1 = (int[])rankObj.rankingSubset().clone();
                rankedFeatureSpace = (int[])rankObj.rankedSubspace.clone();
                for(int j=0; j<newTempFeatureSpaceSize; j++)
                {
                newTempFeatureSpace[j] =
                rankedFeatureSpace[rankedFeatureSpace.length-1-j];
                }

                featureSubspacePartitions[i] = (int[])newTempFeatureSpace.clone();
                tempGivenFeatureSpace = (int[])newTempFeatureSpace.clone();
                }
        }

}
-------------------------------- End of coarser.java ------------------------


/**...............................................................
 * dataClass.java
 *
 * Java class that reads the header and data content of the given dataset file
 * Partitions the data by stratification in the ratio 2:1 for train and test
set respectively.
 * @author    Idowu Olayinka Oduntan
 * @version   August 2005
 */

import java.io.BufferedReader;
import java.io.FileReader;
import java.util.StringTokenizer;

public class dataClass
{
      // Declare variables
      public int noOfHeaderLines;
      public int noOfHeaderParameters;
      public int noOfSamples;
      public int noOfDimension;
      public int noOfTrainSamples; // = 124;
      public int noOfClasses; //=2;
      public int classATrainCounter=0;
      public int classBTrainCounter=0;
      public int classATestCounter=0;
      public int classBTestCounter=0;
      public int testCounter;
      public int trainCounter;
      public int labelCounter;
      public int zeroClassCounter;
      public BufferedReader dataSetFile;
      public String delimitChars, dataFileName;
      public int[] dataSetParameters;
      public String partitionType;

      public int[] trainDataIndex;
      public int[] testDataIndex;
      public int[] trainDataIndexA;
      public int[] trainDataIndexB;
      public int[] testDataIndexA;
      public int[] testDataIndexB;

      public int[] classLabel;
      public int[] nonZeroLabelIndex;
      public float[][] dataSet;
      public float[][] trainClassADataSet;
      public float[][] trainClassBDataSet;
      public float[][] testClassADataSet;
      public float[][] testClassBDataSet;
      public String[] sampleEntities;
      public parameter paramObj;

      // instantiate dataclass object and get global parameters from paramObj
      public dataClass(parameter givenParamObj) //(int noOfHeader, int
      noOfHeaderParam, int samplesNo, int trainSampleNo, int noOfDimesionality,
      int classesNo, String delimiters, String fileName)
      {
            paramObj = givenParamObj;
            noOfHeaderLines = paramObj.noOfHeaderLines;
            noOfHeaderParameters = paramObj.noOfHeaderParameters;
            noOfSamples = paramObj.noOfSamples;
            noOfDimension = paramObj.noOfDimension;
            noOfClasses = paramObj.noOfClasses;
            dataSetParameters = new int[noOfHeaderParameters];
            dataSet = new float [noOfSamples][noOfDimension];
            sampleEntities = new String [noOfSamples];
            classLabel = new int [noOfSamples];
            delimitChars = paramObj.delimiterChar;
            dataFileName = paramObj.dataFile;
            partitionType = paramObj.partitionType;
            readData();
            partitionDataset();
            setDataVariables();
```

```
        }

        public void readHeader()
        {

               String dataSetLines;
               BufferedReader dataSetFile;
               StringTokenizer parameterString;
               FileReader dataFile;
               try
               {
                  dataFile = new FileReader (dataFileName);
                  dataSetFile = new BufferedReader(dataFile);
                  for (int i=0; i<noOfHeaderLines; i++)
                  {
                      int j=0;
                      dataSetLines = dataSetFile.readLine();
                      parameterString = new StringTokenizer(dataSetLines,
                      delimitChars, false);
                      while (j<noOfHeaderParameters)
                      {
                      if (parameterString.hasMoreTokens()==false)
                      break;

        dataSetParameters[j]=Integer.parseInt(parameterString.nextToken());
        j++;
        }
        }
        dataSetFile.close();
        dataFile.close();
     }

catch (Exception e0)
{
System.out.println("Datafile is not available...! - One");
}

}

public void readData()
{
        String dataSetLines;
        BufferedReader dataSetFile;
        StringTokenizer parameterString;
        try
        {
          dataSetFile = new BufferedReader(new FileReader(dataFileName));
          for (int i=1; i<=noOfHeaderLines; i++)
          dataSetLines = dataSetFile.readLine();
          for (int i=0; i<noOfSamples; i++) // For each sample, read a line to et
        the feature values and class label
        {
           int j=0;
           dataSetLines = dataSetFile.readLine();
           parameterString = new StringTokenizer(dataSetLines, delimitChars,
false);
        if (parameterString.hasMoreTokens()==true)
        {
        sampleEntities[i] = parameterString.nextToken();
        while (j<noOfDimension)    // Read feature values
        {
        if (parameterString.hasMoreTokens()==false)
        break;
        dataSet[i][j]=Float.parseFloat(parameterString.nextToken());
        j++;
        }
        if (j==noOfDimension)      // Read sample's class label
        classLabel[i]= Integer.parseInt(parameterString.nextToken());
        }
        }
        dataSetFile.close();
      }
        catch (Exception e0)
        {
        System.out.println (e0.getMessage());
        System.out.println("Datafile file is not available...! - Two");
        }
        }
        public void partitionDataset()    // Partition dataset into train and
        test sets .
        {
              int counter0 = 0;
              int counterA = 0;
              int counterB = 0;

        // Get the no of items with real class labels A or B and unreal class
        label 0
        for(int i =0; i<classLabel.length; i++)
        {
              if(classLabel[i]==0)
                    counter0++;
              if(classLabel[i]==1)
                    counterA++;
              if(classLabel[i]==2)
                    counterB++;
        }
        int[] classAIndex = new int[counterA];
        int[] classBIndex = new int[counterB];
        int classACounter = 0;
        int classBCounter = 0;
        for(int i =0; i<classLabel.length; i++)
        {
              if(classLabel[i]==1)
              {
                    classAIndex[classACounter] = i;
                    classACounter++;
              }
                    if(classLabel[i]==2)
                    {
                    classBIndex[classBCounter] = i;
                    classBCounter++;
              }
        }
        System.out.println("ClassAcounter: "+classAIndex.length);
        System.out.println("ClassBcounter: "+classBIndex.length);
        // Determine the number of samples from each class that constitute the
training set - stratification
        float tempA  = (2f/3f)*classACounter;
        float tempB  = (2f/3f)*classBCounter;
        int noOfClassAtrainSamples = Math.round(tempA);
        int noOfClassBtrainSamples = Math.round(tempB);
        noOfTrainSamples = noOfClassAtrainSamples + noOfClassBtrainSamples;
        trainDataIndex = new int[noOfTrainSamples];
        testDataIndex = new int[classACounter + classBCounter -
        noOfTrainSamples];
```

```
if(partitionType.equals("random"))      // Partition dataset randomly
{
        int[] tempIndexA = (int[])classAIndex.clone();
        int[] tempIndexB = (int[])classBIndex.clone();
        for(int i=0; i<classACounter; i++)
        {
        int randomCounter = (int)(((classACounter - i) * Math.random()) +
        i);
        int temp = tempIndexA[randomCounter];
        tempIndexA[randomCounter]=tempIndexA[i];
        tempIndexA[i] = temp;
        }

        for(int i=0; i<classBCounter; i++)
        {
        int randomCounter = (int)(((classBCounter - i) * Math.random()) +
        i);
        int temp = tempIndexB[randomCounter];
        tempIndexB[randomCounter]=tempIndexB[i];
        tempIndexB[i] = temp;
        }
        int trainCounter=0;
        for(int j=0; j<noOfClassAtrainSamples; j++)
        {
                trainDataIndex[trainCounter] = tempIndexA[j];
                trainCounter++;
        }
        for(int j=0; j<noOfClassBtrainSamples; j++)
        {
                trainDataIndex[trainCounter] = tempIndexB[j];
                trainCounter++;
        }

        int localIndexCounter = 0;
        for(int j=noOfClassAtrainSamples; j<classACounter; j++)
        {
                testDataIndex[localIndexCounter] = tempIndexA[j];
                localIndexCounter++;
        }
        for(int j=noOfClassBtrainSamples; j<classBCounter; j++)
        {
                testDataIndex[localIndexCounter] = tempIndexB[j];
                localIndexCounter++;
        }

        System.out.println("\nSize of training set:
        "+trainDataIndex.length);
        System.out.println("\nSize of test set "+testDataIndex.length);
        }

        int classCounterA = 0;
        int classCounterB = 0;
        for(int i=0; i<trainDataIndex.length; i++)
        {

        if(classLabel[trainDataIndex[i]]==1)
        {
                classCounterA++;
        }

        if(classLabel[trainDataIndex[i]]==2)
        {
```

```
        classCounterB++;
}
}

trainDataIndexA = new int[classCounterA];
trainDataIndexB = new int[classCounterB];
trainClassADataSet = new float[classCounterA][noOfDimension];
trainClassBDataSet = new float[classCounterB][noOfDimension];

classCounterA = 0;
classCounterB = 0;
for(int i=0; i<trainDataIndex.length; i++)
{
if(classLabel[trainDataIndex[i]]==1)
{
        trainDataIndexA[classCounterA] = trainDataIndex[i];
        for(int j=0; j<noOfDimension; j++)
        trainClassADataSet[classCounterA][j] =
        dataSet[trainDataIndex[i]][j];
        classCounterA++;
}

if(classLabel[trainDataIndex[i]]==2)
{
        trainDataIndexB[classCounterB] = trainDataIndex[i];
        for(int j=0; j<noOfDimension; j++)
        trainClassBDataSet[classCounterB][j] =
        dataSet[trainDataIndex[i]][j];
        classCounterB++;
}
}

classCounterA = 0;
classCounterB = 0;
for(int i=0; i<testDataIndex.length; i++)
{
if(classLabel[testDataIndex[i]]==1)
{
        classCounterA++;
}
if(classLabel[testDataIndex[i]]==2)
{
        classCounterB++;
}

}

testDataIndexA = new int[classCounterA];
testDataIndexB = new int[classCounterB];
testClassADataSet = new float[classCounterA][noOfDimension];
testClassBDataSet = new float[classCounterB][noOfDimension];

classCounterA = 0;
classCounterB = 0;

for(int i=0; i<testDataIndex.length; i++)
{
        if(classLabel[testDataIndex[i]]==1)
        {
                testDataIndexA[classCounterA] = testDataIndex[i];
                for(int j=0; j<noOfDimension; j++)
                testClassADataSet[classCounterA][j] =
                dataSet[testDataIndex[i]][j];
```

```
                        classCounterA++;
                    }

                if(classLabel[testDataIndex[i]]==2)
                {
                    testDataIndexB[classCounterB] = testDataIndex[i];
                    for(int j=0; j<noOfDimension; j++)
                    testClassBDataSet[classCounterB][j] =
                    dataSet[testDataIndex[i]][j];
                    classCounterB++;
                    }
                }
            }

        public void setDataVariables() // Reset data-related global parameters
        {
            paramObj.setGlobalDataSet((float[][])dataSet.clone());
            paramObj.setGlobalClassLabel((int[])classLabel.clone());

            paramObj.setGlobalTrainDataIndex((int[])trainDataIndex.clone());
            paramObj.setGlobalTrainDataIndexA((int[])trainDataIndexA.clone());
            paramObj.setGlobalTrainDataIndexB((int[])trainDataIndexB.clone());
            paramObj.setGlobalTrainClassADataSet((float[][])trainClassADataSet
            .clone());
            paramObj.setGlobalTrainClassBDataSet((float[][])trainClassBDataSet
            .clone());
            paramObj.setGlobalTestDataIndex((int[])testDataIndex.clone());
            paramObj.setGlobalTestDataIndexA((int[])testDataIndexA.clone());
            paramObj.setGlobalTestDataIndexB((int[])testDataIndexB.clone());
            paramObj.setGlobalTestClassADataSet((float[][])testClassADataSet.c
            lone());
            paramObj.setGlobalTestClassBDataSet((float[][])testClassBDataSet.c
            lone());

        }

}

--------------------------End of dataClass------------------------------------


/**.............................................................
 * eliteSolution.java
 * Manages the elite solution set.
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
*/
public class eliteSolution
{
        int localEliteSolutionSize, localSolutionSize;
        public int[][] eliteSolutionSet;
        public float[] eliteSolutionValues;
        public int bottomQueueCounter;
        public boolean foundInEliteSet;
        public int eliminateIndex;

        public eliteSolution(int[][] givenEliteSolutionSet, float[]
        givenSolutionValue)
        {
            localEliteSolutionSize = eliteSolutionSet.length;
            localSolutionSize = eliteSolutionSet[0].length;
            eliteSolutionSet = (int[][])givenEliteSolutionSet.clone();
            eliteSolutionValues = (float[])givenSolutionValue.clone();
```

```
                parameter paramObj = new parameter();
                if(paramObj.optimumType.equals("MIN"))
                        eliminateIndex = 0;
                else
                eliminateIndex = givenEliteSolutionSet.length-1;
        }

        public void addToEliteSet(int[] solution, float solutionValue)
        {
        boolean eliteSetChecker = checkInEliteSet((int[])solution.clone(),
        solutionValue);
        for(int m=0; m<eliteSolutionValues.length-1; m++)
        {
                for(int n=0; n<eliteSolutionValues.length-1-m; n++)
                {
                if(eliteSolutionValues[n] < eliteSolutionValues[n+1])
                {
                float temp1 = eliteSolutionValues[n];
                eliteSolutionValues[n] = eliteSolutionValues[n+1];
                eliteSolutionValues[n+1] = temp1;
                int[] temp2 = (int[])eliteSolutionSet[n].clone();
                eliteSolutionSet[n] = (int[])eliteSolutionSet[n+1].clone();
                eliteSolutionSet[n+1] = (int[])temp2.clone();
        }
}
}

if(solutionValue<eliteSolutionValues[eliminateIndex])
{
        removeFromEliteSet();
        for (int i=0; i<localSolutionSize; i++)
                eliteSolutionSet[eliminateIndex][i] = solution[i];
                eliteSolutionValues[eliminateIndex] = solutionValue;
}
}

public void removeFromEliteSet()
{
        for (int i=0; i<localEliteSolutionSize-1; i++)
        {
        for (int j=0; j<localSolutionSize; j++)
        eliteSolutionSet[i][j] = eliteSolutionSet[i+1][j];
        eliteSolutionValues[i] = eliteSolutionValues[i+1];
}
}

public boolean checkInEliteSet(int[] solution, float solutionValue)
{
        foundInEliteSet = false;
        int eliteSetCounter = 0;
        while(eliteSetCounter<localEliteSolutionSize)
        {
                int solutionCounter = 0;
                while(solutionCounter < solution.length &&
                (eliteSolutionSet[eliteSetCounter][solutionCounter]==solution[solu
                tionCounter]))
                {
                        solutionCounter++;
                }
                if(solutionCounter>=solution.length-1)
                {
                        foundInEliteSet = true;
                        break;
                }
                eliteSetCounter++;
                }
                return foundInEliteSet;
}

}
------------------------ End of eliteSolution.java-------------------------


/**.........................................................
 * initialSolutionManager.java
 * Generates an initial solution for the search routines (basic tabu search and
 the multilevel search)
 * @author          Idowu Olayinka Oduntan
 * @version         August 2005
 */

public class initialSolutionManager
{
        int[] newDatasetIndex;
        public int noOfDimensionSpace;
        public int solutionLength;
        public int[] initialSolution;
        int startingDimensionLocal, endingDimensionLocal;

        public initialSolutionManager(int startingDimension, int endingDimension,
int solutionSize)
        {

                startingDimensionLocal = startingDimension;
                endingDimensionLocal = endingDimension;
                noOfDimensionSpace = endingDimension - startingDimension + 1;
                solutionLength = solutionSize;
                newDatasetIndex = new int[noOfDimensionSpace];
                initialSolution = new int [solutionSize];
                int solutionSpaceIndex = startingDimension;
                for (int i=0; i<noOfDimensionSpace; i++)
                {
                        newDatasetIndex[i] = solutionSpaceIndex + i;
                }
        }
        public initialSolutionManager(int[] subFeatureSpace, int solutionSize)
        {
                noOfDimensionSpace = subFeatureSpace.length;
                solutionLength = solutionSize;
                newDatasetIndex = new int[noOfDimensionSpace];
                initialSolution = new int [solutionSize];
                newDatasetIndex = (int[])subFeatureSpace.clone();

        }

        public int[] starterInitialSolution()
        {
                int[] tempNewDatasetIndex = new int[noOfDimensionSpace];
                for (int j=0; j<noOfDimensionSpace; j++)
                tempNewDatasetIndex[j] = newDatasetIndex[j];
                for(int i=0; i<noOfDimensionSpace; i++)
                {
                        int randomCounter = (int)(((endingDimensionLocal -
                        (i+startingDimensionLocal))* Math.random()) + i);
                        int temp = tempNewDatasetIndex[randomCounter];
                        tempNewDatasetIndex[randomCounter]=tempNewDatasetIndex[i];
                        tempNewDatasetIndex[i] = temp;
                }
                for(int k=0;k<solutionLength;k++)
                        initialSolution[k] = tempNewDatasetIndex[k];

                return (int[])initialSolution.clone();

        }

        public int[] starterInitialSolution(boolean featureSpaceFlag)
        {
                int[] tempNewDatasetIndex = new int[noOfDimensionSpace];
                for (int j=0; j<noOfDimensionSpace; j++)
                        tempNewDatasetIndex[j] = newDatasetIndex[j];
                for(int i=0; i<noOfDimensionSpace; i++)
                {
                        int randomCounter = (int)(((noOfDimensionSpace - i) *
                        Math.random()) + i);
                        int temp = tempNewDatasetIndex[randomCounter];
                        tempNewDatasetIndex[randomCounter]=tempNewDatasetIndex[i];
                        tempNewDatasetIndex[i] = temp;
                }
                for(int k=0;k<solutionLength;k++)
                {
                        initialSolution[k] = tempNewDatasetIndex[k];
                }

                return (int[])initialSolution.clone();

        }

}
------------------------End of initialSOlutionManager.java-------------------


/**.....................................................
 * localOptimum.java
 * Finds the local optimum for a given solution and neighborhood.
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
 */
public class localOptimum
{
        public int[][] neighborList;
        public int[] localOptimumSolution;
        public float localOptimumValue;
        public int[][] optimalEliteSolutionSet;
        public float[] optimalEliteSolutionSetValues;
        public int computationCounter = 0;

        float[][] localClassADataSet;
        float[][] localClassBDataSet;
        float[][] localDataset;
        float[][] currentClassADataSet;
        float[][] currentClassBDataSet;
        float[] currentAMeanVector;
        float[] currentBMeanVector;
        float[][] currentClassACovariance;
        float[][] currentClassBCovariance;
        int[] currentSolution;
        int[] localDataIndexA;
        int[] localDataIndexB;
        int[] localGivenSolutionSpace;
```

```
int localNeighborSize, localNoOfIterations, localTabuListSize,
localSolutionSize, localNoOfEliteSolutions;
float localClassifierThreshold;

String fileName, delimiters, localDatasetType, localOptimumType;
boolean searchSpaceGiven;
dataClass dataObject;
matrix matrixObj;
neighborhood neighborObj;
objectiveFunction functionObj;
tabuList tabuObj;

localOptimum(parameter paramObj)
{
       // initialize variables
       localNeighborSize = paramObj.neighborhoodSize;
       localNoOfIterations = paramObj.noOfIterations;
       localTabuListSize = paramObj.tabuListSize;
       localSolutionSize = paramObj.solutionSize;
       localDatasetType = paramObj.datasetType;
       localOptimumType = paramObj.optimumType;

       //create an instance each of the required objects
       matrixObj = new matrix();
       functionObj = new objectiveFunction(paramObj);
       neighborObj = new neighborhood(paramObj);
       tabuObj = new tabuList(localTabuListSize, localSolutionSize,
       paramObj);
       fileName = paramObj.dataFile;
       delimiters = paramObj.delimiterChar;
       searchSpaceGiven = false;
}


localOptimum(parameter paramObj, int[] givenSolutionSpace)
{
       // create a copy of the given solution space
       localGivenSolutionSpace = (int[])givenSolutionSpace.clone();
       localNeighborSize = paramObj.neighborhoodSize;
       localNoOfIterations = paramObj.noOfIterations;
       localTabuListSize = paramObj.tabuListSize;
       localSolutionSize = paramObj.solutionSize;
       localDatasetType = paramObj.datasetType;
       localOptimumType = paramObj.optimumType;
       localNoOfEliteSolutions = paramObj.noOfEliteSolutions;
       matrixObj = new matrix();
       functionObj = new objectiveFunction(paramObj);
       neighborObj = new neighborhood(paramObj, localGivenSolutionSpace);
       tabuObj = new tabuList(localTabuListSize, localSolutionSize,
       paramObj);
       fileName = paramObj.dataFile;
       delimiters = paramObj.delimiterChar;
       searchSpaceGiven = true;
}

public float computeObjValue(int[] contextSolution)
{
       int[] localCurrentSolution = (int[])contextSolution.clone();
       float objFunctValue;
       objFunctValue = functionObj.meanSquareError(localCurrentSolution);
       computationCounter++;
       return objFunctValue;
}
```

```
public int[] localOptimumSearch(int[] givenSolution)
{
       float currentObjFunctValue, bestObjFunctValue, newObjValue;
       float[] trackBestSolution = new float[localNeighborSize];
       int[] trackBestSolutionIndex = new int [localNeighborSize];
       currentSolution = (int[])givenSolution.clone();
       localOptimumSolution = (int[])givenSolution.clone();

       currentObjFunctValue = computeObjValue(currentSolution);
       bestObjFunctValue = currentObjFunctValue;

       for(int i=0; i<localNoOfIterations; i++)
       {
       candidateList((int[])currentSolution.clone()); //Generate
       neighborhood for the current Solution

       //Compute and track the objective function for each feasible
       solution in the neighborhood
       for(int j=0; j<localNeighborSize; j++)
       {
       int[] newCurrentSolution = new int[givenSolution.length];
       trackBestSolutionIndex[j] = j;
       for(int k=0; k<givenSolution.length; k++)
       {
       newCurrentSolution[k] = neighborList[j][k];
       }
       trackBestSolution[j] = computeObjValue(newCurrentSolution);
       }
       //Re-arrange the neighborhood entries in the descending order of
       objective function value.
       for(int m=0; m<localNeighborSize-1; m++)
       {
       for(int n=0; n<localNeighborSize-1-m; n++)
       {
       if(trackBestSolution[n] < trackBestSolution[n+1])
       {
       float temp1 = trackBestSolution[n];
       trackBestSolution[n] = trackBestSolution[n+1];
       trackBestSolution[n+1] = temp1;
       int temp2 = trackBestSolutionIndex[n];
       trackBestSolutionIndex[n] = trackBestSolutionIndex[n+1];
       trackBestSolutionIndex[n+1] = temp2;
       }
       }

       }

       if(trackBestSolution[localNeighborSize-1] < bestObjFunctValue)
       {
       // Set the admissible solution with the best fitness value as
       current and best solutions.
       for(int r=0; r<currentSolution.length; r++)
       {
       currentSolution[r] =
       neighborList[trackBestSolutionIndex[localNeighborSize-1]][r];
       localOptimumSolution[r] =
       neighborList[trackBestSolutionIndex[localNeighborSize-1]][r];
       }
       // Set the fitness value for this admissible solution as the best
       value
       bestObjFunctValue = trackBestSolution[localNeighborSize-1];
       currentObjFunctValue = trackBestSolution[localNeighborSize-1];
```

```
        tabuObj.addToTabu((int[])currentSolution.clone(),
        trackBestSolution[localNeighborSize-1]);
        }

        else
        {
        boolean bestFound = false;
        boolean tabuChecker;
        int[] tabuAdmissibleSolution = new int [localSolutionSize];
        int bestCounter = localNeighborSize-1;
        while(!bestFound && bestCounter>=0)
        {
        for(int r=0; r<localSolutionSize; r++)
        tabuAdmissibleSolution[r] =
        neighborList[trackBestSolutionIndex[bestCounter]][r];
        tabuChecker =
        tabuObj.checkInTabu((int[])tabuAdmissibleSolution.clone());
        if(!tabuChecker)
        {
        currentSolution = (int[])tabuAdmissibleSolution.clone();
        currentObjFunctValue = trackBestSolution[bestCounter];
        tabuObj.addToTabu((int[])tabuAdmissibleSolution.clone(),
        trackBestSolution[bestCounter]);
        bestFound = true;
        }
        else
        bestCounter--;
        }
        }
        candidateList((int[])currentSolution.clone());
        }

        localOptimumValue = bestObjFunctValue;
        return (int[])localOptimumSolution.clone();
}


public int[][] localOptimumSearch(int[][] givenSolutionSet, float[]
givenSolutionSetValues)
{
float currentObjFunctValue, bestObjFunctValue, newObjValue;
float[] trackBestSolution = new float[localNeighborSize];
int[] trackBestSolutionIndex = new int [localNeighborSize];
eliteSolution eliteSolutionObj = new
eliteSolution((int[][])givenSolutionSet.clone(),
(float[])givenSolutionSetValues.clone());
optimalEliteSolutionSet =
(int[][])eliteSolutionObj.eliteSolutionSet.clone();
optimalEliteSolutionSetValues =
(float[])eliteSolutionObj.eliteSolutionValues.clone();
for(int q=0; q<givenSolutionSet.length; q++)
{
currentSolution = (int[])givenSolutionSet[q].clone();
localOptimumSolution = (int[])givenSolutionSet[q].clone();
boolean feasibleSolution = false;
currentObjFunctValue = givenSolutionSetValues[q];
//computeObjValue(currentSolution);
bestObjFunctValue = currentObjFunctValue;
for(int i=0; i<localNoOfIterations; i++)
{
candidateList((int[])currentSolution.clone()); //Generate neighborhood
for the current Solution
```

```
//Compute and track the objective function for each feasible solution in
the neighborhood
for(int j=0; j<localNeighborSize; j++)
{
int[] newCurrentSolution = new int[givenSolutionSet[0].length];
trackBestSolutionIndex[j] = j;
for(int k=0; k<givenSolutionSet[0].length; k++)
{
newCurrentSolution[k] = neighborList[j][k];
}
trackBestSolution[j] = computeObjValue(newCurrentSolution);
}
//Re-arrange the neighborhood entries in the descending order of
objective function value.
for(int m=0; m<localNeighborSize-1; m++)
{
for(int n=0; n<localNeighborSize-1-m; n++)
{
if(trackBestSolution[n] < trackBestSolution[n+1])
{
float temp1 = trackBestSolution[n];
trackBestSolution[n] = trackBestSolution[n+1];
trackBestSolution[n+1] = temp1;
int temp2 = trackBestSolutionIndex[n];
trackBestSolutionIndex[n] = trackBestSolutionIndex[n+1];
trackBestSolutionIndex[n+1] = temp2;
}
}
}
if(trackBestSolution[localNeighborSize-1] < bestObjFunctValue)
{
// Set the admissible solution with the best fitness value as current and
best solutions.
for(int r=0; r<currentSolution.length; r++)
{
currentSolution[r] =
neighborList[trackBestSolutionIndex[localNeighborSize-1]][r];
localOptimumSolution[r] =
neighborList[trackBestSolutionIndex[localNeighborSize-1]][r];
}

// Set the fitness value for this admissible solution as the best value
bestObjFunctValue = trackBestSolution[localNeighborSize-1];
currentObjFunctValue = trackBestSolution[localNeighborSize-1];
// add to tabu list
tabuObj.addToTabu((int[])currentSolution.clone(),
trackBestSolution[localNeighborSize-1]);
// add to elite solution list
eliteSolutionObj.addToEliteSet((int[])localOptimumSolution.clone(),
bestObjFunctValue);

}
else
{
boolean bestFound = false;
boolean tabuChecker;
int[] tabuAdmissibleSolution = new int [localSolutionSize];
int bestCounter = localNeighborSize-1;
while(!bestFound && bestCounter>=0)
{
for(int r=0; r<localSolutionSize; r++)
tabuAdmissibleSolution[r] =
neighborList[trackBestSolutionIndex[bestCounter]][r];
```

```
        if(!(tabuObj.checkInTabu((int[])tabuAdmissibleSolution.clone())))
        {
        currentSolution = (int[])tabuAdmissibleSolution.clone();
        currentObjFunctValue = trackBestSolution[bestCounter];
        tabuObj.addToTabu((int[])tabuAdmissibleSolution.clone(),
        trackBestSolution[bestCounter]);
        bestFound = true;
        }
        else
        bestCounter--;
        }
        }
        candidateList((int[])currentSolution.clone());
        }
        }

        return (int[][])eliteSolutionObj.eliteSolutionSet.clone();
        }

        //generate cadidate list
        public void candidateList(int[] currentLocalSolution)
        {
        neighborList =
        (int[][])(neighborObj.generateNeighbors(currentLocalSolution)).clone();
        }

        //find neighborhood best
        public float evaluateSolution(int[] currentSolution, boolean OptimumType)
        {
        neighborList =
        (int[][])(neighborObj.generateNeighbors(currentSolution)).clone();
        return 0.0f; //return objfuntn value of local optimum.
        }
}

-------------------------------End of localOptimum.java-----------------------


/**..........................................................
 * matrix.java
 * This class creates the matrix object and enables manipulations of matrices
and vectors
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
 */

public class matrix
{
        //Declare variables.
        public int resultType;
        public float[] sumVectorResult;
        public float[][] inverse;

        //Empty matrix class constructor
        public matrix()
        {

        }

        // Method to multiply two matrices.
        public float[][] matrixMultiply(float[][] firstMatrix, float[][]
        secondMatrix)
```

```
{
int firstMatrixRow = firstMatrix.length;
int firstMatrixColumn = firstMatrix[0].length;
int secondMatrixRow = secondMatrix.length;
int secondMatrixColumn = secondMatrix[0].length;
float[][] tempMatrix1 = (float[][])firstMatrix.clone();
float[][] tempMatrix2 = (float[][])secondMatrix.clone();
float[][] newMatrix = new float[firstMatrixColumn][secondMatrixRow];

if (firstMatrixColumn!=secondMatrixRow)
return null;
else
{
for(int i= 0; i<firstMatrixRow; i++)
{
for(int j=0; j<secondMatrixColumn; j++)
{
for(int k=0; k<firstMatrixColumn; k++)
{
newMatrix[i][j] = (tempMatrix1[i][k] * tempMatrix2[k][j]) +
newMatrix[i][j];
}
}
}
}
return (float[][])newMatrix.clone();
}

// Method multiplies a matrix by a scalar and returns the new matrix
public float[][] scalarMultiply(float[][] givenMatrix, float
scalarMultiplier)
{
      float scalarFactor = scalarMultiplier;
      int matrixRow = givenMatrix.length;
      int matrixColumn = givenMatrix[0].length;
      float[][] newScaledMatrix;
      newScaledMatrix = new float[matrixRow][matrixColumn];
      for(int i=0; i<matrixRow; i++)
      {
      for(int j=0; j<matrixColumn; j++)
      {
      newScaledMatrix[i][j] = givenMatrix[i][j] * scalarFactor;
      }
      }
      return (float[][])newScaledMatrix.clone();
}
public float[] scalarVectorMultiply(float multiplier, float[]
givenVector)
{
float[] newVector = new float[givenVector.length];
for(int i=0; i<givenVector.length; i++)
newVector[i] = givenVector[i]*multiplier;
return (float[])newVector.clone();
}

// Method adds two matrices and returns the new matrix;
public float[][] addMatrix(float[][] firstMatrix, float[][] secondMatrix)
{
      int firstMatrixRow = firstMatrix.length;
      int firstMatrixColumn = firstMatrix[0].length;
      int secondMatrixRow = secondMatrix.length;
      int secondMatrixColumn = secondMatrix[0].length;
```

```
            float[][] additionMatrix = new
            float[firstMatrixRow][firstMatrixColumn];
            if(firstMatrixRow!=secondMatrixRow ||
            firstMatrixColumn!=secondMatrixColumn)
            return null;
            for(int i=0; i<firstMatrixRow; i++)
            {
            for(int j=0; j<firstMatrixColumn; j++)
            {
            additionMatrix[i][j] = firstMatrix[i][j] + secondMatrix[i][j];
            }
            }
            return (float[][])additionMatrix.clone();
    }

    //Method adds two vectors and return a new vector
    public float[] addVector(float[] firstVector, float[] secondVector)
    {
            int firstVectorLength = firstVector.length;
            int secondVectorLength = secondVector.length;
            float[] additionVector = new float[firstVectorLength];
            if(firstVectorLength!=secondVectorLength)
                    return null;

            for(int j=0; j<firstVectorLength; j++)
            {
                    additionVector[j] = firstVector[j] + secondVector[j];
            }

            return (float[])additionVector.clone();

    }

    //Method subtracts a matrix from another matrix and returns a new matrix
    public float[][] subtractMatrix(float[][] firstMatrix, float[][]
    secondMatrix)
    {
            int firstMatrixRow = firstMatrix.length;
            int firstMatrixColumn = firstMatrix[0].length;
            int secondMatrixRow = secondMatrix.length;
            int secondMatrixColumn = secondMatrix[0].length;
            float[][] subtractMatrix = new float
            [firstMatrixRow][firstMatrixColumn];
            if(firstMatrixRow!=secondMatrixRow ||
            firstMatrixColumn!=secondMatrixColumn)
            return null;
            for(int i=0; i<firstMatrixRow; i++)
            {
                    for(int j=0; j<firstMatrixColumn; j++)
                    {
                            subtractMatrix[i][j] = firstMatrix[i][j] -
                            secondMatrix[i][j];
                    }
            }

            return (float[][])subtractMatrix.clone();
    }

    //Method subtracts a vector from another vector and returns a new vector
    public float[] subtractVector(float[] firstVector, float[] secondVector)
    {
            int firstVectorLength = firstVector.length;
            int secondVectorLength = secondVector.length;
```

```
        float[] subtractVector = new float[firstVectorLength];

        if(firstVectorLength!=secondVectorLength)
              return null;
        for(int j=0; j<firstVectorLength; j++)
        {
              subtractVector[j] = firstVector[j] - secondVector[j];
        }
        return (float[])subtractVector.clone();

  }

  public float columnVectorMultiply()
  {
        return 0;
  }

  //Method creates the transpose of the given matrix and return same
  public float[][] transposeMatrix(float[][] givenMatrix)
  {
        int matrixRow = givenMatrix.length;
        int matrixColumn = givenMatrix[0].length;
        float tempEntry =0;
        float[][] tempMatrix = new float[matrixRow][matrixColumn];
        float[][] transposeMatrix = new float[matrixColumn][matrixRow];
        for(int i=0; i<matrixColumn; i++)
        {
              for(int j=0; j<matrixRow; j++)
              {
                    transposeMatrix[i][j] = givenMatrix[j][i];
              }
        }
        return (float[][])transposeMatrix.clone();
  }

  //Method multiplies a matrix by a vector and returns a vector
  public float[] vectorMatrixMultiply(float[] givenVector,  float[][]
  givenMatrix)
  {
        int vectorLength = givenVector.length;
        int matrixRow = givenMatrix.length;
        int matrixColumn = givenMatrix[0].length;

        float[] newVector = new float[matrixColumn];

        if (vectorLength!=matrixRow)
        {
              return null;
        }
        else
        {
              for(int j=0; j<matrixColumn; j++)
              {
              for(int k=0; k<vectorLength; k++)
              {
              newVector[j] = (givenVector[k] * givenMatrix[k][j]) +
              newVector[j];
              }
              }
        }

        return (float[])newVector.clone();
  }
```

```
//Method multiplies a vector by a matrix and returns a vector
public float[] matrixVectorMultiply(float[][] givenMatrix, float[]
givenVector)
{
      int vectorLength = givenVector.length;
int matrixRow = givenMatrix.length;
int matrixColumn = givenMatrix[0].length;
float[] newVector = new float[matrixColumn];
if (vectorLength!=matrixColumn)
{
return null;
}
else
{
for(int j=0; j<matrixRow; j++)
{
for(int k=0; k<vectorLength; k++)
{
newVector[j] = (givenVector[k] * givenMatrix[j][k]) + newVector[j];
}
}
}
return (float[])newVector.clone();
}

//Method multiplies a vector by a vector and returns a number
public float vectorVectorMultiply (float[] firstVector, float[]
secondVector)
{
      int firstVectorLength = firstVector.length;
      int secondVectorLength = secondVector.length;
      float multiplyValue =0.0f;
      if (firstVectorLength!=secondVectorLength)
            return 0.0f;
      else
      {
      for(int k=0; k<firstVectorLength; k++)
      {
      multiplyValue = (firstVector[k] * secondVector[k]) +
      multiplyValue;
      }
      }
      return multiplyValue;
}

public float getElement()
{
      return 0.0f;
}

public int returnRow(float[][] givenMatrix)
{
      return 0;
}

public int returnColumn()
{
      return 0;
}

public float[][] inverseMatrix(float[][] givenMatrix)
{
```

```
             int matrixDimension = givenMatrix.length;
             float[][] tempMatrix0 = new
             float[matrixDimension][matrixDimension];
             for(int i=0; i<matrixDimension; i++)
             for(int j=0; j<matrixDimension; j++)
             tempMatrix0[i][j] = givenMatrix[i][j];

             inverse = new float[matrixDimension][matrixDimension];
             float[] tempMatrix = new float[matrixDimension];
             float pivotElement, factor;
             for(int i=0; i<matrixDimension; i++)
             {
             for(int j=0; j<matrixDimension; j++)
             {
             if(i==j)
             inverse[i][j] = 1.0f;
             else
             inverse[i][j] = 0.0f;
             }
             }
             for(int i=0; i<matrixDimension; i++)
             {
             for(int j=i+1; j<matrixDimension; j++)
             {
             if(Math.abs(tempMatrix0[i][i]) < Math.abs(tempMatrix0[j][i]))
             {
             for(int k =0; k<matrixDimension; k++)
             {
             tempMatrix[k] = tempMatrix0[i][k];
             tempMatrix0[i][k] = tempMatrix0[j][k];
             tempMatrix0[j][k] = tempMatrix[k];
             tempMatrix[k] = inverse[i][k];
             inverse[i][k] = inverse[j][k];
             inverse[j][k] = tempMatrix[k];
             }
       }
       }
  }
  for(int i=0; i<matrixDimension; i++)
  {
pivotElement = tempMatrix0[i][i];
for(int j=matrixDimension-1; j>=0; j--)
{
tempMatrix0[i][j] = tempMatrix0[i][j]/pivotElement;
inverse[i][j] = inverse[i][j]/pivotElement;
}
for(int j=i+1; j<matrixDimension; j++)
{
factor = -1 *  tempMatrix0[j][i];
for(int k=0; k<matrixDimension; k++)
{
tempMatrix0[j][k] = tempMatrix0[j][k] + (tempMatrix0[i][k]* factor);
inverse[j][k] = inverse[j][k] + (inverse[i][k]* factor);
}
}
}
for(int i=matrixDimension-1; i>=1; i--)
{
for(int j=i-1; j>=0; j--)
{
factor = -1 * tempMatrix0[j][i];
for(int k=0; k<matrixDimension; k++)
{
```

```
tempMatrix0[j][k] = tempMatrix0[j][k] + tempMatrix0[i][k]*factor;
inverse[j][k] = inverse[j][k] + inverse[i][k]*factor;
}
}
}
return (float[][])inverse.clone();
}
}

--------------------------- End of matrix.java-------------------------------


/**...............................................................
 * neighbourhood.java
 * This class generates the neighborhood set for a given solution
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
 */

public class neighborhood
{
       public int startDimension, endDimension;
       public int neighborLength;
       public int solutionLength;
       public int noOfReplaceables;
       public int totalNoOfFeatures;
       public int[] removeIndex;
       public int[] replaceIndex;
       public int[][] neighborCandidateList;
       public int[] replaceableSolution;
       public int[] newSolutionSpace;
       public boolean newSolutionSpaceGiven = false;
       int[] randomGenerator;
       int[] tempRandomGenerator;

       public neighborhood(parameter paramObj)
       {
       startDimension = paramObj.startDimension;
       endDimension = paramObj.endDimension;
       noOfReplaceables = paramObj.hammingDistance;
       solutionLength = paramObj.solutionSize;
       neighborLength = paramObj.neighborhoodSize;
       totalNoOfFeatures = endDimension - startDimension + 1;
       replaceableSolution = new int [noOfReplaceables];
       neighborCandidateList = new int[neighborLength][solutionLength];
       for(int i=0; i<noOfReplaceables; i++)
       replaceableSolution[i]=0;

       newSolutionSpaceGiven = false;
       }

       public neighborhood(parameter paramObj, int[] givenSolutionSpace)
       {
       noOfReplaceables = paramObj.hammingDistance;
       solutionLength = paramObj.solutionSize;
       neighborLength = paramObj.neighborhoodSize;
       totalNoOfFeatures = givenSolutionSpace.length;

       replaceableSolution = new int [noOfReplaceables];
       neighborCandidateList = new int[neighborLength][solutionLength];
       newSolutionSpace = new int[solutionLength];
       for(int i=0; i<noOfReplaceables; i++)
             replaceableSolution[i]=0;
```

```
newSolutionSpaceGiven = true;
newSolutionSpace = (int[])givenSolutionSpace.clone();

}

public int[][] generateNeighbors(int[] argSolution)
{
int[] updatedSolution1 = new int[solutionLength];
int[] tempSolutionList = new int[solutionLength];
int[] tempSolutionList2 = new int[solutionLength];
int[] localGivenSolution = (int[])argSolution.clone();
tempSolutionList2 = (int[])argSolution.clone();
for(int m=0; m<neighborLength; m++)
{
tempSolutionList = (int[])argSolution.clone();
updatedSolution1 = (int[])argSolution.clone();
//randomly re-arrange the solution items in tempSolutionList
for(int i=0; i<solutionLength; i++)
{
int random1 = (int)(((solutionLength - i)* Math.random()) + i);
int temp1 = tempSolutionList[random1];
tempSolutionList[random1]=tempSolutionList[i];
tempSolutionList[i] = temp1;
}
//Select one of the re-arranged entries as the replaceable item.
for(int k=0;k<noOfReplaceables; k++)
{
replaceableSolution[k] = tempSolutionList[k];
}

// generate replacing solution for the replaceable item by creating a
list of items that are not in the present solution.
tempRandomGenerator = new int[totalNoOfFeatures - solutionLength];
int tempSolutionCounter = 0;
boolean solutionFlag;
for(int i=0; i<totalNoOfFeatures; i++)
{
//check if the ith feature is in the current solution feature subset.
solutionFlag = true;
for(int j=0; j<solutionLength; j++)
{
if(newSolutionSpaceGiven==true)
{
if(newSolutionSpace[i]==localGivenSolution[j])
{
solutionFlag=false;
}
}
else
{
if((i+startDimension)==localGivenSolution[j])
{
solutionFlag=false;
}
}
}
if(solutionFlag==true)
{
if(newSolutionSpaceGiven==true)
{
tempRandomGenerator[tempSolutionCounter] = newSolutionSpace[i];
}
else
```

```
        tempRandomGenerator[tempSolutionCounter] = i+startDimension;
        tempSolutionCounter++;
        solutionFlag = true;
        }
        }
        //Randomly re-arrange the features that are not in the present solution
        feature subset.
        int random2;
        for(int i=0; i<(totalNoOfFeatures-solutionLength); i++)
        {
        random2 = (int)(((totalNoOfFeatures-solutionLength-i)* Math.random()) +
        i);
        int temp2 = tempRandomGenerator[random2];
        tempRandomGenerator[random2]=tempRandomGenerator[i];
        tempRandomGenerator[i] = temp2;
        }
        int counter2 =0;
        boolean replaceFlag, neighborFlag=false;
        for(int i=0; i<solutionLength; i++)
        {
        replaceFlag = true;
        for(int j=0; j<noOfReplaceables; j++)
        {
        if(updatedSolution1[i] == replaceableSolution[j])
        {
        replaceFlag=false;
        neighborCandidateList[m][i] = tempRandomGenerator[counter2];
        counter2++;
        break;
        }
        }
        if(replaceFlag==true)
        neighborCandidateList[m][i] = updatedSolution1[i];
        if(neighborCandidateList[m][i]==0)
        neighborFlag = true;
        }
        if(neighborFlag==true)
        {
        System.out.println("Neighborhood of a Zero entry");
        for(int i=0; i<solutionLength; i++)
        {
        System.out.print(neighborCandidateList[m][i]+"\t");
        }
        System.out.println();
        System.out.println("Tempgenerator is: ");
        for(int j=0; j<totalNoOfFeatures - solutionLength; j++)
        {
        System.out.println(tempRandomGenerator[j]);
        }
        System.out.println();
        }
        }
        return (int[][])neighborCandidateList.clone();
        }
}

---------------------------------End of neighbourhood.java--------------------


/**...............................................................
 * objectiveFunction.java
 * This class computes all objective function-related values
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
 */
public class objectiveFunction
{

// Declare variables.
public int[] validatingTrainSet; //Complete training set index
public int[] validatingClassLabel; //Complete dataset class labels
public float[][] validatingDataSet; //Complete training set dataset
matrix matrixInstance;     //Create an instance of the matrix class for matrix
                           and vector manipulation
parameter localParamObj;
int[] localTempFeatureSpace;

public objectiveFunction(parameter paramObj)
{
       localParamObj = paramObj;
       matrixInstance = new matrix();
       validatingTrainSet = (int[])paramObj.globalTrainDataIndex.clone();
       validatingClassLabel = (int[])paramObj.globalClassLabel.clone();
       if(!paramObj.tempDataFlag)
       validatingDataSet = (float[][])paramObj.globalDataSet.clone();
       else
       {
       validatingDataSet = (float[][])paramObj.tempGlobalDataset.clone();
       localTempFeatureSpace = (int[])paramObj.globalTempLevelSubspace.clone();
       }
}

public objectiveFunction(parameter paramObj, float[][] givenDataset)
{
       localParamObj = paramObj;
       matrixInstance = new matrix();
       validatingTrainSet = (int[])paramObj.globalTrainDataIndex.clone();
       validatingClassLabel = (int[])paramObj.globalClassLabel.clone();
       validatingDataSet = (float[][])givenDataset.clone();
}

public float[][] computeSpooledCovariance(float[][] firstCovariance, int
sampleSize1, float[][] secondCovariance, int sampleSize2)
{
       int scalarFactor1 = sampleSize1 - 1;
       int scalarFactor2 = sampleSize2 - 1;
       int scalarFactor3 = sampleSize1 + sampleSize2 - 2;
       float[][] tempCovariance1 =
       (float[][])matrixInstance.scalarMultiply(firstCovariance,
       scalarFactor1).clone();
       float[][] tempCovariance2 =
       (float[][])matrixInstance.scalarMultiply(secondCovariance,
       scalarFactor2).clone();
       float[][] tempCovariance3 =
       (float[][])matrixInstance.addMatrix(tempCovariance1,
       tempCovariance2).clone();
       float[][] spooledCovariance =
       (float[][])matrixInstance.scalarMultiply(tempCovariance3,
       1.0f/scalarFactor3).clone();
       return (float[][])spooledCovariance.clone();
}

public float[][] computeInverseSpooledCovariance(float[][] firstCovariance, int
sampleSize1, float[][] secondCovariance, int sampleSize2)
{
        float[][] localSpooledCovariance =
        (float[][])computeSpooledCovariance(firstCovariance, sampleSize1,
        secondCovariance, sampleSize2).clone();
        float[][] inverseSpooledCovariance =
        (float[][])matrixInstance.inverseMatrix(localSpooledCovariance);
return (float[][])inverseSpooledCovariance.clone();
}

public float computeMahalanobis(float[][] firstCovariance, int sampleSize1,
float[][] secondCovariance, int sampleSize2, int dimension, float[]
firstMeanVector, float[] secondMeanVector)
{
        float[][] localInverseSpooledCovariance =
        (float[][])computeInverseSpooledCovariance(firstCovariance, sampleSize1,
        secondCovariance, sampleSize2);
        System.out.println("Inverse Spooled covariance");
        for(int i=0; i<localInverseSpooledCovariance.length; i++)
        {
        System.out.println("\n");
        for(int j=0; j<localInverseSpooledCovariance[0].length; j++)
        {
                System.out.print(localInverseSpooledCovariance[i][j]+"\t");
        }
        }
float[] meanVectorDifference =
(float[])matrixInstance.subtractVector(firstMeanVector,
secondMeanVector).clone();
for(int i=0; i<meanVectorDifference.length; i++)
System.out.print(meanVectorDifference[i]+"\t");
float[] tempMultiply1 =
(float[])matrixInstance.vectorMatrixMultiply(meanVectorDifference,
localInverseSpooledCovariance).clone();
for(int i=0; i<tempMultiply1.length; i++)
System.out.print(tempMultiply1[i]+"\t");
float tempMultiply2 = matrixInstance.vectorVectorMultiply(meanVectorDifference,
tempMultiply1);
Double doubleObj = new Double(Math.sqrt(tempMultiply2));
float mDistanceValue = doubleObj.floatValue();
if(mDistanceValue>=0.0f)
return mDistanceValue;
else
return 0.0f;
}

public float[] computeMeanVector(float [][]dataValues)
{
        int rowLength = dataValues.length;
        int colLength = dataValues[0].length;
        float sumMeanValue;
        float[] meanVector;
        float[][] datasetValues;
        datasetValues = new float [rowLength][colLength];
        meanVector = new float [colLength];
        for(int i=0; i<rowLength; i++)
        {
        for(int j=0; j<colLength; j++)
        {
        datasetValues[i][j] = dataValues[i][j];
        }
        }
        for(int i = 0; i<colLength; i++)
        {
        sumMeanValue = 0.0f;
        for(int j=0; j<rowLength; j++)
        {
        sumMeanValue = sumMeanValue + datasetValues[j][i];
        }
        meanVector[i] = sumMeanValue/rowLength;
        }

return (float[])meanVector.clone();
}

public float[][] computeCovariance(float[][] dataValues, float[]
meanVectorValues)
{
        int rowLength = dataValues.length;
        int colLength = dataValues[0].length;
        float[] meanVector;
        float[] referenceColumnVector;
        float[] columnVector;
        float[][] datasetValues;
        float[][] covarianceMatrix;
        float covarianceMatrixValue=0.0f;
        datasetValues = new float [rowLength][colLength];
        meanVector = new float [colLength];
        covarianceMatrix = new float[colLength][colLength];
        referenceColumnVector = new float [rowLength];
        columnVector = new float [rowLength];
        datasetValues = dataValues;
        meanVector = (float[])meanVectorValues.clone();
        for(int i=0; i<rowLength; i++)
        {
        for(int j=0; j<colLength; j++)
        {
        datasetValues[i][j] = dataValues[i][j];
        }
        }
        for(int i=0; i<colLength; i++)
        {
        for(int j=i; j<colLength; j++)
        {
        if(i==j)
        {
        for(int k=0; k<rowLength; k++)
        {
        referenceColumnVector[k]=datasetValues[k][j]-meanVector[j];
        }
        }
        else
        {
        for(int k=0; k<rowLength; k++)
        {
        columnVector[k]=datasetValues[k][j]-meanVector[j];
        }
        }
        for(int n=0; n<rowLength; n++)
        {
        if(i==j)
        covarianceMatrixValue = referenceColumnVector[n]*referenceColumnVector[n]
        + covarianceMatrixValue;
        else
        covarianceMatrixValue = referenceColumnVector[n]*columnVector[n] +
        covarianceMatrixValue;
        }
        covarianceMatrix[i][j] = covarianceMatrixValue/(rowLength-1);
        covarianceMatrix[j][i] = covarianceMatrix[i][j];
        covarianceMatrixValue =0.0f;
        }
        }
        return (float[][])covarianceMatrix.clone();
        }

public float meanSquareError(int[] currentSolution)
{
        int[] localCurrentSolution = (int[])currentSolution.clone();
        float meanSquareError=0.0f;
        float validateReference;
        float[] errorValues = new float[validatingTrainSet.length];
        int[] tempTrainSet = new int[validatingTrainSet.length-1];
        int[] completeTrainset = (int[])validatingTrainSet.clone();
        float[][] validateDataSetA;
        float[][] validateDataSetB;
        float[][] validateCovarianceA;
        float[][] validateCovarianceB;
        float[][] validateInverseCovariance;
        float[] validateMeanA;
        float[] validateMeanB;
        float[] validateSumMeanVector;
        float[] validateDiffMeanVector;
        float[] validateConstantVector;
        float[] testValidateVector = new float[localCurrentSolution.length];
        int classLabelIndicator;
        int validateClassLabelIndicator;
        int completeCounterA = 0;
        int completeCounterB = 0;
        // Compute whole-sample covariance inverse.
        for(int j=0; j<completeTrainset.length; j++)
        {
        if(validatingClassLabel[completeTrainset[j]]==1)
        completeCounterA++;
        else if(validatingClassLabel[completeTrainset[j]]==2)
        completeCounterB++;
        }
        float[][] completeDataSetA = new
        float[completeCounterA][currentSolution.length];
        float[][] completeDataSetB = new
        float[completeCounterB][currentSolution.length];
        int completeCounter1=0;
        int completeCounter2=0;
        int totalCompleteCounter;
        for(int j=0; j<completeTrainset.length; j++)
        {
        if(validatingClassLabel[completeTrainset[j]]==1)
        {
        for(int k=0; k<currentSolution.length; k++)
        {
        if(!localParamObj.tempDataFlag)
        completeDataSetA[completeCounter1][k] =
        validatingDataSet[completeTrainset[j]][localCurrentSolution[k]-1];
        else
        {
        for(int m=0; m<localTempFeatureSpace.length; m++)
        {
        if(localCurrentSolution[k]==localTempFeatureSpace[m])
        {
        completeDataSetA[completeCounter1][k] =
        validatingDataSet[completeTrainset[j]][m];
        break;
}
}
}
}
completeCounter1++;
}
else if(validatingClassLabel[completeTrainset[j]]==2)
{
for(int k=0; k<currentSolution.length; k++)
if(!localParamObj.tempDataFlag)
completeDataSetB[completeCounter2][k] =
validatingDataSet[completeTrainset[j]][localCurrentSolution[k]-1];
else
{
for(int m=0; m<localTempFeatureSpace.length; m++)
{
if(localCurrentSolution[k]==localTempFeatureSpace[m])
{
completeDataSetB[completeCounter2][k] =
validatingDataSet[completeTrainset[j]][m];
break;
}
}
}
completeCounter2++;
}
}
// Total number of samples in both classes
totalCompleteCounter = completeCounter1 + completeCounter2;
// Mean vector of class A using all the samples
float[] completeMeanA =
(float[])computeMeanVector(completeDataSetA).clone();

// Mean vector of class B using all the samples
float[] completeMeanB =
(float[])computeMeanVector(completeDataSetB).clone();

// Sum of mean vectors for class A and B (i.e. y in the reference text)
float[] completeMeanSum =
(float[])matrixInstance.addVector(completeMeanA, completeMeanB); //i.e. Y

// Difference of mean vectors for class A and B (i.e. z in the reference
text)
float[] completeMeanDifference =
(float[])matrixInstance.subtractVector(completeMeanA, completeMeanB);
//i.e. Z

// Class A covariance using all the samples
float[][] completeCovarianceA =
(float[][])computeCovariance(completeDataSetA, completeMeanA).clone();

// Class B covariance using all the samples
float[][] completeCovarianceB =
(float[][])computeCovariance(completeDataSetB, completeMeanB).clone();

// inverse spooled covariance using all the samples
float[][] completeInverseSpooled =
(float[][])computeInverseSpooledCovariance(completeCovarianceA,
completeDataSetA.length, completeCovarianceB,
completeDataSetB.length).clone();

// Estimate train classifier parameters for cross-validation
for(int i=0; i<validatingTrainSet.length; i++)
```

```
{
classLabelIndicator = validatingClassLabel[validatingTrainSet[i]];
int counter =0;
int counterA = 0;
int counterB = 0;
int classCounter = 0;
float constantCK =0.0f;
float[] adjustedMeanVector = new float[currentSolution.length];
float[][] inverseSpooledCovaraianceWithout;
float[] excludedVector = new float[currentSolution.length]; // i.e. x(i)

// Extract the excluded vector, i.e. x(i)
for(int k=0; k<currentSolution.length; k++)
excludedVector[k] = validatingDataSet[validatingTrainSet[i]][k];

// Training set without the excluded vector
for(int j=0; j<validatingTrainSet.length; j++)
{
if(i!=j)
{
tempTrainSet[counter] = validatingTrainSet[j];
counter++;
}
}

for(int j=0; j<tempTrainSet.length; j++)
{
if(validatingClassLabel[tempTrainSet[j]]==1)
counterA++;
else if(validatingClassLabel[tempTrainSet[j]]==2)
counterB++;
}
//Compute the constantCK i.e. c(k) in the reference.
if(classLabelIndicator==1)
{
adjustedMeanVector =
(float[])matrixInstance.subtractVector(excludedVector, completeMeanA);
constantCK = completeCounter1/((completeCounter1-
1)*(totalCompleteCounter-2));
classCounter = completeCounter1;
}

else if(classLabelIndicator==2)
{
adjustedMeanVector =
(float[])matrixInstance.subtractVector(excludedVector, completeMeanB);
constantCK = completeCounter2/((completeCounter2-
1)*(totalCompleteCounter-2));
classCounter = completeCounter2;
}

//Compute the inverse of the spooled covariance without the excluded
vector.

float[] numeratorpProduct1 =
(float[])matrixInstance.vectorMatrixMultiply(adjustedMeanVector,
completeInverseSpooled).clone();
float numeratorpProduct2 =
matrixInstance.vectorVectorMultiply(adjustedMeanVector,
numeratorpProduct1);
float[] denorminatorpProduct1 =
(float[])matrixInstance.matrixVectorMultiply(completeInverseSpooled,
adjustedMeanVector);
```

```
float denorminatorpProduct2 =
matrixInstance.vectorVectorMultiply(denorminatorpProduct1,
adjustedMeanVector);
float multiplierConstant = (constantCK * numeratorpProduct2)/(1 -
constantCK * denorminatorpProduct2);
float[][] tempInverseSpooledCovaraianceWithout =
(float[][])matrixInstance.scalarMultiply(completeInverseSpooled,
multiplierConstant).clone();
validateInverseCovariance =
(float[][])matrixInstance.addMatrix(completeInverseSpooled,
tempInverseSpooledCovaraianceWithout);

validateDataSetA = new float[counterA][currentSolution.length];
validateDataSetB = new float[counterB][currentSolution.length];
int tempCounter1=0;
int tempCounter2=0;
for(int j=0; j<tempTrainSet.length; j++)
{
if(validatingClassLabel[tempTrainSet[j]]==1)
{
for(int k=0; k<currentSolution.length; k++)
{
if(!localParamObj.tempDataFlag)
{
validateDataSetA[tempCounter1][k] =
validatingDataSet[tempTrainSet[j]][localCurrentSolution[k]-1];
}
else
{
for(int m=0; m<localTempFeatureSpace.length; m++)
{
if(localCurrentSolution[k]==localTempFeatureSpace[m])
{
validateDataSetA[tempCounter1][k] =
validatingDataSet[tempTrainSet[j]][m];
break;
}
}
}
}
tempCounter1++;
}
else if(validatingClassLabel[tempTrainSet[j]]==2)
{
for(int k=0; k<currentSolution.length; k++)
{
if(!localParamObj.tempDataFlag)
{
validateDataSetB[tempCounter2][k] =
validatingDataSet[tempTrainSet[j]][localCurrentSolution[k]-1];
}
else
{
for(int m=0; m<localTempFeatureSpace.length; m++)
{
if(localCurrentSolution[k]==localTempFeatureSpace[m])
{
validateDataSetB[tempCounter2][k] =
validatingDataSet[tempTrainSet[j]][m];
break;
}
}
}
```

```
        }
        tempCounter2++;
        }
        }
        validateMeanA = (float[])computeMeanVector(validateDataSetA).clone();
        validateMeanB = (float[])computeMeanVector(validateDataSetB).clone();

        validateSumMeanVector = (float[])(matrixInstance.addVector(validateMeanA,
        validateMeanB)).clone();
        validateDiffMeanVector =
        (float[])(matrixInstance.subtractVector(validateMeanA,
        validateMeanB)).clone();
        validateConstantVector =
        (float[])matrixInstance.vectorMatrixMultiply(validateDiffMeanVector,
        validateInverseCovariance).clone();
        validateReference = 0.5f *
        matrixInstance.vectorVectorMultiply(validateConstantVector,
        validateSumMeanVector);

        for(int m=0; m<localCurrentSolution.length; m++)
        {
        if(!localParamObj.tempDataFlag)
        testValidateVector[m] =
        validatingDataSet[validatingTrainSet[i]][localCurrentSolution[m]-1];
        else
        {
        for(int n=0; n<localTempFeatureSpace.length; n++)
        {
        if(localCurrentSolution[m]==localTempFeatureSpace[n])
        {
        testValidateVector[m] = validatingDataSet[validatingTrainSet[i]][n];
        break;
        }
        }
        }
        }
        float testClassLabelValue =
        matrixInstance.vectorVectorMultiply(validateConstantVector,
        testValidateVector);
        if(matrixInstance.vectorVectorMultiply(validateConstantVector,
        testValidateVector)>=validateReference)
        validateClassLabelIndicator = 1;
        else
        validateClassLabelIndicator = 2;
        if(validateClassLabelIndicator!=classLabelIndicator)
        meanSquareError = meanSquareError + 1;
        }
        return meanSquareError/validatingTrainSet.length;
        }
}

------------------------- End of objectiveFunction.java-----------------------


/**.............................................................
 * parameter.java
 * This class sets and resets all global parameters for the multilevel and
basic tabu search alogrithms.
 * @author          Idowu Olayinka Oduntan
 * @version         August 2005
 */

public class parameter
```

```
{
      // Object initialization

      //Dataset parameters
      public String dataFile = "albi_vs_dubl";
      public int noOfSamples = 337;
      public int noOfTrainSamples = 202;
      int origNoOfSamples = 337;
      int origNoOfTrainSamples = 202;
      //public String dataFile = "albi_vs_para";
      //public int noOfSamples = 304;
      //public int noOfTrainSamples = 124;
      //int origNoOfSamples = 304;
      //int origNoOfTrainSamples = 124;
      public String delimiterChar = " []=;:\t ";
      public String datasetType = "train";
      public String optimumType = "MIN";
      public int noOfHeaderLines = 3;
      public int noOfHeaderParameters = 4;
      public int noOfClasses = 2;
      public int noOfDimension = 1500;
      int origNoOfHeaderLines = 3;
      int origNoOfHeaderParameters = 4;
      int origNoOfClasses = 2;
      int origNoOfDimension = 1500;
      String origDatasetType = "train";

      //Dataset variables
      public int[] globalTrainDataIndex;
      public int[] globalTestDataIndex;
      public int[] globalTrainDataIndexA;
      public int[] globalTrainDataIndexB;
      public int[] globalTestDataIndexA;
      public int[] globalTestDataIndexB;
      public int[] globalClassLabel;

      public float[][] globalDataSet;
      public float[][] globalTrainClassADataSet;
      public float[][] globalTrainClassBDataSet;
      public float[][] globalTestClassADataSet;
      public float[][] globalTestClassBDataSet;
      public float[][] tempGlobalDataset;
      public int[] globalTempLevelSubspace;
      public int[] globalTempLevelSubspaceWithComponents;
      public boolean tempDataFlag = false;

      //Search parameters
      public String searchType = "tabu" ;
      public int searchSpace;
      public int solutionSize = 10;
      public int defaultSolutionSize = 10;
      public int partialSolutionSize = 10;
      public int finalSolutionSize = 10;
      public int baseNoOfIterations = 10;
      public int noOfIterations = 25;
      public int tempNoOfIterations = 25;
      public int tabuListSize=1500/50; //750;
      public int neighborhoodSize = 1500/15;//1500;
      public int hammingDistance = 1;

      String origSearchType = "tabu" ;
      int origSearchSpace = 1500;
      int origSolutionSize = 10;
int origDefaultSolutionSize = 10;
int origPartialSolutionSize = 10;
int origFinalSolutionSize = 10;
int origBaseNoOfIterations = 10;
int origNoOfIterations = tempNoOfIterations;
int origTabuListSize = 1500/50; //750
int origNeighborhoodSize = 1500/15; //1500
int origHammingDistance = 1;

//Multilevel parameters
public int noOfPartitions = 3;
public int noOfLevels = 3;
public int reductionFactor = 2;
public int partitionSolutionSize = 30;
public int noOfEliteSolutions = 5;
public int clusterWindow = 2;

//String partitionType = "fixed";
String partitionType = "random";
int origNoOfPartitions = 3;
int origNoOfLevels = 3;
int origStartDimension = 1;
int origEndDimension = 1500;
int origReductionFactor = 2;

//Misc parameters
public String outputMedia = "file";
public String outputFileName = "outputFile";
public int noOfRuns = 10;
public int randomnessFactor = 5; //5;
public int startDimension = 1;
public int endDimension = 1500;

public parameter()
{

}
// Set dataset variables
public void setGlobalDataSet(float[][] dataSet)
{
      globalDataSet = (float[][])dataSet.clone();
}

public void setGlobalClassLabel(int[] classLabel)
{
      globalClassLabel = (int[])classLabel.clone();
}

// Set temp dataset variables
public void setTempGlobalDataset(int[] tempLevelSubspace, int[]
tempLevelWithComponents, float[][] tempDataset)
{
      tempGlobalDataset = (float[][])tempDataset.clone();
      globalTempLevelSubspace = (int[])tempLevelSubspace.clone();
      globalTempLevelSubspaceWithComponents =
      (int[])tempLevelWithComponents.clone();
      tempDataFlag = true;
}

// Set temp dataset variables
public void resetTempGlobalDataset()
{
      tempDataFlag = false;
}

// Set train dataset variables:
public void setGlobalTrainDataIndex(int[] trainDataIndex)
{
      globalTrainDataIndex = (int[])trainDataIndex.clone();
}

public void setGlobalTrainDataIndexA(int[] trainDataIndexA)
{
      globalTrainDataIndexA = (int[])trainDataIndexA.clone();
}

public void setGlobalTrainDataIndexB(int[] trainDataIndexB)
{
      globalTrainDataIndexB = (int[])trainDataIndexB.clone();
}

public void setGlobalTrainClassADataSet(float[][] trainClassADataSet)
{
      globalTrainClassADataSet = (float[][])trainClassADataSet.clone();
}

public void setGlobalTrainClassBDataSet(float[][] trainClassBDataSet)
{
      globalTrainClassBDataSet = (float[][])trainClassBDataSet.clone();
}


// Set test dataset variables:
public void setGlobalTestDataIndex(int[] testDataIndex)
{
      globalTestDataIndex = (int[])testDataIndex.clone();
}

public void setGlobalTestDataIndexA(int[] testDataIndexA)
{
      globalTestDataIndexA = (int[])testDataIndexA.clone();
}

public void setGlobalTestDataIndexB(int[] testDataIndexB)
{
      globalTestDataIndexB = (int[])testDataIndexB.clone();
}

public void setGlobalTestClassADataSet(float[][] testClassADataSet)
{
      globalTestClassADataSet = (float[][])testClassADataSet.clone();
}

public void setGlobalTestClassBDataSet(float[][] testClassBDataSet)
{
      globalTestClassBDataSet = (float[][])testClassBDataSet.clone();
}

/* Dataset parameters */
// Set the number of hearderlines
public void setNoOfHeaderLines(int newValue)
{
      noOfHeaderLines = newValue;
}
// Reset the number of hearderlines
public void resetNoOfHeaderLines()
```

```
{
        noOfHeaderLines = origNoOfHeaderLines;
}

// Set the number of hearder parameters
public void setNoOfHeaderParameters(int newValue)
{
        noOfHeaderParameters = newValue;
}
// Reset the number of hearder parameters
public void resetNoOfHeaderParameters()
{
        noOfHeaderParameters = origNoOfHeaderParameters;
}

// Set the number of samples
public void setNoOfSamples(int newValue)
{
        noOfSamples = newValue;
}
// Reset the number of samples
public void resetNoOfSamples()
{
        noOfSamples = origNoOfSamples;
}

// Set the number of train samples
public void setNoOfTrainSamples(int newValue)
{
        noOfTrainSamples = newValue;
}
// Reset the number of train samples
public void resetNoOfTrainSamples()
{
        noOfTrainSamples = origNoOfTrainSamples;
}

// Set the number of classes
public void setNoOfClasses(int newValue)
{
        noOfClasses = newValue;
}
// Reset the number of classes
public void resetNoOfClasses()
{
        noOfClasses = origNoOfClasses;
}

// Set the number of dimension
public void setNoOfDimension(int newValue)
{
        noOfDimension = newValue;
}
// Reset the number of dimension
public void resetNoOfDimension()
{
        noOfDimension = origNoOfDimension;
}

// Set the dataset type (i.e. 'train' or 'test' set)
public void setDatasetType(String newValue)
{
        datasetType = newValue;
}
// Reset the dataset type (i.e. 'train' or 'test' set)
public void resetDatasetType(String newValue)
{
      datasetType = origDatasetType;
}


/* Search parameter */
// Set the serach type
public void setSearchType(String newValue)
{
      searchType = newValue;
}
// Reset the serach type
public void resetSearchType()
{
      searchType = origSearchType;
}

// Set the search space size
public void setSearchSpace(int newValue)
{
      searchSpace = newValue;
}
// Reset the search space size
public void resetSearchSpace()
{
      searchSpace = origSearchSpace;
}

// Set the solution size
public void setSolutionSize(int newValue)
{
      solutionSize = newValue;
}
// Reset the solution size
public void resetSolutionSize()
{
      solutionSize = origSolutionSize;
}

// Set the default solution size
public void setDefaultSolutionSize(int newValue)
{
      defaultSolutionSize = newValue;
}
// Reset the default solution size
public void resetDefaultSolutionSize()
{
      defaultSolutionSize = origDefaultSolutionSize;
}

// Set the partial solution size
public void setPartialSolutionSize(int newValue)
{
      partialSolutionSize = newValue;
}
// Reset the partial solution size
public void resetPartialSolutionSize()
{
      partialSolutionSize = origPartialSolutionSize;
}
```

```
// Set the final solution size
public void setFinalSolutionSize(int newValue)
{
      finalSolutionSize = newValue;
}
// Reset the final solution size
public void resetFinalSolutionSize()
{
      finalSolutionSize = origFinalSolutionSize;
}

// Set the base number of iterations
public void setBaseNoOfIterations(int newValue)
{
      baseNoOfIterations = newValue;
}
// Reset the base number of iterations
public void resetBaseNoOfIterations()
{
      baseNoOfIterations = origBaseNoOfIterations;
}

// Set the number of iterations
public void setNoOfIterations(int newValue)
{
      noOfIterations = newValue;
}
// Reset the base number of iterations
public void resetNoOfIterations()
{
      noOfIterations = origNoOfIterations;
}

// Set tabu list size
public void setTabuListSize(int newValue)
{
      tabuListSize = newValue;
}
// Reset tabu list size
public void resetTabuListSize()
{
      tabuListSize = origTabuListSize;
}

// Set neighborhood size
public void setNeighborhoodSize(int newValue)
{
      neighborhoodSize = newValue;
}
// Reset neighborhood size
public void resetNeighborhoodSize()
{
      neighborhoodSize = origNeighborhoodSize;
}

// Set hamming distance value
public void setHammingDistance(int newValue)
{
      hammingDistance = newValue;
}
// Reset hamming distance value
public void resetHammingDistance()
```

```
        {
              hammingDistance = origHammingDistance;
        }

        // Set start value of dimension
        public void setStartDimension(int newValue)
        {
              startDimension = newValue;
        }
        // Reset start value of dimension
        public void resetStartDimension()
        {
              startDimension = origStartDimension;
        }

        // Set end value of dimension
        public void setEndDimension(int newValue)
        {
              endDimension = newValue;
        }
        // Reset end value of dimension

        public void resetEndDimension()
        {
              endDimension = origEndDimension;
        }


        /* Multilevel parameters */
        // Set the number of partitions at each level
        public void setNoOfPartitions(int newValue)
        {
              noOfPartitions = newValue;
        }
        // Reset the number of partitions at each level
        public void resetNoOfPartitions()
        {
              noOfPartitions = origNoOfPartitions;
        }

        // Set the number of levels in the multilevel hierarchy
        public void setNoOfLevels(int newValue)
        {
              noOfLevels = newValue;
        }
        // Reset the number of levels in the multilevel hierarchy
        public void resetNoOfLevels()
        {
              noOfLevels = origNoOfLevels;
        }

        //Set the reduction factor across levels in the multilevel hierarchy
        public void setReductionFactor(int newValue)
        {
              reductionFactor = newValue;
        }
        // Reset the number of levels in the multilevel hierarchy
        public void resetReductionFactor()
        {
              reductionFactor = origReductionFactor;
        }

}
```

```
--------------------------------End of parameter.java----------------------


/**............................................................
 * printClass.java
 * This class provides methods that handle all output print requirements.
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
 */
import java.io.PrintStream;
import java.io.FileOutputStream;
import java.io.IOException;
public class printClass
{
        PrintStream outputResult;
        parameter paramObj;
        String outputMedia, localOutputFile;
        public printClass()
        {
                paramObj = new parameter();
                localOutputFile = paramObj.outputFileName;
                outputMedia = paramObj.outputMedia;
                try
                {
                        outputResult = new PrintStream(new
        FileOutputStream(localOutputFile, true));
                }

                catch (Exception e0)
                {
                        System.out.println("Cannot print to output file...!");
                }
        }

        public void printHeader(String headerString)
        {
                if(outputMedia.equals("file"))
                {
                        outputResult.println();
                        outputResult.println(headerString);
                        outputResult.println();

                }
                else if(outputMedia.equals("file and screen"))
                {
                        outputResult.println();
                        outputResult.println(headerString);
                        outputResult.println();
                        System.out.println();
                        System.out.println(headerString);
                        System.out.println();
                }
                else
                {
                        System.out.println();
                        System.out.println(headerString);
                        System.out.println();
                }
        }

        public void printLiteral(String literal)
	{
		if(outputMedia.equals("file"))
		{
			outputResult.print(literal);
		}
		else
		{
			System.out.print(literal);
		}
	}

	public void printOutput(int[] currentSolution, float currentValue, int
	noOfRuns)
	{
		if(outputMedia.equals("file"))
		{
			outputResult.println();
			outputResult.print(noOfRuns+"\t" + currentValue+"\t\t");

			for(int z= 0; z<currentSolution.length; z++)
				outputResult.print(currentSolution[z]+"\t");

		}
		else
		{
			System.out.println();
			System.out.print(noOfRuns+"\t" + currentValue+"\t");
		}

	}
}

--------------------------------End of printClass.java---------------------

/**...........................................................
 * randomFS.java
 * This class Selects the 'optimal'feature subsets randomly
 * @author   Idowu Olayinka Oduntan
 * @version  2005-Feb-21
 */

public class randomFS
{
	parameter localParamObj;
	localOptimum localOptimumObj;
	initialSolutionManager initialSolutionObj;
	int localStartDimension, localEndDimension, localSolutionSize,
	randomSolutionSetSize, localNoOfIterations;
	public int[][] randomSolutionSet;
	public int[] randomOptimalSolution;
	public float[] randomOptimalSolutionValues;
	public float randomOptimalValue =1000000f;
	public int randomSearchCost = 0;

	// Create initial solution
	// Generate random solutions
	// Find the best amongst the randomly generated set

public randomFS(parameter paramObj)
{
localParamObj = paramObj;
localStartDimension = localParamObj.startDimension;
localEndDimension = localParamObj.endDimension;
localSolutionSize = localParamObj.solutionSize;
localNoOfIterations = localParamObj.noOfIterations;
randomSolutionSetSize = localParamObj.neighborhoodSize *
localParamObj.noOfIterations;
initialSolutionObj = new initialSolutionManager(localStartDimension,
localEndDimension, localSolutionSize);
localOptimumObj = new localOptimum(localParamObj);
randomSolutionSet = new int[randomSolutionSetSize][localSolutionSize];
randomOptimalSolutionValues = new float[randomSolutionSetSize];
}

public int[] optimalRandomSearch()
{
       for(int i=0; i<randomSolutionSetSize; i++)
       {
       randomSolutionSet[i] =
       (int[])initialSolutionObj.starterInitialSolution().clone();
       randomOptimalSolutionValues[i] =
       localOptimumObj.computeObjValue(randomSolutionSet[i]);
       randomSearchCost++;
       }

       for(int m=0; m<randomSolutionSetSize-1; m++)
       {
       for(int n=0; n<randomSolutionSetSize-1-m; n++)
       {
       if(randomOptimalSolutionValues[n] < randomOptimalSolutionValues[n+1])
       {
       float temp1 = randomOptimalSolutionValues[n];
       randomOptimalSolutionValues[n] = randomOptimalSolutionValues[n+1];
       randomOptimalSolutionValues[n+1] = temp1;
       int[] temp2 = (int[])randomSolutionSet[n].clone();
       randomSolutionSet[n] = (int[])randomSolutionSet[n+1].clone();
       randomSolutionSet[n+1] = (int[])temp2.clone();
       }
       }
       }
       if(randomOptimalSolutionValues[randomSolutionSetSize-
       1]<=randomOptimalValue)
       {
       randomOptimalSolution = (int[])randomSolutionSet[randomSolutionSetSize-
       1].clone();
       randomOptimalValue = randomOptimalSolutionValues[randomSolutionSetSize-
       1];
       }

       return (int[])randomOptimalSolution.clone();
       }

}

-------------------------End of randomFS.java-------------------------------

/**.........................................................
 * rankFeatureSubset.java
 * This class ranks a subset of feature based on discriminatory capability
 * @author   Idowu Olayinka Oduntan
 * @version  2005-Feb-21
 */
public class rankFeatureSubset
{
       public int[] solutionSubset;
public float bestRankValue;
public int rankingCost = 0;
public int[] rankedSubspace;
objectiveFunction objFunction;
int[] bestRanked;
int[] bestRankedArrays;
int[] localCurrentSubsets;
int[][] localCurrentSubsetArrays;
int[] rankedSubsets;
int[][] rankedSubsetArrays;
float[] rankValues;

float objFunctValue;
boolean subsetArrays;

public rankFeatureSubset(parameter paramObj, int[] givenSubsets)
{
       //Get the training datasets
       objFunction = new objectiveFunction(paramObj);
       localCurrentSubsets = (int[])givenSubsets.clone();
       rankedSubsets = (int[])givenSubsets.clone();
       bestRanked = new int[1];
       rankValues = new float[givenSubsets.length];
       subsetArrays = false;
}

public rankFeatureSubset(parameter paramObj, int[][] givenSubsets)
{
       //Get the training datasets
       objFunction = new objectiveFunction(paramObj);
       localCurrentSubsetArrays = (int[][])givenSubsets.clone();
       rankedSubsetArrays = (int[][])givenSubsets.clone();
       bestRankedArrays = new int[givenSubsets[0].length];
       rankValues = new float[givenSubsets.length];
       subsetArrays = true;
}

public int[] rankingSubset()
{

if(!subsetArrays)
{
for(int i=0; i<localCurrentSubsets.length; i++)
{
int[] tempSolution = new int[1];
tempSolution[0] = localCurrentSubsets[i];
rankingCost++;
rankValues[i] = objFunction.meanSquareError(tempSolution);
}

// Order the subsets in descending order of ranking criterion
for(int j=0; j<localCurrentSubsets.length-1; j++)
{
for(int k=0; k<localCurrentSubsets.length-1-j; k++)
{
if(rankValues[k] < rankValues[k+1])
{
float temp1 = rankValues[k];
rankValues[k] = rankValues[k+1];
rankValues[k+1] = temp1;
int temp2 = rankedSubsets[k];
rankedSubsets[k] = rankedSubsets[k+1];
rankedSubsets[k+1] = temp2;
```

```
        }
        }
        }
        rankedSubspace = (int[])rankedSubsets.clone();
        //The best subset is the last array element.
        bestRanked[0] = rankedSubsets[localCurrentSubsets.length-1];
        //The best subset value is the last array element.
        bestRankValue = rankValues[localCurrentSubsets.length-1];
        return (int[]) bestRanked.clone();
        }

        else
        {
        for(int i=0; i<localCurrentSubsetArrays.length; i++)
        {
                rankingCost++;
                rankValues[i] =
                objFunction.meanSquareError((int[])localCurrentSubsetArrays[i].clo
                ne());
                }
                for(int j=0; j<localCurrentSubsetArrays.length-1; j++)
                {
                for(int k=0; k<localCurrentSubsetArrays.length-1-j; k++)
                {
                if(rankValues[k] < rankValues[k+1])
                {
                float temp1 = rankValues[k];
                rankValues[k] = rankValues[k+1];
                rankValues[k+1] = temp1;
                int[] temp2 = (int[])rankedSubsetArrays[k].clone();
                rankedSubsetArrays[k] = (int[])rankedSubsetArrays[k+1].clone();
                rankedSubsetArrays[k+1] = (int[])temp2.clone();
        }
        }
        }
//The best subset is the last array element.
bestRanked = (int[])rankedSubsetArrays[localCurrentSubsetArrays.length-
1].clone();
//The best subset value is the last array element.
bestRankValue = rankValues[localCurrentSubsetArrays.length-1];
return (int[]) bestRanked.clone();
}
}
}

----------------------------End of rankFeatureSubset.java---------------------


/**.......................................................
 * refiner.java
 * This class generates an improved solution from a less coarse
 * feature subspace using solution from an immediate coarser feature space
 * @author   Idowu Olayinka Oduntan
 * @version  August 2005
 */

public class refiner
{
        public int multilevelCost = 0;
        int[][] localLevelSubSpaces;
        int[][] localLevelSubSpacesWithComponents;
        int[] finalRefineSolution;
        int[][] finalRefineEliteSolution;
```

```
int[] initialSolution;
int[][] initialEliteSolution;
int solutionSize, localNoOfIterations, localEliteSize, localSampleSize,
localClusterWindowSize;
float finalRefineSolutionValue;
float[] finalRefineEliteSolutionValue;
float[][] localGlobalDataset;
float initialRefineSolutionValue;
float[] initialRefineEliteSolutionValue;
float[] tempFinalRefineEliteSolutionValue;
float[][][] tempNewDataValues;
int[][] tempFinalRefineEliteSolution;

parameter paramObj;
initialSolutionManager initialSolutionObj, withinInitSolutionObj;
localOptimum localOptimumObj;

// Refiner for single solution variable pre-setting coarsening strategy
refiner(int[][] givenLevelSubSpaces, parameter givenParam)
{
        paramObj = givenParam;
        localLevelSubSpaces = (int[][])givenLevelSubSpaces.clone();

        localNoOfIterations = paramObj.noOfIterations;
        paramObj.resetSolutionSize();
        solutionSize = paramObj.solutionSize;
        finalRefineSolution = (int[])refineProcess().clone();

        paramObj.resetTabuListSize();
        paramObj.resetNeighborhoodSize();
        paramObj.resetNoOfIterations();

}

// Refiner for elite solution set with variable pre-setting coarsening
strategy
refiner(int[][] givenLevelSubSpaces, parameter givenParam, int eliteSize)
{
paramObj = givenParam;
localEliteSize = eliteSize;
localLevelSubSpaces = (int[][])givenLevelSubSpaces.clone();

localNoOfIterations = paramObj.noOfIterations;
paramObj.resetSolutionSize();
solutionSize = paramObj.solutionSize;
finalRefineEliteSolution =
(int[][])refineProcess(localEliteSize).clone();
tempFinalRefineEliteSolution = (int[][])finalRefineEliteSolution.clone();
tempFinalRefineEliteSolutionValue =
(float[])finalRefineEliteSolutionValue.clone();

for(int i=0; i<initialEliteSolution.length; i++)
{
System.out.println("Initial elite solution set");
System.out.print(initialRefineEliteSolutionValue[i]+"\t");
for(int j=0; j<initialEliteSolution[0].length; j++)
{
System.out.print(initialEliteSolution[i][j]+"\t");
}
}

System.out.println();
for(int i=0; i<finalRefineEliteSolution.length; i++)
```

```
{
System.out.println("Final elite solution set");
System.out.print(finalRefineEliteSolutionValue[i]+"\t");
for(int j=0; j<finalRefineEliteSolution[0].length; j++)
{
System.out.print(finalRefineEliteSolution[i][j]+"\t");
}
}

System.out.println();
for(int j=0; j<finalRefineEliteSolution.length-1; j++)
{
for(int k=0; k<finalRefineEliteSolution.length-1-j; k++)
{
if(tempFinalRefineEliteSolutionValue[k] <
tempFinalRefineEliteSolutionValue[k+1])
{
float temp1 = tempFinalRefineEliteSolutionValue[k];
tempFinalRefineEliteSolutionValue[k] =
tempFinalRefineEliteSolutionValue[k+1];
tempFinalRefineEliteSolutionValue[k+1] = temp1;
int[] temp2 = (int[])tempFinalRefineEliteSolution[k].clone();
tempFinalRefineEliteSolution[k] =
(int[])tempFinalRefineEliteSolution[k+1].clone();
tempFinalRefineEliteSolution[k+1] = (int[])temp2.clone();
}
}
}
finalRefineSolution =
tempFinalRefineEliteSolution[finalRefineEliteSolution.length-1];
finalRefineSolutionValue =
tempFinalRefineEliteSolutionValue[finalRefineEliteSolution.length-1];
paramObj.resetTabuListSize();
paramObj.resetNeighborhoodSize();
paramObj.resetNoOfIterations();
}
//Refine for single solution with clustered coarsening strategy
refiner(int[][] givenLevelSubSpaces, int[][]
givenSubspacesWithComponents, parameter givenParam)
{
        paramObj = givenParam;
        localGlobalDataset = (float[][])paramObj.globalDataSet.clone();
        localLevelSubSpaces = (int[][])givenLevelSubSpaces.clone();
        localLevelSubSpacesWithComponents =
        (int[][])givenSubspacesWithComponents.clone();
        localNoOfIterations = paramObj.noOfIterations;
        paramObj.resetSolutionSize();
        solutionSize = paramObj.solutionSize;
        localSampleSize = paramObj.globalDataSet.length;
        localClusterWindowSize = paramObj.clusterWindow;

        finalRefineSolution =
        (int[])refineProcess(givenSubspacesWithComponents).clone();
        paramObj.resetTabuListSize();
        paramObj.resetNeighborhoodSize();
        paramObj.resetNoOfIterations();

}

//Refine processing for single solution with variable pre-setting
coarsening
public int[] refineProcess()
{
        initialSolutionObj = new
        initialSolutionManager(localLevelSubSpaces[localLevelSubSpaces.len
        gth-1], solutionSize);
        initialSolution =
        (int[])initialSolutionObj.starterInitialSolution(true).clone();

for(int i=localLevelSubSpaces.length - 1; i>=0; i--)
{
if(localLevelSubSpaces[i].length < paramObj.tabuListSize)
paramObj.setTabuListSize(10);
if((localLevelSubSpaces[i].length < paramObj.neighborhoodSize))
paramObj.setNeighborhoodSize(15);
paramObj.setNoOfIterations((localLevelSubSpaces.length -
i)*localNoOfIterations);
localOptimumObj = new localOptimum(paramObj, localLevelSubSpaces[i]);
initialSolution =
(int[])localOptimumObj.localOptimumSearch(initialSolution).clone();
multilevelCost = multilevelCost + localOptimumObj.computationCounter;
}
paramObj.resetTabuListSize();
paramObj.resetNeighborhoodSize();
paramObj.resetNoOfIterations();
finalRefineSolutionValue = localOptimumObj.localOptimumValue;
return (int[])initialSolution.clone();
}
public int[] refineProcess(int[][] givenSubspaceWithComponents)
{
tempNewDataValues = new
float[localLevelSubSpaces.length][localSampleSize][];
tempNewDataValues[0] = (float[][])localGlobalDataset.clone();
int[][] localSubspaceWithComponents =
(int[][])givenSubspaceWithComponents.clone();
initialSolutionObj = new
initialSolutionManager(localLevelSubSpaces[localLevelSubSpaces.length-1],
solutionSize);
initialSolution =
(int[])initialSolutionObj.starterInitialSolution(true).clone();
for(int i=1; i<localLevelSubSpaces.length; i++)
{
float[][] tempValues = new
float[localSampleSize][localLevelSubSpaces[i].length];
for(int j=0; j<localSampleSize; j++)
{
for(int k=0; k<localLevelSubSpaces[i].length; k++)
{
int windowCounter = 0;
float newDataValue = 0.0f;
int componentCounter1 = (localClusterWindowSize+1)*k + 1;
for(int m=componentCounter1; m<componentCounter1+localClusterWindowSize;
m++)
{
if(m<localSubspaceWithComponents[i].length)
{
newDataValue = newDataValue + tempNewDataValues[i-
1][j][localClusterWindowSize*k+windowCounter];
windowCounter++;
}
}
tempValues[j][k]=newDataValue/windowCounter;
}
tempNewDataValues[i] = (float[][])tempValues.clone();
}
```

```
}
for(int i=localLevelSubSpaces.length - 1; i>=0; i--)
{
paramObj.setTempGlobalDataset(localLevelSubSpaces[i],
localSubspaceWithComponents[i], tempNewDataValues[i]);
if(localLevelSubSpaces[i].length < paramObj.tabuListSize)
paramObj.setTabuListSize(10);
if((localLevelSubSpaces[i].length < paramObj.neighborhoodSize))
paramObj.setNeighborhoodSize(15);
// Set the number of iteration for the given coarse subspace.
paramObj.setNoOfIterations((localLevelSubSpaces.length -
i)*localNoOfIterations);
localOptimumObj = new localOptimum(paramObj, localLevelSubSpaces[i]);
if(i==localLevelSubSpaces.length - 1)
{
initialSolution =
(int[])localOptimumObj.localOptimumSearch(initialSolution).clone();
multilevelCost = multilevelCost + localOptimumObj.computationCounter;
}
else
{
int[] tempInitialSolution = (int[])initialSolution.clone();
int[] tempInitialSolutionSpace = new
int[paramObj.clusterWindow*solutionSize];
int initialSolutionSpaceCounter = 0;
for(int m=0; m<tempInitialSolution.length; m++)
{
for(int j=0; j<localLevelSubSpaces[i+1].length; j++)
{
int indexCounter = j*(paramObj.clusterWindow+1);
if(tempInitialSolution[m]==localSubspaceWithComponents[i+1][indexCounter]
)
{
for(int n=0; n<paramObj.clusterWindow; n++)
{
tempInitialSolutionSpace[initialSolutionSpaceCounter] =
localSubspaceWithComponents[i+1][++indexCounter];
initialSolutionSpaceCounter++;
}
break;
}
}
}
int nonZeroCounter = 0;
for(int j=0; j<tempInitialSolutionSpace.length; j++)
{
if(tempInitialSolutionSpace[j]!=0)
nonZeroCounter++;
}
int initialCounter = 0;
int[] initialSolutionSpace = new int[nonZeroCounter];
System.out.println("The values of tempInitialSolution:");
for(int j=0; j<tempInitialSolutionSpace.length; j++)
{
System.out.print(tempInitialSolutionSpace[j]+"\t");
if(tempInitialSolutionSpace[j]!=0)
{
initialSolutionSpace[initialCounter] = tempInitialSolutionSpace[j];
initialCounter++;
}
}
System.out.println("The solution of new feature space:
"+initialSolutionSpace.length);
withinInitSolutionObj = new initialSolutionManager(initialSolutionSpace,
solutionSize);
int[] tempInitialSolution_2 =
(int[])withinInitSolutionObj.starterInitialSolution(true).clone();
initialSolution =
(int[])localOptimumObj.localOptimumSearch(tempInitialSolution_2).clone();
}
paramObj.resetTempGlobalDataset();
}
finalRefineSolutionValue = localOptimumObj.localOptimumValue;
paramObj.resetTempGlobalDataset();
paramObj.resetTabuListSize();
paramObj.resetNeighborhoodSize();
paramObj.resetNoOfIterations();
return (int[])initialSolution.clone();
}
//Refine processing for elite solution set with variable pre-setting
coarsening
public int[][] refineProcess(int eliteSize)
{
initialEliteSolution = new int[eliteSize][];
initialRefineEliteSolutionValue = new float[eliteSize];
finalRefineEliteSolutionValue = new float[eliteSize];
initialSolutionObj = new
initialSolutionManager(localLevelSubSpaces[localLevelSubSpaces.length-1],
solutionSize);
objectiveFunction objFunction = new objectiveFunction(paramObj);

for(int i=0; i<eliteSize; i++)
{
initialEliteSolution[i] =
(int[])initialSolutionObj.starterInitialSolution(true).clone();
initialRefineEliteSolutionValue[i] =
objFunction.meanSquareError(initialEliteSolution[i]);
}

for(int j=0; j<eliteSize; j++)
{
for(int i=localLevelSubSpaces.length - 1; i>=0; i--)
{
if(localLevelSubSpaces[i].length < paramObj.tabuListSize)
paramObj.setTabuListSize(10);
if((localLevelSubSpaces[i].length < paramObj.neighborhoodSize))
paramObj.setNeighborhoodSize(15);
paramObj.setNoOfIterations((localLevelSubSpaces.length -
i)*localNoOfIterations);

localOptimumObj = new localOptimum(paramObj, localLevelSubSpaces[i]);
initialEliteSolution[j] =
(int[])localOptimumObj.localOptimumSearch(initialEliteSolution[j]).clone(
);
multilevelCost = multilevelCost + localOptimumObj.computationCounter;
}

paramObj.resetTabuListSize();
paramObj.resetNeighborhoodSize();
paramObj.resetNoOfIterations();
finalRefineEliteSolutionValue[j] = localOptimumObj.localOptimumValue;
}
return (int[][])initialEliteSolution.clone();
}
```

```
}
------------------------End of refiner.java--------------------------------

/**..........................................................
 * SFS.java
 * This class implements the SFS (Sequential Forward Selection) technique
 * @author          Idowu Olayinka Oduntan
 * @version         August 2005
 */
public class SFS
{
	public float optimalValue;
	public int SFSCost = 0;
	parameter localParameter;
	int localSolutionSize, featureSpaceDimension;
	int[] optimalSubset;
	int[] originalFeatureSet;
	float[] solutionValueSet;

	public SFS(parameter paramObj)
	{
	localParameter = paramObj;
	localSolutionSize = localParameter.solutionSize;
	featureSpaceDimension = localParameter.noOfDimension;
	originalFeatureSet = new int[featureSpaceDimension];
	for(int i=0; i<featureSpaceDimension; i++)
	{
		originalFeatureSet[i] = i+1;
	}

	rankFeatureSubset rankingObj = new rankFeatureSubset(localParameter,
	originalFeatureSet);

	// Generate initial solution
	optimalSubset = (int[])rankingObj.rankingSubset().clone();
	SFSCost = SFSCost + rankingObj.rankingCost;

	}

	public SFS(parameter paramObj, int[] givenFeatureSpace, int
	newSolutionSize)
	{
	localParameter = paramObj;
	localSolutionSize = newSolutionSize;
	featureSpaceDimension = givenFeatureSpace.length;
	originalFeatureSet = (int[])givenFeatureSpace.clone();
	rankFeatureSubset rankingObj = new rankFeatureSubset(localParameter,
	originalFeatureSet);

	// Generate initial solution
	optimalSubset = (int[])rankingObj.rankingSubset().clone();
	SFSCost = SFSCost + rankingObj.rankingCost;
	}

	public int[][] generateNewSolutionSubset(int[] currentSolution)
	{
	int[] tempOriginalFeatures = (int[])originalFeatureSet.clone();
	int featureSpaceSize = featureSpaceDimension - currentSolution.length;
	int[][] solutionSubsets = new
	int[featureSpaceSize][currentSolution.length+1];
	int[] remainingFeatures = new int[featureSpaceDimension-
	currentSolution.length];
```

```
int m=0;
int n=0;
// Assign the current solution to all the subsets in the solutionSubsets
for(int i=0; i<featureSpaceSize; i++)
for(int j=0; j<currentSolution.length; j++)
solutionSubsets[i][j] = currentSolution[j];
// Begin: Find the features that are in the feature space but not in the
current solution set
for(int j=0; j<featureSpaceDimension; j++)
{
//boolean statusFlag = false;
int k=0;
while(k<currentSolution.length)
{
if(tempOriginalFeatures[j]==currentSolution[k])
{
tempOriginalFeatures[j] = 0;
break;
}
k++;
}
}
while(m<tempOriginalFeatures.length && n<remainingFeatures.length)
{
if(tempOriginalFeatures[m]!=0)
{
remainingFeatures[n] = tempOriginalFeatures[m];
n++;
}
m++;
}
// End: Find the features that are in the feature space but not in the
current solution set

for(int i=0; i<featureSpaceSize; i++)
{
solutionSubsets[i][currentSolution.length] = remainingFeatures[i];
}
return (int[][])solutionSubsets.clone();
}

public int[] findOptimalSubset()
{
int[] currentSolution;
currentSolution = (int[])optimalSubset.clone();
for(int i=0; i<localSolutionSize; i++)
{
       System.out.println();
       for(int r=0; r<currentSolution.length; r++)
              System.out.print(currentSolution[r]+"\t");
       int[][] currentSolutionSubset =
       (int[][])generateNewSolutionSubset((int[])currentSolution.clone())
       .clone();
       rankFeatureSubset rankingObj = new
       rankFeatureSubset(localParameter,
       (int[][])currentSolutionSubset.clone());
       currentSolution = (int[])rankingObj.rankingSubset().clone();

       // Return the OPTIMAL feature subset
       SFSCost = SFSCost + rankingObj.rankingCost;
       optimalValue = rankingObj.bestRankValue;
       }
```

```
                System.out.println();
                System.out.print("\n The computational cost for SFS is:
                "+SFSCost+"\n");

                return (int[])currentSolution.clone();

        }

}

------------------------------End of SFS-------------------------------


/**
 * tabuList.java
 * implements tabu list in tabu search
 * The tabu list is implemented as a prioritized queue
 * @author       Idowu O. Oduntan
 * @version      August 2005
 */

public class tabuList
{
        public int tabuListLength;
        public int noOftabuListElements;
        public int[][] tabuElements;
        public float[] tabuValues;
        public int bottomQueueCounter;
        public boolean foundInTabuList;
        public int eliminateIndex;
        public int tabuEffectCounter;

        // Proper tabu list construct
        public tabuList(int tabuLength, int noOfElements, parameter paramObj)
        {
        tabuListLength = tabuLength;
        noOftabuListElements = noOfElements;
        tabuElements = new int [tabuListLength][noOftabuListElements];
        tabuValues = new float[tabuListLength];
        bottomQueueCounter = 0;
        if(paramObj.optimumType.equals("MIN"))
                eliminateIndex = 0;
        else
                eliminateIndex = tabuListLength-1;
        for (int i=0; i<tabuListLength; i++)
        {
                for (int j=0; j<noOftabuListElements; j++)
                        tabuElements[i][j] = 0;
                tabuValues[i] = 10000000f;
        }
        }

        public void addToTabu(int[] solution, float tabuSolutionValue)
        {
                boolean tabuChecker = checkInTabu((int[])solution.clone());
                for(int m=0; m<tabuListLength-1; m++)
                {
                for(int n=0; n<tabuListLength-1-m; n++)
                {
                if(tabuValues[n] < tabuValues[n+1])
                {
                float temp1 = tabuValues[n];
                tabuValues[n] = tabuValues[n+1];
                tabuValues[n+1] = temp1;
                int[] temp2 = (int[])tabuElements[n].clone();
                tabuElements[n] = (int[])tabuElements[n+1].clone();
                tabuElements[n+1] = (int[])temp2.clone();
        }
        }
        }
        if(!tabuChecker)
        {
        for (int i=0; i<noOftabuListElements; i++)
        tabuElements[eliminateIndex][i] = solution[i];
        tabuValues[eliminateIndex] = tabuSolutionValue;
        }
        }

public void removeFromTabu()
{
        for (int i=0; i<tabuListLength-1; i++)
                for (int j=0; j<noOftabuListElements; j++)
                        tabuElements[i][j] = tabuElements[i+1][j];
}

public boolean checkInTabu(int[] solution)
{
        foundInTabuList = false;
        int tabuListCounter = 0;
        while(tabuListCounter<tabuListLength)
        {
        int solutionCounter = 0;
        while(solutionCounter < solution.length &&
        (tabuElements[tabuListCounter][solutionCounter]==solution[solutionCounter
        ]))
        {
        solutionCounter++;
        }
        if(solutionCounter>=solution.length-1)
        {
        foundInTabuList = true;
        break;
        }
        tabuListCounter++;
        }
        if(foundInTabuList)
        tabuEffectCounter++;
        return foundInTabuList;
}

--------------------------------End of tabuList.java--------------------------


/**..............................................................
 * This class coordinates the overall searches and the generated solutions
 * @author          Idowu Olayinka Oduntan
 * @version         August 2005
 */
public class testCoarseAlgorithms
{
        public static void main(String[] args)
        {
        Classifier classifierObj;
        parameter initParamObj = new parameter();
        printClass printObj = new printClass();
        int localNoOfLevels, localNoOfRuns, localRandomnessFactor;
```

```
int[] testCoarseSpace;
int[][] localFeatureSubspaceWithComponents;
int[][] levelSubSpaces;
int[] finalSolution;
float finalSolutionValue;
float meanAccuracy = 0f;
float meanAccuracyTB = 0f;
float meanObjFunct = 0f;
float meanObjFunctTB = 0f;
float standardDeviation = 0f;
float standardDeviationTB = 0f;
double stdAccuracy;
double stdAccuracyTB;
float[] classifyAccuracyValues;
float[] classifyAccuracyValuesTB;
float[] classifyAccuracyValuesRFS;
float[] deviationValues;
float[] deviationValuesTB;
float[] deviationValuesRFS;
float[] objFunctValues;
float[] objFunctValuesTB;
float[] objFunctValuesRFS;
String outputFile;
String levelVariables;
localNoOfLevels = initParamObj.noOfLevels;
localRandomnessFactor = initParamObj.randomnessFactor;
outputFile = initParamObj.outputFileName;
localNoOfRuns = initParamObj.noOfRuns;
classifyAccuracyValues = new float[localNoOfRuns];
classifyAccuracyValuesTB = new float[localNoOfRuns];
classifyAccuracyValuesRFS = new float[localNoOfRuns];
deviationValues = new float[localNoOfRuns];
deviationValuesTB = new float[localNoOfRuns];
deviationValuesRFS = new float[localNoOfRuns];
objFunctValues = new float[localNoOfRuns];
objFunctValuesTB = new float[localNoOfRuns];
objFunctValuesRFS = new float[localNoOfRuns];
String literal;
float[] tempClassifyAccuracyValues = new float[localRandomnessFactor];
float[] tempFinalSolutionValues = new float[localRandomnessFactor];
int[][] tempFinalSolution = new
int[localRandomnessFactor][initParamObj.solutionSize];

System.out.println("Starts processing...");
printObj.printHeader("\n\nFinal selection solution \n\nRunNo \t
ObjectiveFunctValue \t 'optimal' Subset");
for(int k=0; k<localNoOfRuns; k++)
{
parameter paramObj = new parameter();
dataClass dataObj = new dataClass(paramObj);
int[] featureSolutionSpace = new int[1500];
levelSubSpaces = new int[localNoOfLevels][];
for(int i=0; i<1500; i++)
{
featureSolutionSpace[i] = i+1;
}
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 5";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
// Set the no. of levels and the no. of iterations at levels.
paramObj.setNoOfIterations(5); //Set no. of iteration to 50;
```

```
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j],
j);

classifierObj = new Classifier((int[])tempFinalSolution[j].clone(),
paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);

System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");

//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default
value;
paramObj.resetNoOfLevels();      //Set no. of levels to default value;
}

// Training and classification using multilevel search with different no.
of levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 10";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
       // Set the no. of levels and the no. of iterations at levels.
       paramObj.setNoOfIterations(10); //Set no. of iteration to 50;
       //paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
       testCoarseSpace = (int[])featureSolutionSpace.clone();
       coarser coarseObj = new coarser(paramObj);
       coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
       levelSubSpaces =
       (int[][])coarseObj.featureSubspacePartitions.clone();

       refiner refinerObj = new refiner(levelSubSpaces, paramObj);
       tempFinalSolution[j] =
       (int[])refinerObj.finalRefineSolution.clone();
       tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;

       printObj.printOutput(tempFinalSolution[j],
       tempFinalSolutionValues[j], j);

       classifierObj = new
       Classifier((int[])tempFinalSolution[j].clone(), paramObj);
       tempClassifyAccuracyValues[j] =
       classifierObj.classificationAccuracy;
       literal  = "\nTrainset Classification accuracy:
       "+classifierObj.trainClassificationAccuracy+"%";
       printObj.printLiteral(literal);
       literal  = "\nIndependent Testset Classification accuracy:
       "+classifierObj.classificationAccuracy+"%";
       printObj.printLiteral(literal);
```

```
            System.out.println("Computation cost of Multilevel FS:
            "+refinerObj.multilevelCost+"\n");

            //Reset the no. of levels and the no. of iterations at levels.
            paramObj.resetNoOfIterations(); //reset no. of iteration to
            default value;
            paramObj.resetNoOfLevels();//Set no. of levels to default value;
            }

            // Training and classification using multilevel search with
            different no. of levels
            literal = "\n\nMultilevel result for 3-levels; reduction factor 3;
            random coarsening; no of basic iteration 15";
            printObj.printLiteral(literal);
            for(int j=0; j<localRandomnessFactor; j++)
            {
            // Set the no. of levels and the no. of iterations at levels.
            paramObj.setNoOfIterations(15); //Set no. of iteration to 50;
            //paramObj.setNoOfLevels(2);      //Set no. of levels to 2;

            testCoarseSpace = (int[])featureSolutionSpace.clone();
            coarser coarseObj = new coarser(paramObj);
            coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
            levelSubSpaces =
            (int[][])coarseObj.featureSubspacePartitions.clone();

            refiner refinerObj = new refiner(levelSubSpaces, paramObj);
            tempFinalSolution[j] =
            (int[])refinerObj.finalRefineSolution.clone();
            tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
            printObj.printOutput(tempFinalSolution[j],
            tempFinalSolutionValues[j], j);

            classifierObj = new
            Classifier((int[])tempFinalSolution[j].clone(), paramObj);
            tempClassifyAccuracyValues[j] =
            classifierObj.classificationAccuracy;
            literal  = "\nTrainset Classification accuracy:
            "+classifierObj.trainClassificationAccuracy+"%";
            printObj.printLiteral(literal);
            literal  = "\nIndependent Testset Classification accuracy:
            "+classifierObj.classificationAccuracy+"%";
            printObj.printLiteral(literal);

            System.out.println("Computation cost of Multilevel FS:
            "+refinerObj.multilevelCost+"\n");

            //Reset the no. of levels and the no. of iterations at levels.
            paramObj.resetNoOfIterations(); //reset no. of iteration to
            default value;
            paramObj.resetNoOfLevels(); //Set no. of levels to default value;
            }

// Training and classification using multilevel search with different no. of
levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 20";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
// Set the no. of levels and the no. of iterations at levels.
paramObj.setNoOfIterations(20); //Set no. of iteration to 50;
//paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
```

```
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j], j);
classifierObj = new Classifier((int[])tempFinalSolution[j].clone(), paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);
System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");
//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default value;
paramObj.resetNoOfLevels();      //Set no. of levels to default value;
}
// Training and classification using multilevel search with different no. of
levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 25";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
// Set the no. of levels and the no. of iterations at levels.
paramObj.setNoOfIterations(25); //Set no. of iteration to 50;
//paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j], j);
classifierObj = new Classifier((int[])tempFinalSolution[j].clone(), paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);
System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");
//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default value;
paramObj.resetNoOfLevels();             //Set no. of levels to default value;
}

// Training and classification using multilevel search with different no. of
levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 30";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
// Set the no. of levels and the no. of iterations at levels.
```

```
paramObj.setNoOfIterations(30); //Set no. of iteration to 50;
//paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j], j);
classifierObj = new Classifier((int[])tempFinalSolution[j].clone(), paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);
System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");
//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default value;
paramObj.resetNoOfLevels();             //Set no. of levels to default value;
}
// Training and classification using multilevel search with different no. of
levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 35";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
// Set the no. of levels and the no. of iterations at levels.
paramObj.setNoOfIterations(35); //Set no. of iteration to 50;
//paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j], j);
classifierObj = new Classifier((int[])tempFinalSolution[j].clone(), paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);
System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");
//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default value;
paramObj.resetNoOfLevels();             //Set no. of levels to default value;
}
// Training and classification using multilevel search with different no. of
levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 40";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
```

```
// Set the no. of levels and the no. of iterations at levels.
paramObj.setNoOfIterations(40); //Set no. of iteration to 50;
//paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j], j);
classifierObj = new Classifier((int[])tempFinalSolution[j].clone(), paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);
System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");
//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default value;
paramObj.resetNoOfLevels();            //Set no. of levels to default value;
}
// Training and classification using multilevel search with different no. of
levels
literal = "\n\nMultilevel result for 3-levels; reduction factor 3; random
coarsening; no of basic iteration 45";
printObj.printLiteral(literal);
for(int j=0; j<localRandomnessFactor; j++)
{
// Set the no. of levels and the no. of iterations at levels.
paramObj.setNoOfIterations(45); //Set no. of iteration to 50;
//paramObj.setNoOfLevels(2);            //Set no. of levels to 2;
testCoarseSpace = (int[])featureSolutionSpace.clone();
coarser coarseObj = new coarser(paramObj);
coarseObj.randomCoarsen((int[])testCoarseSpace.clone());
levelSubSpaces = (int[][])coarseObj.featureSubspacePartitions.clone();
refiner refinerObj = new refiner(levelSubSpaces, paramObj);
tempFinalSolution[j] = (int[])refinerObj.finalRefineSolution.clone();
tempFinalSolutionValues[j] = refinerObj.finalRefineSolutionValue;
printObj.printOutput(tempFinalSolution[j], tempFinalSolutionValues[j], j);
classifierObj = new Classifier((int[])tempFinalSolution[j].clone(), paramObj);
tempClassifyAccuracyValues[j] = classifierObj.classificationAccuracy;
literal  = "\nTrainset Classification accuracy:
"+classifierObj.trainClassificationAccuracy+"%";
printObj.printLiteral(literal);
literal  = "\nIndependent Testset Classification accuracy:
"+classifierObj.classificationAccuracy+"%";
printObj.printLiteral(literal);
System.out.println("Computation cost of Multilevel FS:
"+refinerObj.multilevelCost+"\n");
//Reset the no. of levels and the no. of iterations at levels.
paramObj.resetNoOfIterations(); //reset no. of iteration to default value;
paramObj.resetNoOfLevels();            //Set no. of levels to default value;
}
}

if(initParamObj.outputMedia.equals("file"))
System.out.println("\nEnd of process... check the output file
"+"'"+initParamObj.outputFileName+"'"+" for the process reults.");
else
```

```
System.out.println("\nEnd of process... ");
}
}
-------------------------End of testCaorseAlgorithm.java---------------------
```

References

[1] D. W. Aha and R. L. Bankert. A comparative evaluation of sequential feature selection algorithms. In *Proceedings of the Fifth International Workshop on Artificial Intelligence and Statistics*, pages 1-7, Ft. Lauderdale, FL, January 1995.

[2] C. J. Alpert, J. - H. Huang and A. B. Kahng. Multilevel circuit partitioning. In *Proc. 34th ACM/IEEE Design Automation Conference*, pages 530-533, June 1997.

[3] C. Ambroise and G. J. McLachlan. Selection bias in gene extraction on the basis of microarray gene-expression data. *Proceedings of the National Academy of Sciences of the USA*, 99(10): 6562–6566, 2002.

[4] S. T. Barnard and H. D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. *Concurrency: Practice & Experience*, 6(2):101-117, 1994.

[5] A. Brandt. Multi-level adaptive solutions to boundary value problems. *Mathematics of Computation*, 31:333-390, 1977.

[6] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A multigrid tutorial, second edition*. SIAM 2000.

[7] S.A. Bustin and S. Dorudi. The value of microarray techniques for quantitative gene profiling in molecular diagnostics. *Trends in Molecular Medicine,* 8:269–272, 2002.

[8] R. Caruana and D. Freitag. Greedy attribute selection. *In Proceedings of the 11th International Conference in Machine Learning*, pages 28-36, New Brunswick, New Jersey, July 1994.

[9] S. Das. Filters, Wrappers and a Boosting-Based Hybrid for Feature Selection. *In Proceedings of the 18th International Conference in Machine Learning*, pages 359-366, Williamstown, Massachusetts, June 2001.

[10] K. Fukunaga. *Introduction to Statistical Pattern Recognition, 2nd edition.* Academic Boston, 1990.

[11] C. Furlanello, M. Serafini, S. Merler and G. Jurman. An accelerated procedure for recursive feature ranking on microarray data. *Neural Networks* 16:641-648, 2003.

[12] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, 1979.

[13] F. Glover and M. Laguna. *Tabu Search.* Kluwer Academic Publishers, 1997.

[14] F. Glover, E. Taillard, M. Laguna, D. de Werra. A user's guide to taboo search. *Annals of Operations Research,* 41:3-28, 1993.

[15] I. Guyon and A. Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(7-8):1157-1182, 2003.

[16] M. A. Hall. Correlation-based feature selection for discrete and numeric class machine learning. *In Proceedings of the 17th International Conference in Machine Learning*, pages 359-366, Stanford, California, June 2000.

[17] A. Hertz and D. de Werra. The tabu search metaheuristic: how we used it. *Annals of Mathematics and Artificial Intelligence,* 1:111-121, 1990.

[18] F. van der Heijden, R. P. W. Duin. *Classification, parameter estimation and state estimation: an engineering approach using MATLAB.* J. Wiley, England, 2004.

[19] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. S. Karin, ed. *Proceedings of Supercomputing '95*, San Diego. ACM Press, New York, December 1995.

[20] R. A. Iles, A. N. Stevens and J. R. Griffiths. NMR Studies of metabolites in living tissue. *Progress in Nuclear Magnetic Resonance Spectroscopy*, 15(1-2):49-200, 1982.

[21] A. K. Jain, R. P. W. Duin, and J. Mao. Statistical pattern recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(1):4-37, 2000.

[22] I. T. Jolliffe. *Principal Component Analysis*, Springer-Verlag, New York, 1986.

[23] G. Karypis, V. Aggarwal, V. Kumar and S. Shekhar. Multilevel hypergraph partitioning: application in VLSI domain. *IEEE Transactions on VLSI Systems.* 7(1):69-79, 1999.

[24] K. Kira and L.A. Rendell. The feature selection problem: traditional methods and a new algorithm. In *10th National Conference on Artificial Intelligence*, pages 129-134, San Jose, California, June 1992.

[25] J. Kittler. Feature set search algorithms. In *Pattern Recognition and Signal Processing*, C.H. Chen, Ed., Sijthoff and Noordhoff, The Netherlands, 1978.

[26] R. Kohavi and G. H. John. Wrappers for Feature Subset Selection. *Artificial Intelligence*, 97(1-2):273-324, 1997.

[27] J. B. Kruskal and M. Wish. *Multidimensional Scaling*, Sage Publications, Beverly Hills, CA 1977.

[28] H. Liu and R. Setiono. A probabilistic approach to feature selection - a filter solution. *In Proceedings of the 13th International Conference in Machine Learning* pages 319-327, Bari, Italy, July 1996.

[29] H. Liu and L. Yu. Towards integrating feature selection algorithms for classification and clustering. *IEEE Transactions on Knowledge and Data Engineering*, 17(4):491-502, 2005.

[30] *Microarray Techniques*. Retrieved August 23, 2005, from the National Health Museum Website: http://www.accessexcellence.org/RC/VL/GG/nhgri_PDFs/microarray_technology.pdf

[31] U. R. Muller and D. V. Nicolau. *Microarray technology and its applications*. Springer, New York, 2005.

[32] P. M. Narendra and K. Fukunaga. A branch and bound algorithm for feature subset selection. *IEEE Trans. Computers*, 26(9):917-922, 1977.

[33] A. E. Nikulin, B. Dolenko, T. Bezabeh and R. L. Somorjai. Near-optimal region selection for feature space reduction: novel preprocessing methods for classifying MR spectra. *NMR Biomedicine* 11:209 – 216, 1998.

[34] M. Ouyang, M. Toulouse, K. Thulasiraman, F. Glover, and J. S. Deogun. Multilevel cooperative search for the circuit/hypergraph partitioning problem. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 21(6):685-694, 2002.

[35] M. Pirlot. General local search methods. *European journal of operational research*, 92:493-511, 1996.

[36] P. Pudil, J. Novovicova, and J. Kittler. Floating search methods in feature selection. *Pattern Recognition Letters* 15(11):1119-1125, 1994.

[37] S. J. Raudys and A. K. Jain. Small sample size effects in statistical pattern recognition: Recommendation for practitioners. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(3):252-264, 1991.

[38] W. Siedlecki and J. Sklansky. A note on genetic algorithm for large-scale feature selection. *Pattern Recognition Letters*, 10(11):335-347, 1989.

[39] R. L. Somorjai, B. Dolenko, and R. Baumgartner. Class prediction and discovery using gene microarray and proteomics mass spectroscopy data: curses, caveats, and cautions. *Bioinformatics* 19(12):1484-1491, 2003.

[40] R. L. Somorjai and A. Nikulin. The curse of small sample sizes in medical diagnosis via MR spectroscopy. In *Proceedings Twelfth Annual Scientific Meeting of the Society of Magnetic Resonance in Medicine*. New York, pages 685, August 1993.

[41] R. L. Somorjai, A. Nikulin, B. Dolenko, R. Baumgartner, and C. Bowman. *Class prediction from mass spectroscopy data.* A poster at the National Research Council's Institute for Biodiagnostics, Winnipeg.

[42] P. Somol, P. Pudil, F. J. Ferri, and J. Kittler. Fast branch and bound algorithm in feature selection. In *Proceedings Fourth World Multiconferenc in Systemics, Cybernetics, and Informatics* 7(1):646-651, 2000.

[43] P. Somol, P. Pudil, and J. Kittler. Fast branch and bound algorithm in feature selection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(7):900-912, 2004.

[44] D. D. Stark and W. G. William. *Magnetic resonance imaging, 2nd edition.* Mosby Year Book, St. Louis, 1992.

[45] S. Stearns. On selecting features for pattern classifiers. In *3rd International Joint Conference on Pattern Recognition*, pages 71-75, Coronado, California, November 1976.

[46] M. Stone. Cross-validatory choice and assessment of statistical predictions (with discussion), *Journal of the Royal Statistical Society B*, 36:111-147, 1974.

[47] M. Toulouse, K. Thulasiraman, and F. Glover. Multi-level cooperative search: a new paradigm for combinatorial optimization and an application to graph partitioning. In *Proceedings of the 5th International Euro-Par Conference on Parallel Processing*, pages 533-542, Toulouse, France, August 1999.

[48] C. Walshaw. A multilevel approach to the traveling salesman problem. *Operations Research* 50(5):862-877, 2002.

[49] C. Walshaw. A multilevel approach to the graph colouring problem. *Technical Report 01/IM69*, Computing and Mathematical Sciences, University of Greenwich, London, UK, 2001.

[50] L.A. Wosley. *Integer programming*. J. Wiley, New York, 1998.

[51] J. Xu. *An introduction to multilevel methods*. Oxford University Press, 1997.

[52] L. Yu and H. Liu. Feature selection for high-dimensional data: a fast correlation-based filter solution. *In Proceedings of the 20th International Conference in Machine* Learning, pages 856-863, Washington, D.C., August 2003.

[53] H. Zhang and G. Sun. Feature selection using tabu search method. *Pattern recognition* 35(3):701-711, 2002.

[54] L. Zheng and X. He. Classification Techniques in Pattern Recognition, In Proceedings of the *13th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision*, page 77, Plzen, January 2005.