Department of Computer Science

Hamilton, NewZealand

Using Output Codes for Two-class Classification Problems

Fanhua Zeng

This thesis is submitted in partial fulfilment of the requirements for the degree of Master of Science at The University of Waikato.

Abstract

Error-correcting output codes (ECOCs) have been widely used in many applications for multi-class classification problems. The problem is that ECOCs cannot be applied directly on two-class datasets. The goal of this thesis is to design and evaluate an approach to solve this problem, and then investigate whether the approach can yield better classification models.

To be able to use ECOCs, we turn two-class datasets into multi-class datasets first, by using clustering. With the resulting multi-class datasets in hand, we evaluate three different encoding methods for ECOCs: exhaustive coding, random coding and a "pre-defined" code that is found using random search. The exhaustive coding method has the highest error-correcting abilities. However, this method is limited due to the exponential growth of bit columns in the codeword matrix precluding it from being used for problems with large numbers of classes. Random coding can be used to cover situations with large numbers of classes in the data. To improve on completely random matrices, "pre-defined" codeword matrices can be generated by using random search that optimizes row separation yielding better error correction than a purely random matrix. To speed up the process of finding good matrices, GPU parallel programming is investigated in this thesis.

From the empirical results, we can say that the new algorithm, which applies multi-class ECOCs on two-class data using clustering, does improve the performance for some base learners, when compared to applying them directly to the original two-class datasets.

Acknowledgments

First of all, I would like to thank my supervisor Eibe Frank. This thesis could not exist without his generous and monumental support. He has always offered an open office door for me to discuss any ideas or questions. He has patiently endured my less acceptable academic writing and commented on every aspects of my thesis.

I would also like to thank all the people in the Department of Computer Science at the University of Waikato, especially the members of machine leaning group for providing such a supportive and fun environment . A Special thank gives Quan Sun and Jinjin Ma for their ideas and help.

My final but no less deserving thanks are reserved for the people outside university. I would like to thank Bruce, & Doesjka Trevarthen at LayerX. They have offered me an opportunity to work with such a great team and let me to have a very flexible working hours. Thanks go to my parents. They may not always understand what I do, but the encouragements are just the same. My wife Fen and my lovely boy Leyu are the source of happiness and I thank them for making my life the pleasure that it is.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Machine learning has been widely used in many applications, such as weather predictions, medical diagnosis and face detection. It is a field dedicated to finding ways to automatically extract information form data. It involves the design and development of algorithms that allow computers to learn and evolve behaviours based on empirical data.

The algorithms of machine learning allow one to make a prediction for a missing value in a dataset or for future data based on statistical principles. An important task of machine learning is classification, which assigns instances into one of a fixed set of classes. Classification learning involves finding a definition of an unknown function $f(x)$ (Dietterich & Bakiri, 1995) based on a training set consisting of known input/output pairs.

1.1 Multi-class classification

Unlike two-class classification, which assigns observations into one of 2 classes, the multi-class classification problem refers to assigning instances into one of k classes. Many publications have proposed methods for using two-class classifiers for multi-class classification because two-class problems are much easier to solve. The methods include decision trees, k-nearest neighbour classification, naive Bayes, neural networks and support vector machines (Witten & Frank, 2005). Nevertheless, decomposing multi-class problems into two-class problems is another approach, which may yield better performance. Based on decomposition, we can then use two-class classifiers for multi-class classification problems. There are several methods that have been proposed for such a decomposition. This thesis investigates empirically whether error-correcting output codes, one such method, can be profitably applied to two-class problems by artificially creating a multi-class problem using clustering.

Let us briefly review the basic decomposition methods here. They will be discussed in more detail in Chapter 2.

1.1.1 1-vs-all

1-vs-all (Rifkin & Klautau, 2004) is one of the simplest approaches to turn multi-class problems into two-class classification problems. Suppose we have k classes, then this method reduces the problem of classifying the k classes into k binary problems so that we have k binary learners for those k problems. The i^{th} learner learns the i^{th} class against the remaining $k-1$ classes, where we treat the i^{th} class as positive and the rest as negative.

At classification time, the classifier with the maximum output is considered the winner and we assign the class based on the classifier corresponding to this class. Even though 1-vs-all is very simple, Rifkin and Klautau (2004) point out that the performance of this approach is comparable to other complicated methods. I provided careful parameter tuning for the underlying learning algorithm is performed.

1.1.2 1-vs-1

1-vs-1 (Allwein & Shapire, 2000) is another simple strategy to convert multi-class classification problems into binary classification problems. In this approach, each class is compared to each other class and the remaining classes are ignored. A

	f_1	f_2	f_3	f_4	f_5	f_6	f_7
C_1	0	0	0	0	0	0	0
C_2	1	1	1	1	0	0	0
C_3	1	1	0	0	1	1	0
C_4	1	0	1	0	1	0	1
output	1	1	0	0	1	1	1

1: positive; 0: negative.

Table 1.1: An example codeword matrix

binary classifier is built to discriminate between each pair of classes. For k classes, it requires $\frac{k(k-1)}{2}$ binary classifiers.

At classification time, a voting is performed among the classifiers and the class that receives the maximum number votes wins. Allwein and Shapire (2000) state that this approach is generally better than 1-vs-all.

1.1.3 Error-Correcting Output Codes (ECOCs)

The ECOC (Sejnowski & Rosenberg, 1987) method was first pioneered by Sejnowski and Rosenberg in their well-known NETtalk system. It involves designing a distributed output code matrix M. Each class is assigned a unique binary string of length n. We call this string codeword.

The number of classes is the number of rows in the matrix M and the length of the binary strings n is the number of columns in M. Then n binary functions are learnt, where the i^{th} binary classifier learns the i^{th} column.

As an example, consider Table 1.1, which has 7 columns and 4 rows for a four-class problem. Note that each class has a unique codeword because each row is distinct. The columns can be chosen to be meaningful in a real world situation, but this is only possible if sufficient prior knowledge is available, and generally not the case. For example, in this thesis, we use all possible combinations of bits in the columns with the so-called exhaustive coding method. Regardless of the method used it is important that all columns are distinct to each other so that each binary leaner learns a unique problem.

To classify an instance for which we do not know the class value, the seven binary classifiers $f_1, f_2, ..., f_7$ are evaluated on this instance to obtain a 7 bits string, for example 1100111. We then compute the Hamming distance of this string to each of the 4 codewords (rows). The codeword (class) who has the smallest Hamming distance is considered as the class label and output as the classification for the original multi-class problem.

The process of mapping the output string to the nearest codeword (class) is called decoding. The way that we define the matrix is call encoding. In this thesis, we will discuss some encoding and decoding methods in Chapter 2 and 4.

The measure of the quality of the codeword matrix is the minimum Hamming distance between any pair of codewords (rows). Suppose d is the minimum Hamming, then we can correct up to $\frac{d-1}{2}$ bit errors. Note that for this method to work, the errors of the classifiers should be correlated as little as possible so that they do not all make a mistake for the same instance. As a minimum, all columns in the matrix need to be distinct, so that the learning problems are different. Therefore, a good codeword matrix should satisfy two aspects:

Row Separation

We maximize the minimum Hamming distance between any pair of codeword.

Column Separation:

- All columns are distinct.
- There is no inverse from one to another
- Every column is composed of 0s and 1s, not only 0s or 1s.

Note that this is the minimum requirement. Ideally, the Hamming distance between any pair of columns should also be as large as possible. How many bits of errors one can correct is given by row separation and how likely any two binary learners make similar mistakes is influenced by the column separation. We will cover more detail in Chapter 4.

1.2 Objective and motivation

We have discussed some multi-class classification strategies based on decomposition and we know that these techniques may yield high accuracy models, e.g., using ECOCs can often improve on applying a multi-class capable classifier directly to a multi-class dataset. However, we cannot use those techniques on two-class datasets without any further processing steps.

The objective of this thesis is to investigate whether ECOCs and similar methods can be profitably used for two-class classification problems. As ECOCs cannot be applied directly to two-class situations. The motivation of this thesis is to design and evaluate an approach to turn two-class problems into multi-class problems and then apply these multi-class techniques.

1.3 Thesis structure

Chapter 2 reviews the background on important concepts used in this thesis. It includes multi-class decomposition-based methods, i.e. 1-vs-all, 1-vs-1, ECOCs and ENDs. It also covers the machine learning algorithms that will be used in the experiments, such as C4.5, AdaBoost, Bagging and RandomForest.

Chapter 3 introduces clustering techniques that can be used for turning two-class datasets into multi-class ones, for example, simple k-Means and x-Means. This chapter also gives detail on how to use these techniques in experiments with real classification problems.

Chapter 4 describes several decoding and encoding methods. Decoding methods include Hamming decoding, inverse Hamming decoding and Euclidean decoding. Encoding methods includes exhaustive coding, random coding and ”pre-defined” code found using random search. It also has a detailed discussion on GPU programming that we used to find good ”pre-defined” code matrices using random search.

Chapter 5 presents the datasets used for evaluation and the results of the experiments.

Chapter 6 draws some conclusions and discusses possibilities for future work.

Chapter 2
Background

In machine learning, multi-class classification refers to assigning one of k classes to an input object. The classification of multi-class problems involves finding a function $f(x)$ whose range contains more than two classes. Compared to better understood two-class classification, which classifies instances into two given classes, multi-class classification is more complex and delicate. Many of the existing algorithms were originally developed to solve binary classification problems.

Some existing standard algorithms can be naturally extended to be used in multi-class classification settings. Various techniques have been proposed. They include decision trees (Quinlan, 1993), k-nearest neighbours (L. & et al., 1996), naive Bayes (Witten & Frank, 2005) and support vector machines (Barnhill & Vapnik, 2002). On the other hand, decomposing multi-class classification into binary classification is a possible approach, which can be universally applied, and which may yield better performance. Based on decomposition, multi-class classification can be solved using output labels or probability estimates of standard two-class classifiers.

Decision trees learning is a well-know and powerful classification algorithm that can be directly applied to multi-class classification. It has been used in the research presented in this thesis in conjunction with ensemble learning methods, on behalf of other standard classification techniques to compare to the decomposition-based methods. In particular, C4.5 (Quinlan, 1993), which is implemented using Java in WEKA as J48 (Witten & Frank, 2005), is used to build decision trees for bagged and boosted classifiers.

In this chapter, we will look at some of the existing algorithms that can deal with multi-class classification using decomposition. We will also look at the standard learning algorithms used for the experiments in this thesis: decision tree learning using C4.5, boosting, bagging and randomization.

2.1 Decomposition-based methods

Decomposing multi-class problems into binary problems is a popular way to deal with multi-class classification. In some case, the classification performance of these approaches is greater than that of applying a standard multi-class-capable learning algorithm directly, e.g. decision tree learning. In this section, we are going to review some of the existing decomposition-based methods.

2.1.1 1-vs-rest

A simple way to solve the problem of classifying k ($k \geqslant 3$) classes is to use the 1-vs-rest method (Rifkin & Klautau, 2004). k is the number of classes. In this approach k dichotomizers (i.e. two-class classifiers) are learnt for k classes, where each learner learns one class against the remaining $k-1$ classes. For this approach, we require n ($n = k$) binary classifiers. Each classifier treats the k^{th} class as positive and the remaining $k-1$ classes as negative.

For example, for four classes A, B, C and D, we have four learners f_1, f_2, f_3 and f_4, where learner f_1 learns class A against classes B, C and D, learner f_2 learns class B against classes A, C and D, and so on. See Table 2.1. The classifications in the training data are relabelled based on this scheme so that a standard learning algorithm can be used to learn four different classification models, each responses for identifying one class.

	class A	class B	class C	class D
f_1	1	0	0	0
f_2	0	1	0	0
f_3	0	0	1	0
f_4	0	0	0	1

1: positive; 0: negative.

Table 2.1: 1-vs-rest

When classifying a new instance, the class whose classifier produce maximum output, i.e. which is the most confident that the classification should be, is the winner and this class is assigned to the instance.

Although 1-vs-rest algorithm is simple, Rifkin and Klautau (2004) state that this approach provides performance that is comparable to other more complicated algorithms when the binary classifier is tuned well.

2.1.2 Pairwise classification

Pair-wise classification is also known as the 1-vs-1 algorithm or all-vs-all and it is another simple way to convert multi-class classification problems into binary classification problems. The pairwise classification algorithm requires that for a given numbers of classes, each possible combination of values for any pair of classes is covered by one classifier. Each binary classifier is built to discriminate between one pair of classes, while ignoring the rest of the classes. For k given classes, we thus have: $N = \sum_{i=1}^{k-1}$, where N is the number of required classifiers. Each classifier treats one class as positive, another class as negative and the rest of the classes are ignored.

Assume that we have a dataset with 4 ($k = 4$) classes, then we have $\sum_{i=1}^{4-1} = 6$ classifiers. The 6 corresponding two-class problems can be described as in Table 2.2.

	class A	class B	class C	class D
f_1	1	-1	0	0
f_2	1	0	-1	0
f_3	1	0	0	-1
f_4	0	1	-1	0
f_5	0	1	0	-1
f_6	0	0	1	-1

1: positive; -1: negative; 0: ignored.

Table 2.2: Pairwise

As we can see from Table 2.2, f_1 only learn two classes, A or B, for example. Classes C and D are ignored. At classification time, a voting is performed among all the binary classifiers and the class who receives the maximum number of votes wins.

In practice, the pair-wise approach is generally better than the 1-versus-rest approach, but it builds more classifiers than one-versus-rest algorithm(Allwein & Shapire, 2000).

2.1.3 ECOCs

Applying Error-Correcting Output Code (ECOC) is another technique that combines binary classifiers in order to address a multi-class problem. The classification error rate of a learning algorithm can be decomposed into a bias component and a variance component, and it is known that ECOC can reduce the bias and variance of the base classifiers. ECOCs have been successfully applied to a wide range of applications in machine learning (Aly, 2005) .

The ECOC method involves designing a code matrix M for a given multi-class problem and each column in the matrix represents a bit column for one base learner. Assume that there are n columns for a k-class problems. We then have n base learners. The i^{th} base learner aims to learn the i^{th} column respectively so that n binary classifiers are obtained in total. The outputs of these base learners can be

	f_1	f_2	f_3	f_4	f_5	f_6	f_7
C_1	0	0	0	0	0	0	0
C_2	1	1	1	1	0	0	0
C_3	1	1	0	0	1	1	0
C_4	1	0	1	0	1	0	1
output	1	1	1	0	1	1	1

1: positive; 0: negative.

Table 2.3: Example codeword matrix for four classes classification

used to distinguish between the k classes.

Consider Table 2.3 as an example to explain how ECOCs work. f_1, f_2, ..., f_7 are the base learners. C_1, C_2, ..., C_4 are codewords. The row labelled "output" is the predicted class of all base learners for a certain hypothetical test instance. In this case, there are 7 columns and 4 rows. The i^{th} column is a learning scheme for the i^{th} base learner. Each row represents one codeword corresponding to one class in the training dataset.

At classification time, when a new instance comes in, we use those 7 base learners to predict the class label. In our example, we receive the string 1100111. We then calculate the Hamming distance between each row and the output string 1100111. The row that has minimum Hamming distance to the output string is the predicted class label. In this case, class C_2 has the smallest Hamming distance, namely 1. Therefore, we assign the new instance to class C_2.

In this example, the assumption is that classifier f_4 made a mistake, which can fortunately be corrected because the code matrix allows for this. The number of bits that can be corrected depends on the minimum Hamming distance between each pair of rows in the matrix. Details on this, and methods for designing good ECOC matrices will be discussed in Chapter 4 of this thesis.

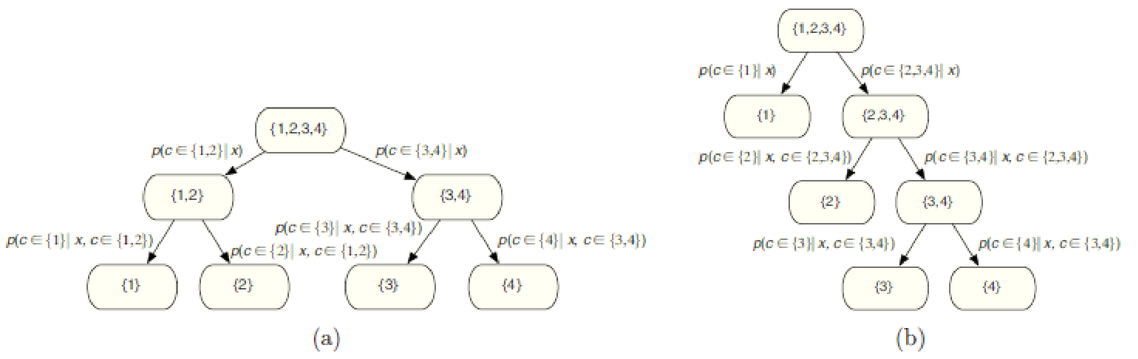


Figure 2.1: Two different systems of nested dichotomies, reproduced from END(Frank and Kramer, 2004)

2.1.4 END

Ensembles of Nested Dichotomies (END) were proposed by Frank and Kramer (2004). It is method that uses a tree structure to decompose a multi-class problem into binary classification problems. A nested dichotomies system randomly recursively splits a set of classes from multi-class classification into smaller and smaller subset (Frank & Kramer, 2004).

Nested dichotomies can be described as binary trees. A root node contains all classes and a leaf node only contains one class. At each node, the tree divides the set A into two subsets B and C that contain all the classes in set A. There are different possible ways to split a node. Figure 2.1 shows an example of two possible ways of constructing the binary trees for a four-class problem.

Classifiers are learned for the internal nodes of the tree. The estimated class probability distribution for the original multi-class problem can be obtained by multiplying the the probability of all the internal nodes that need to be visited to reach the leaf. For example, the probability of class 4 for an instance x is given by:

$p(c = 4|x) = p(c \in 3, 4|x) \times p(c \in 4|x, c \in 3, 4)$, based on Figure 2.1a, and

$p(c = 4|x) = p(c \in 2, 3, 4|x) \times p(c \in 3, 4|x, c \in 2, 3, 4) \times p(c \in 4|x, c \in 3, 4)$ based on Figure 2.2b

Both trees are valid class probability estimators. However, the estimates obtained from different trees are normally different. Frank and Kramer state that there are $3^n 2^{n+1} - 1$ possible two-class problems for an n-class dataset. In this four-class example, therefore there are 25 different ways to build the binary trees. This number grows extremely quickly since the term 3^n arises. It becomes a problem that it is impossible to generate all trees exhaustively. Hence, Instead of doing an exhaustive search, the END method evaluates the performance of ensembles of randomly generated trees, where probability estimates from different trees are simple averaged.

According to Frank and Kramer, ensembles of nested dichotomies produces more accurate classifiers than applying C4.5 to multi-class problem directly. Frank and Kramer also point out that this approach produces more accurate classification models than pair-wise algorithm if both techniques are applied with C4.5. Compared with error-correcting output codes, it has similar performance.

2.2 Tree and ensemble learning

In this section, we review the algorithms that are used in our experiments, namely C4.5(Quinlan, 1993) , AdaBoost (Freund & Schapire, 1995), Bagging (Breiman, 1996) and RandomForest (Ho, 1995). C4.5 is used as the base learner of AdaBoost and Bagging. AdaBoost is a popular and efficient boosting algorithm and it combines many "weak" learners as one "strong" learner. Bagging is a short word for bootstrap aggregating. RandomForest is an ensemble meta-algorithm that consists of many decision trees that are generated using a partially randomized decision tree learner.

2.2.1 C4.5

C4.5 (Quinlan, 1993) is one of the most well-known decision tree algorithms. It is an extension of the earlier ID3 algorithm also developed by Ross Quinlan. J48 is an open source Java implementation of C4.5 in the WEKA data mining tool (Witten & Frank, 2005). Since J48 is used as the based learner for most of the experiments in this thesis, it is very important to understand the approach that it uses to construct a decision tree as well as the strategy of pruning the tree.

Constructing the decision tree

C4.5 uses "divide and conquer" to build a decision tree using dataset T. There are three possibilities of constructing a decision tree from a set T of training data of k classes $\{C_1, C_2, ..., C_k\}$ (Quinlan, 1993):

- T contains one or more instances that have the same class C_j. In other words, all the instances belong to a single class C_j, and a leaf node should be created for them..

- T contains no instance. In this situation, the decision tree is a leaf. The class of the instances in the leaf must be determined from information other than T. C4.5 uses the most frequent class at the parent of this node.

- T contains instances that belong to a mixture of classes. In this case, the decision tree is not a single leaf normally. The idea is to split T into subsets of instances. C4.5 chooses a test that is based on a single attributes to generate mutually exclusive outcomes $\{O1, O2, ..., On\}$. T is then split into subsets $T1, T2, ..., Tn$, where subset T_i has all instance in T that have outcome O_i.

Constructing a decision tree can be done recursively based on the above three possibilities. The basic algorithm is to select an attribute to use at the root node to split the instances. For each possible values of the attribute, we create a branch and then assign all the instances to the different branches. We repeat the process for each branch using only the instances that reach the branch. We stop the process when all instances in one branch have the same classification or where there are no instances left.

Now the question is what attribute we select when splitting instances. Information gain is widely used. Let $Attr$ be the set of all attributes and T be the set of all training instances. $value(x, a)$ with $x \in T$ defines the value of a specific instance x for attribute $\alpha \in Attr$, and $H(\mathrm{s})$ specifies the entropy of the class distribution in subset S of T. The information gain for a given attribute $\alpha \in Attr$ is calculated as follows:

$$InformationGain(T, a)$$

$$= H(T) - \sum_{v \in value(a)} \frac{|\{x \in T | value(x,a) = v\}|}{|T|} H(\{x \in T | value(x, a) = v\})$$

We calculate the information gain for all attributes and choose the one that gains the most information to split on.

Pruning the decision tree

If the decision tree overfits the training data, the performance will get worse. Hence it is important that we prune the tree to produce a simpler tree that is more robust with respect to variance in the training data. There are two basic ways that can be used to modify the tree: prepruning (or forward pruning) and postpruning (or backward pruning).

Preprunning involves trying to decide not to divide a set of training instances any further. Preprunning is more efficient because time is not wasted on assembling structure that is not used in the final tree. However, we need to have a measure to stop splitting a subset. The measures that have been used include information gain, error reduction or statistical significance (Witten & Frank, 2005). For example, in a subset, if the assessment is smaller than a particular threshold value, the division is stopped. However, Breiman (Breiman & et al., 1984) point out that it is not always easy to find the right stopping value. If the threshold is too high, it reduces the accuracy due to underfitting, while if the threshold is too low, the tree may overfit the data.

Postpruning builds the complete trees first and then remove some of the structure. The C4.5 decision tree algorithm uses postpruning to prune trees. Quinlan (1993) states that preprunning is quite satisfactory but uneven in some domains. The postpruning process is to develop a completed and overfitted tree and then prune the tree. Trees are normally pruned by replacing one or more sub trees with leaves. The class of the leaf can be determined by examining the training instances covered in the leaf and identifying the most frequent class. Subtree replacement works from the leave nodes back up toward the root node. This approach is quite simple. First, replace the child nodes with a single leaf node. Then continue to work back from the leaves. We prune the tree until the decision is made not to.

Subtree raising is more complex and is also used by C4.5. With the subtree raising operation, we raise the subtree and replace its parent node. Then we reclassify the instances in the other branch of the parent node into one of the leaf nodes in the raised subtree. The general procedure is the same as for subtree replacement, we prune the tree until the decision is made not to

These two pruning methods require a decision whether to replace an internal node with a leaf for subtree replacement, and whether to replace an internal node with one of the nodes below it for subtree raising. To achieve this, we need to estimate the error at the internal node and the leaf nodes. The decision can be made by comparing the estimate error between the un-replaced/un-raised trees and replaced/raised subtrees. C4.5 uses the upper limit of a confidence interval for the error on the training data as the error estimate.

2.2.2 Adaboost

Adaboost, short for Adaptive Boosting, is a machine learning algorithm that constructs a "strong" classifier as a linear combination of "simple" and "weak" classifiers. It was formulated by Yoav Freund and Robert Schapire (1995). Adaboost is one of the most popular machine learning algorithms. The idea is quite intriguing: It generates a set of weak classifiers and simultaneously learns how to linearly combine them so that the error is reduced. The result is a strong classifier built by boosting the weak classifiers. Therefore, AdaBoost can be used in conjunction with many other machine learning learning algorithms to improve the accuracy of learning models. The algorithm of AdaBoost is shown in Figure 2.2.

First we initialise m all training instances to have equal weight. In each iteration of the algorithm, based on the current weighted version of the data, we learn a classifier C_k. Then we increase the weight of training instances if they are misclassified by classifier C_k and decrease an instance's weight if it is correctly classified by C_k.

The weight for iteration $k+1$ are calculated as follows:

$W_{k+1}(i) = \frac{W_k(i)e^{-\alpha_k y_i C_k(x_i)}}{Z_k}$, where

$\alpha_k = \frac{1}{2}ln[\frac{(1-E_k)}{E_k}]$, and

$Z_k = \Sigma_{i=1}^{m} W_{k+1}(i)$.

Note the following:

- The class value y_i of training instance x_i is assumed to be either -1 or 1, and this also holds for the classification $C_k(x_i)$.

- The E_k error is calculated based on the summation and normalization of all wrongly classified weighted training instances by the weak learner C_k. The weak learner should be better than random guess $E_k < 0.5$.

- The measurement α_k measures the importance assigned to C_k. Note that $\alpha >= 0$ if $E <= \frac{1}{2}$ and α gets larger when E gets smaller.

- Z_k is a normalization factor so that W_{k+1} will be a distribution.

At classification time, the classifiers C_k are linearly combined using the importance factor α_k. Therefore, for any given input training dataset, we can describe the final classifier as:

$$H(x) = sign(\textstyle\sum_{k=1}^{K} \alpha_k C_k(x))$$

AdaBoost often produces classifiers that are significantly more accurate than the base learner (Witten & Frank, 2005), and it does not require prior knowledge of the weak learner. The performance is completely dependent on the learner and the training data. Note that AdaBoost can identify outliers based on their weight. It is susceptible to noise with very large number of outliers. In practical situations, it can sometimes generate a classifier that overfits the data and produce a significantly less accurate one than a single weak learner.

```
Initialize
    Dataset : $D$={ $x_1$ , $y_1$ ;...; $x_m$, $y_m$ };
    Iterations: $K$;
    Weight: $W_1(i) = \frac{1}{m}, i = 1, ..., m$, where $m$ is number of instances.
              Assign equal weight to each training instance.
Iterations
    For $k = 1$ to $K$
              Train weak learner $C_k$ using weighted dataset $D_k$ sampled
              according to $W_k(i)$.
              Compute error $E_k$ of the model based on dataset $D_k$.
              For each instance in dataset:
                  If instance classified correctly by model:
                      Decrease weight of training instance.
                  If instance misclassified correctly by model:
                      Increase weight of training instance.
              Normalise weight of all instances.
```

Figure 2.2: Adaboost Algorithm

2.2.3 Bootstrap aggregating (Bagging)

Boosting and bagging (Breiman, 1996) both adopt a similar approach that combines the decisions of different models to create a single prediction. But they derive the individual models in different ways. In boosting, we modify the weight of instances according to their classification based on whether it is correct or not, while in bagging, all models receive instances of equal weight but differently sampled datasets.

Bagging is a meta-algorithm that uses several training datasets of the same size to improve stability and classification accuracy. The training datasets are randomly chosen from the original training data. The algorithm is described in Figure 2.3.

For a given dataset D size of n, bagging generates m new training datasets D_i of the same size. Each new dataset D_i is generated by sampling examples from D with replacement. By sampling with replacement, it is likely that some instances may be chosen more than once. Statistically, set D_i is expected to have 63.2% unique examples of D, the rest being duplicates. The m models are built using the m new

```
model generation
Let $n$ be the number of instances in the training data,
For each of $t$ iterations
    Sample $n$ instances with replacement from training data.
    Apply the learning algorithm to the sample
    Store the resulting model

classification
For each of the $t$ models:
    Predict class of instance using model.
Return class that has been predicted most often
```

Figure 2.3: Bagging Algorithm

training dataset.

Bagging reduces variance and helps to avoid overfitting. It helps most if the underlying learning algorithm is unstable, which means a small change in the input data can lead to very different classifiers, since the classification of Bagging is obtained by averaging the output or by voting. Because it averages several predictors built from similar training datasets, bagging does not improve very stable algorithms like k-nearest neighbours.

2.2.4 Random forest

A random forest is also an ensemble meta-algorithm and consists of many decision trees. The term random forest comes from the term random decision forest, which was proposed by Tin Ho of Bell Labs in 1995. It combines the bagging idea and the random selection of features in order to construct a collection of decision trees.

The selection of a random subset of attributes is an example of the random subspace method that is also called attribute bagging (Ho, 1995). In random forest, a different random subspace is chosen at each node of a decision tree. A standard attribute selection criterion such as information gain is then applied to choose a

```
model generation
Let $N$ be the number of instances in the training data, and
    $M$ be the number of attributes of the instance.
For each tree:
    A $m$ Training dataset is chosen by randomly selecting $n$ ($n < N$) time from all $N$
        training instances with replacement like in bagging.
    Use number of attributes to determine the decision at a node of the tree,
        where $m$ should be much smaller than $M$ ($m < M$).
    The rest of the instances are used to estimate the error of the tree.
    For each node of the decision tree, randomly choose $m$ attributes, and then
        calculate the best split based on these $m$ variables in the training set.
    All individual trees are fully grown and not pruned

classification
Iterate over all trees in the ensemble;
and the average vote of all trees is the prediction of random forest
```

Figure 2.4: Random forest algorithm

splitting attribute based on this subspace. For each individual decision tree, the algorithm that is used for constructing trees is described in Figure 2.4.

The random forest algorithm is one of the most popular learning algorithms. In practice, it often produces highly accurate classifiers. Random forest is also very efficient regarding running time. It can handle large input data with very high dimensionality. The disadvantage of random forest is that it overfits some datasets with noisy classification (Ho, 1995).

Chapter 3

From two to many classes using clustering

The goal of this thesis is to investigate whether Error-Correcting Output Codes (ECOCs) and similar methods can be profitably used for two-class classification problems. However, an immediate obstacle is that ECOCs cannot be applied directly to two-class situations. The problem is that we can correct only $\frac{d-1}{2}$ (rounded down) errors with the ECOC prediction scheme when the row separation (he minimum Hamming distance between pairs of rows) is d. Suppose that there are k classes. With exhaustive ECOCs, which deliver maximum possible error correction, and which will be discussed in detail in the next chapter, the number of columns in the code matrix is $n = 2^{k-1} - 1$. Moreover, the pairwise Hamming distance between rows is $d = 2^{k-2}$. This means we can correct up to $x = 2^{k-3} - 1$ bit errors. Therefore, we need to have at least $k \geqslant 4$ classes to be able to correct 1 bit error. In that case, we can still get the correct classification even if one base learner misclassifies an example . In contrast, here is no guarantee that we can get the correct classification if one of the base learners makes an incorrect decision in a situation with less than four classes ($k < 4$). Therefore, we cannot the apply ECOC algorithm on 2 or 3 class-classification problem directly.

To be able to apply ECOCs on two-class or three-class datasets we need to develop an algorithm to turn the problem into a situation where there are more than 3 classes ($k > 3$). The basic idea of our approach is quite simple. With the binary-class datasets in hand, we first transfer them to multi-class ones by creating clusters within each class, and then we can apply ECOCs on the transformed multi-class dataset at training time. At classification time, we then transfer the output of the ECOCs models back to one of the original binary classes. To be able to transfer the ECOCs' output to the final classification, we have to keep a look-up table to store the reference of which clusters were generated from which class. By transferring the binary dataset to a multi-class one and then applying ECOCs to it, we can hopefully improve the accuracy of classification models.

In this chapter we are going to look at some existing techniques, namely clustering and k-means, as well as our approach in detail. When we review the existing clustering techniques, we focus on how they cluster instances and the discussion of the parameter settings. Our approach includes how to use these techniques to turn two-class problems into multi-class classification problems in order to be able to apply ECOCs algorithm indirectly. We also list the detail of how we transfer the binary-class dataset to a multi-class dataset at training time and why this approach can potentially improve the performance of the classification models. Note that three-class problems can be dealt in the same way as two-class ones so that we only consider two classes as our example here to explain the process and the algorithms.

3.1 Clustering

Clustering is an existing technique that can be used to group instances in machine learning. It is a simple and straightforward approach that has been used for many decades. Clustering techniques are normally applied when there is no class that needs to be predicted but rather when the instances are to be divided into natural groups. We will use the idea of this approach to group instances in our algorithm.

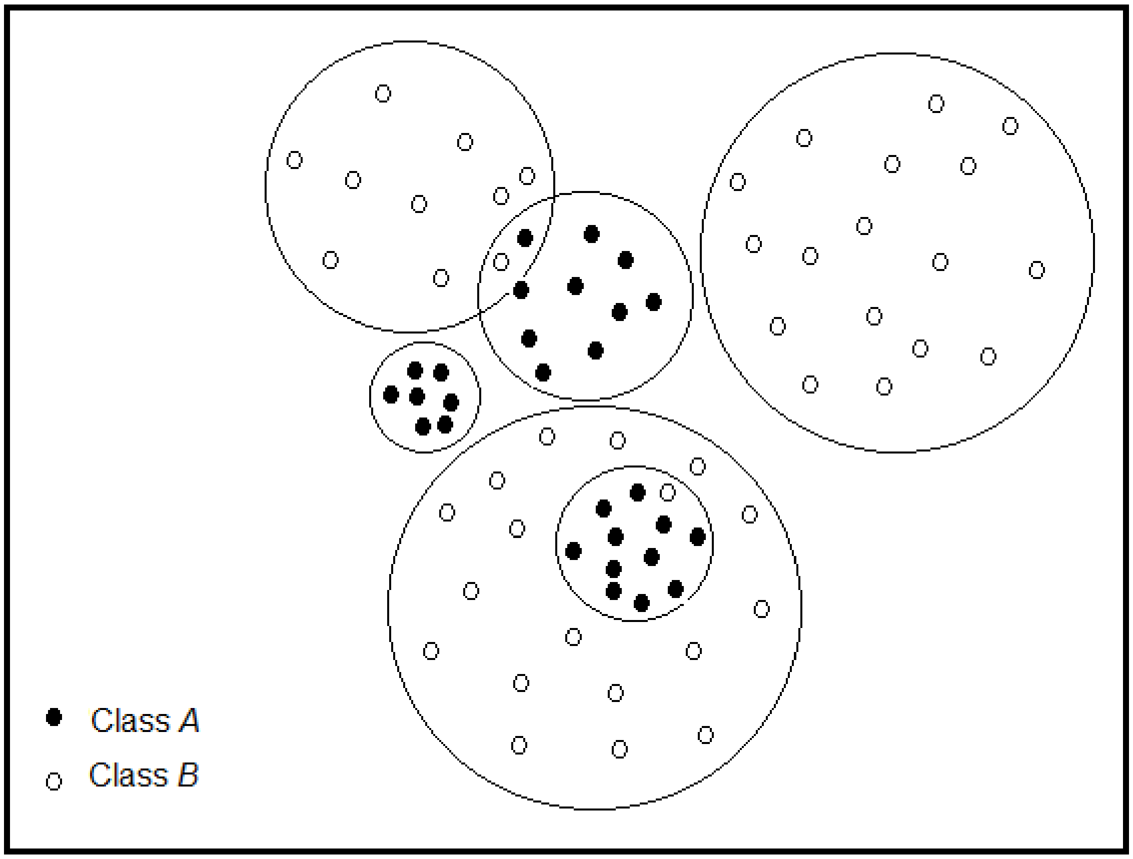


Figure 3.1: An example of clustering expression

There are many different ways that can be expressed to cluster instances. Figure 3.1 show an example of a clustering expression, where data from 2 classes has been split into three clusters each, to create a six-“class” problem. There are a few possible different situations. First, the clusters may be overlapping so that an instance will fall in several groups. The clusters may be exclusive in which case an instance only belongs to one group. There are also some other situations where they may be probabilistic or they may be hierarchical.

The experiments in this thesis are based on the WEKA software. There are several different clustering algorithms that are available in WEKA, such as simple k-Means (Witten & Frank, 2005) and x-Means (Witten & Frank, 2005) . k-Means is a classic clustering techniques and is simple and effective. x-Means is k-Means extended by an method to automatically find the appropriate number of clusters. Instead of generating a fixed number of clusters, x-Means attempts to split a cluster into sub-clusters. In x-Means, the decision between the children of each cluster and itself is done by comparing the Bayesian Information Criterion (BIC) values of the two structures.

For the experiments in this thesis, we want to see how different numbers of clusters affect our final classification models. Hence simple k-Means is used to cluster instances in the research presented here, and in this chapter, we only look at k-Means in detail.

3.2 k-Means

k-Means is a classic clustering algorithm and it uses a distance function like instance based learning algorithms, for example IBK in WEKA. It is an algorithm that is very popular and easy to understand. k is a parameter that we need to specify to indicate how many clusters we want to build. After we build the k clusters using the given dataset, when an instance needs to be clustered using k-Means at clustering time, it assigns the instance to its nearest cluster (one of those k clusters). By nearest cluster, we mean the cluster that has the smallest Euclidean distance from its mean to the new instance. The output of k-Means consists of k groups of instances.

3.2.1 The algorithm

The training algorithm of k-Means is quite simple. Firstly, we need to specify how many clusters k we want, as mentioned above. Then k-Means chooses k initial points as the cluster centres randomly. All instances in the dataset are assigned to their closest cluster centre using ordinary Euclidean distance as the measure. Next, we calculate the centroid (mean) of the instances in each cluster. Those centroids are used as the new centre values for their respective clusters. All processes are repeated with the new cluster centres until the same points are assigned to each cluster in consecutive rounds. Finally, k-Means converges to a stabilized cluster centres and these centres will remain the same (Witten & Frank, 2005) .

k-Means is effective. Choosing the cluster centres to be the centroid minimizes the total squared distance from each of the cluster's points to its centre. Once all cluster centres have stabilized, all instances are assigned to their closest cluster centres. The overall effect is to minimize the total squared Euclidean distance from all instances to their cluster centres. According to Witten and Frank the minimum is a local one and there is no guarantee that it is global minimum. Different initial cluster centres can lead very different final cluster models. In other words, a completely different clustering can arise when there is a small change in the initial random choice. To address the problem, we can run the algorithm several times with different initial choice to find a good final cluster arrangement. The final chosen result is the one with the smallest total squared Euclidean distance.

There are two issues that we may consider with k-Means. Firstly, k-Means may fail to find good cluster arrangements. This problem can be solved by running k-Means several times. Secondly, there is processing time required for finding the k cluster centres. We can borrow the ideas of kD-Trees (Witten & Frank, 2005) and *ball*-Trees (Witten & Frank, 2005) that are used in instance based learning algorithms. They are faster distance calculating algorithms. However, the simple k-Means clustering algorithm in WEKA does not use these two techniques by default and using these more sophisticated techniques was not necessary for the experiments presented in this thesis.

3.2.2 Choosing the parameter k (the number of clusters)

The parameter k indicates how many initial cluster centres we want to have as well as the number of final clusters. As the experimental results presented in this thesis will show that it is important to specify an appropriate value to it. In particular, when we transfer a two-class dataset into a multi-class dataset, k is the coefficient that is used to multiply the number c of classes in the original dataset ($c = 2$ in binary data). The number of classes in the new transferred multi-class data can be defined as follows:

$N = k \times c$, where

N is the number of new classes in the transferred dataset.
k is the number of clusters.
c is the number of classes in the original dataset.

Different values of k can generate different numbers classes in the newly generated multi-class dataset. For example, let:

$k = 3$ and $c = 2$.

Then we have

$N = k \times c = 6$ classes

Since $N = 6$ and this is greater than 4, the ECOC method can then successfully be applied to this new dataset. We are going to dive into more detail on this in Section 3.3. We know that k is an important parameter as different values of k lead to different codewords matrices. In principle, the larger the k value is, the more errors we can correct using the ECOC method. In practice, however, finding a good codeword matrix is becoming very costly when the value of k gets larger. In the experiments presented in this thesis, the value of k is in the range $k \in \{2, 15\}$. We will discuss how to find good codeword matrices in Chapter 4 .

3.3 Turning two-class problems into multi-class ones

Because we can only apply ECOCs on multi-class classification problems and more specifically ECOCs require that the dataset has at least four classes to be able to correct at least one bit error, it is necessary to transfer two-class data into multi-class data to be able to apply ECOC at training time. At classification time, we change the obtained classification back to corresponding classes in the two-class dataset.

3.3.1 Creating a multi-class dataset

When we apply ECOC algorithms on binary class problem, clustering (k-Means) is a natural way to turn the two-class dataset into a multi-class dataset. To keep the problem simple, we use the same number of clusters for each class in the dataset.

Suppose we have two classes A and B in our binary dataset T and the distribution of dataset T is shown in Figure 3.2.

If we specify 3 as the number of clusters for each class for example, we will end up with a 6-class dataset. The new classes are:

$A'_1, A'_2, A'_3, B'_1, B'_2$ and B'_3, where

A'_1, A'_2 and A'_3 are generated from A, and

B'_1, B'_2 and B'_3 are generated from B.

The new class label of an instance in the new dataset T' will be one of these six classes. The new clustered dataset T' will be our training dataset. The new

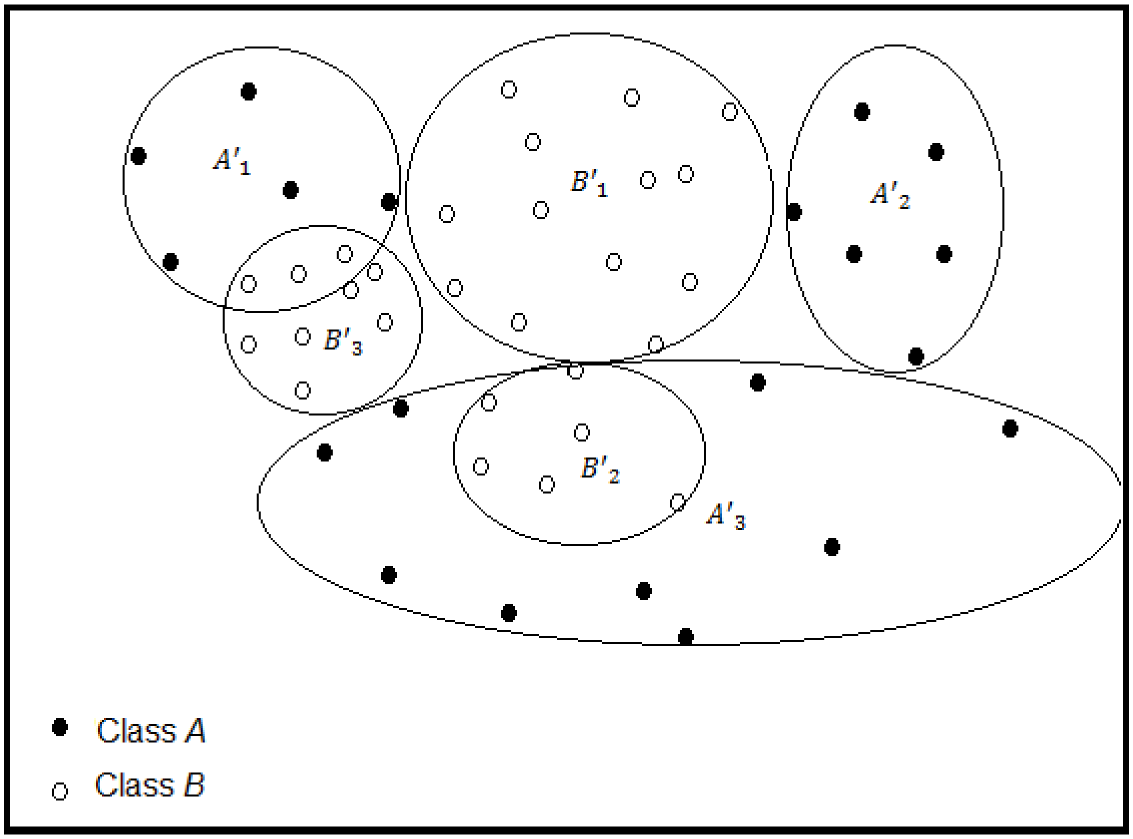


Figure 3.2: An example of 3 clusters per class for a two-class dataset

dataset T' can be generated using the algorithm described in Figure 3.3, assuming the clustering technique has already been applied to find the k clusters per class, which can be done by running k-Means on the data of each class separately.

The created dataset T' has the same number of instances as original dataset T. There are also the same number of attributes and the same attributes values. The only difference is that there are more classes in T' than T. More specifically, the number of classes in T' is the product of the number of classes in T and the number of clusters k we specified when we created T' using clustering. Therefore, the more clusters (larger k values) we provide, the more classes we will end up with in our new dataset T'. Figure 3.4 shows an example of T and T' with $k = 2$.

With the dataset T' in hand, we can apply ECOCs algorithm to it. This is

Steps:

1. Build instances set T' with empty data in it that has the same structure as T but $k \times 2$ class values.
2. Iterate over all instances in the original binary class dataset T.
3. For each instance i in T:
4. Create a new instance i' with attributes in which values are copied from the instance i.
5. Find the closest cluster center for instance i and get the new class value (one of the clusters, for example, A'_1, A'_2 or A'_3, when the original class is A). Assign the cluster value to the new instance i as its new class label.
6. Store the new instance i' in T'.
7. Repeat step 4, 5, 6 until iteration is finished.

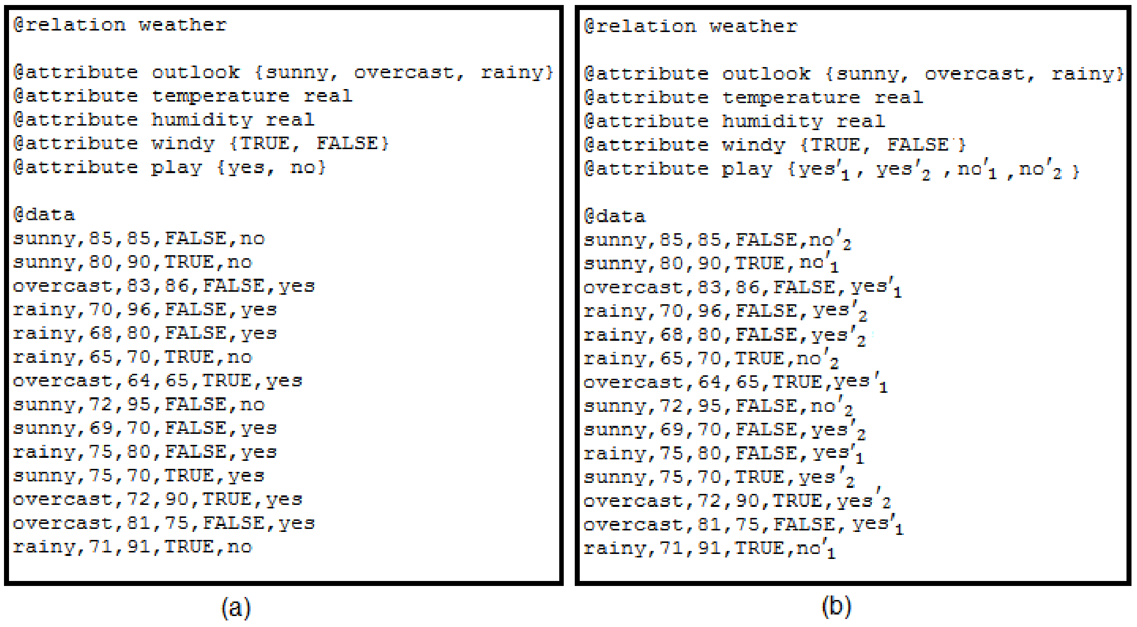
Figure 3.3: The process of creating a multi-class dataset

the key: we always turn the binary-class into a multi-class problem classification before using ECOCs. So rather than using dataset T, we are using dataset T' as our training data to build classification models. The prediction of those models will be one of these classes in dataset T'. In fact, the classification is one of the clusters. In the next two sections, we are going to look at how we can use these classification models to produce the final output.

3.3.2 Look-up table

When we transfer the binary-class dataset T into a multi-class dataset T', we do not want to lose the connection between T and T'. Therefore, we create a look-up table to keep a reference of the classes in T and T'. Figure 3.4 is an example of a look-up table with i clusters.

This look-up table is very important even though it is not necessary to have it at training time: we do not need to worry about this table when we build classification models. It will only be used for producing the final binary classification. At the time of outputing the final decision, the classification models based on T' will only



```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE}
@attribute play {yes, no}

@data
sunny,85,85,FALSE,no
sunny,80,90,TRUE,no
overcast,83,86,FALSE,yes
rainy,70,96,FALSE,yes
rainy,68,80,FALSE,yes
rainy,65,70,TRUE,no
overcast,64,65,TRUE,yes
sunny,72,95,FALSE,no
sunny,69,70,FALSE,yes
rainy,75,80,FALSE,yes
sunny,75,70,TRUE,yes
overcast,72,90,TRUE,yes
overcast,81,75,FALSE,yes
rainy,71,91,TRUE,no
```

(a)

```
@relation weather

@attribute outlook {sunny, overcast, rainy}
@attribute temperature real
@attribute humidity real
@attribute windy {TRUE, FALSE }
@attribute play {yes'1 , yes'2 , no'1 , no'2 }

@data
sunny,85,85,FALSE,no'2
sunny,80,90,TRUE,no'1
overcast,83,86,FALSE,yes'1
rainy,70,96,FALSE,yes'2
rainy,68,80,FALSE,yes'2
rainy,65,70,TRUE,no'2
overcast,64,65,TRUE,yes'1
sunny,72,95,FALSE,no'2
sunny,69,70,FALSE,yes'2
rainy,75,80,FALSE,yes'1
sunny,75,70,TRUE,yes'2
overcast,72,90,TRUE,yes'2
overcast,81,75,FALSE,yes'1
rainy,71,91,TRUE,no'1
```

(b)

Figure 3.4: weather.arff dataset: (a) two-class , (b) transferred multi-class

$$\underbrace{A'_0, A'_1, \cdots, A'_i}_{A}\ \underbrace{B'_0, B'_1, \cdots, B'_i}_{B}$$

Figure 3.5: Look-up table

predict the classes in T', for example, A'_2, A'_3 or etc. This look-up table holds a reference so that we can transfer the classification A'_1 or A'_3 back to the original A later.

3.3.3 The classification

We have already talked about how we turn the two-class dataset T into a multi-class dataset T' so that we apply ECOCs to T' in order to improve the accuracy of the classification models. This is the first step: we have created the required training dataset (with at least four classes to be able to correct at least one bit error).

With the dataset T', we can use whatever base learner in ECOCs to learn classification models. As we mentioned before, these models will produce one of the clusters, which is not the final output. We need to have a way to transfer the output of ECOC models to be one of the binary classes in the original dataset T. After we have transferred these clusters into classes, we will get the final classification.

The algorithm for transferring the output of the ECOC model to the final classification can be very simple. Assume that the output of the ECOC model is A'_1, one of the classes in the generated multi-class dataset. We simply assign the final predicted class to class A. The reason is that A'_1 is one of the clusters split from A. This process is done using the reference in the look-up table that we created when we generated the training dataset.

3.4 Overview of the clustering approach

We have discussed the process for turning two classes to many classes using clustering. It is useful to draw all these process in one graph so that we can easily navigate and understand. Figure 3.6 shows the whole process.

We summarise the process using the following steps:

- 1. Use clustering approach to transfer the binary-class data into a multi-class one;
- 2. Save a look-up table to store the reference;
- 3. Learn classification models using ECOC method;
- 4. Classify instance;
- 5 and 6 Use the look-up table to produce the final class label.

Note that we only consider the case where we cannot use ECOCs directly, i.e., binary datasets and three-class datasets. Actually this approach is not limited to two-class and three-class problems. It is available to be applied on multi-class data

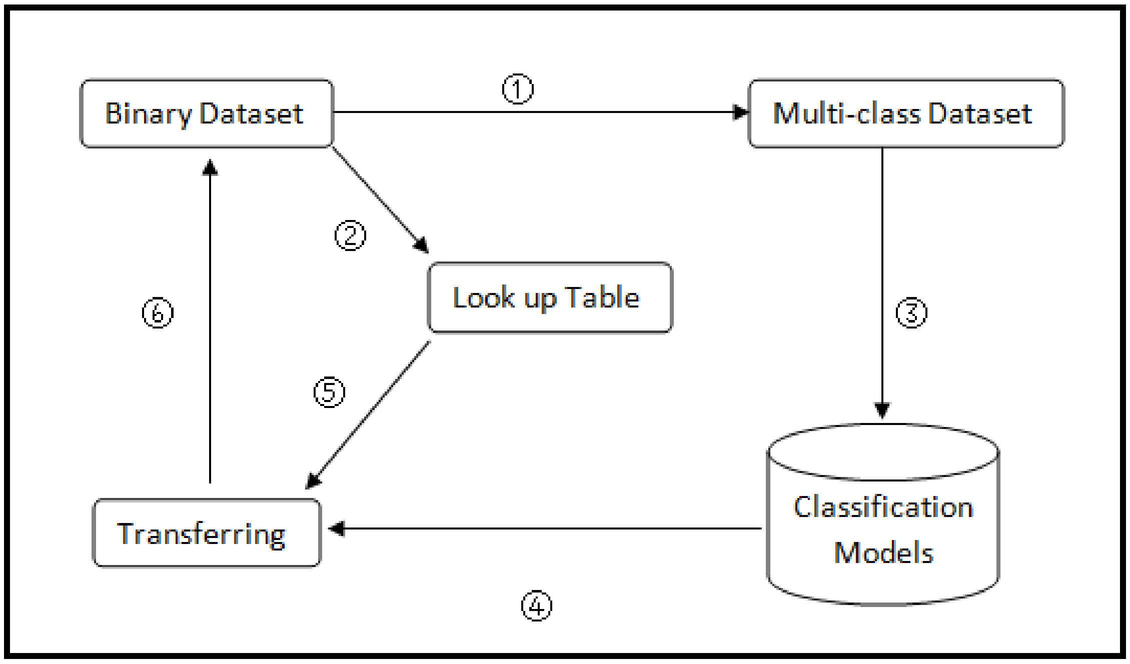


Figure 3.6: The process of applying ECOCs to a binary-class dataset

if we want to have more classes in the training set. However, this scenario is beyond the scope of this thesis.

Chapter 4

Generating and using ECOCs

It is known that the lowest error rate is not always reliably achieved by applying a single classifier for some classification problems. That is the reason why using an Error Correcting Output Code (ECOC) was proposed as a combination of binary problems to address multi-class problems. We have discussed some background of ECOCs in Chapter 2 and we know that the ECOC technique is widely applied in many applications. In this chapter, we consider the principle and the usage of ECOCs.

The ECOC technique involves two distinct stages: encoding and decoding. For a given set of classes, the encoding method builds a codeword for each class. The decoding process uses the codeword matrix to produce an output code. The output codeword string can be used to make a classification decision for a given test sample.

4.1 Encoding method

At the encoding stage, suppose we have k classes that need to be learnt for a given dataset T, then n different learners are trained in the ECOC ensemble. In other words, n dichotomizers need to be trained. A codeword of length n is obtained for each class, where the i^{th} bit of the code corresponds to the i^{th} dichotomizer. The code is composed of 0s and 1s for binary problems.

Arranging the codewords as rows of a matrix, we have a matrix M, where $M \in$

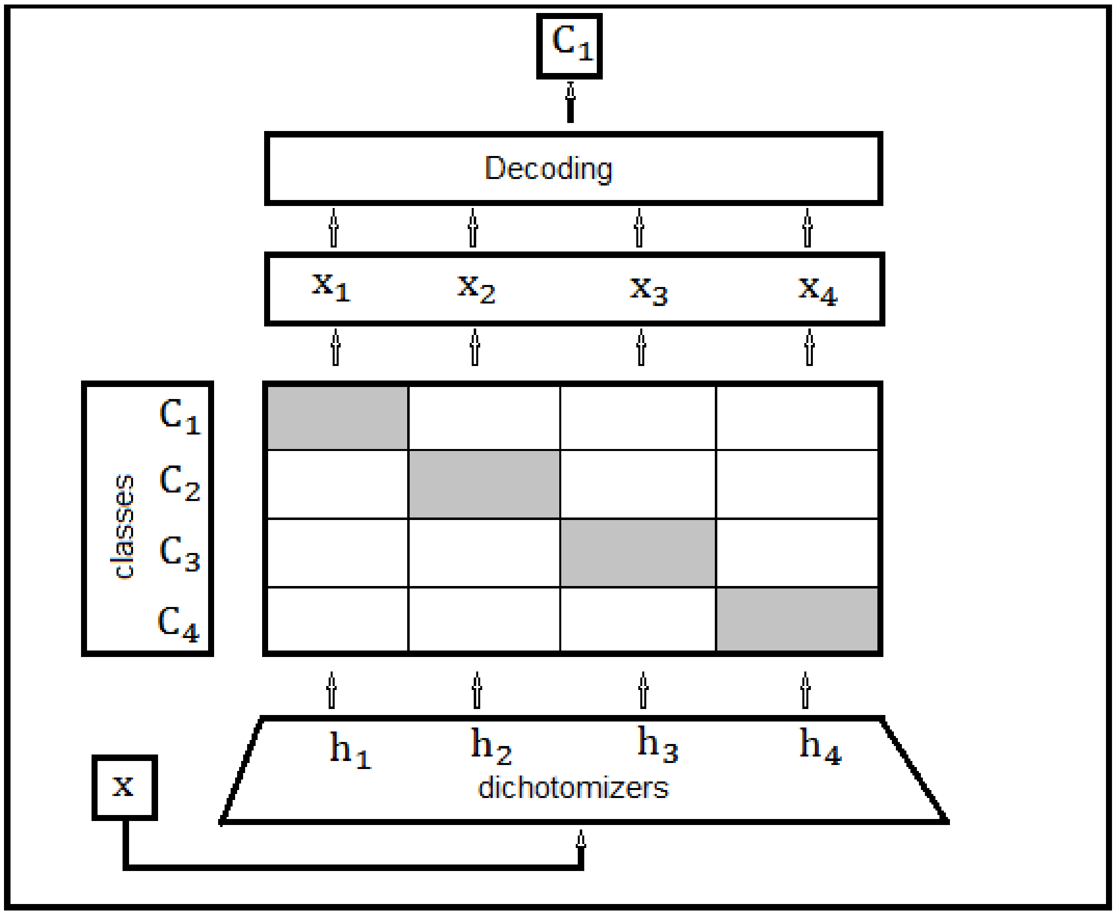


Figure 4.1: An example of output code design (1-vs-all)

$\{1, 0\}^k \times n$. Figure 4.1 shows an example: the so-called one-vs-all method which is one of the simplest choices for the output code. Here, 1s are marked as grey cells and 0s as white ones. Note that this one-vs-all matrix is not actually error-correcting and just used as an example.

We also can add another symbol, namely -1, to the matrix which now contains $\{1, 0, -1\}$. In this case, we treat 1 as positive, -1 as negative and 0 as ignored, e.g. we can represent the pairwise method that we mentioned in the chapter 2. We will introduce two more encoding methods in this chapter, namely exhaustive and random, which we will consider in detail in Sections 4.3 and 4.4.

4.2 Decoding methods

The decoding process is to apply the n binary classifiers and then obtain an output code x from the learners. This output code is used to compare to the base codewords (rows) that are defined in the matrix M. The new instance is assigned to the class with the closest codeword. The most frequently decoding designs are: Hamming Decoding, Inverse Hamming Decoding and Euclidean Decoding.

4.2.1 Hamming decoding

The Hamming decoding method is based on measurement of Hamming distance and it is one of the most common decoding techniques. The experimental results in this thesis are based on Hamming decoding method. In this section, we will give a brief introduction of Hamming distance and then we will state how the Hamming decoding method has been used.

Hamming distance

We have a short review of the Hamming distance here. Hamming distance was first introduced by Richard Hamming in 1950. It is used in telecommunication to detect and correct flipping errors. In machine learning, the term Hamming distance between two equal length words is the number of different bits at the same position where the corresponding symbols are different. In other words, it describes the minimum number of substitutions need to change from one word to the other word.

To calculate the Hamming distance between two words is quite simple. The Hamming distance calculation can be processed as follows (suppose there are k bit symbols in each string):

- Initialise a distance counter d to be 0

- Iterate i from 0 to k.
- Compare the symbols at the i^{th} position in both strings.
- If they are different.

  increase counter d by 1
- Stop when we reach the last bit of the string.

Hamming distance represents the distance between two strings. In other words, it is the number of different bits in two strings. For example, the Hamming Distance between:

"*apple*" and "*apply*" is 1;

"10010011" and "01011111" is 4;

"10101010" and "01010101" is 8.

Hamming distance can be used for any string of symbols. However, in this thesis, the examples are binary cases so that most strings are composed of 0s and 1s.

Hamming decoding method

The Hamming decoding method (Hamming, 1950) is one of the most popular strategies for ECOCs. From its name, it is obvious that the initial proposal to decode is to use the Hamming decoding measure. There is an alternative way to calculate Hamming Distance. It is defined as follows:

$$HD(x, y_i) = \sum_{j=1}^{n}(1 - sign(x^j {y_i}^j))$$

The Hamming decoding method is based on the error correcting principle under the assumption that two possible symbols can be found at each position of the sequence. Each learning task can be modelled as a binary problem.

	f_1	f_2	f_3	f_4	f_5	f_6	f_7
class A	0	0	0	0	0	0	0
class B	0	0	0	1	1	1	1
class C	0	1	1	0	0	1	1
class D	1	0	1	0	1	0	1
output	1	0	1	1	1	0	1

Table 4.1: An example of an exhaustive matrix using Hamming distance

Hamming decoding can guarantee to correct up to $\frac{d-1}{2}$ bit errors, where d is the minimum Hamming distance between all possible pairs in the codeword matrix. Suppose we have the following codeword matrix:

Here $f_1, f_2, ..., f_7$ are the base learners. These learners can be any binary learners as they learn to discriminate between 0s and 1s. For example, f_1 learns class D against class A, class B and class C. In the output example in the table, the prediction of f_1 is positive so that the output for f_1 is 1. This is a very clear case. Learner f_3 is a little bit different. It learns class C and class D against class A and class B. In other words, f_3 predicts whether the class belongs to either class A and class B or class C and class D. In this example, the prediction of f_3 is also positive (1). We keep tracking the classifiers $f_1, f_2, ..., f_7$ and then we get the output code string 1011101.

With the output code string 1011101 in hand, we then calculate the Hamming distance to each base codeword. The class with the smallest Hamming distance is the predicted class. In our example, the Hamming distances to each base codeword are:

class A: 0000000 *vs* 1011101 is : 5

class B: 0001111 *vs* 1011101 is : 3

class C: 0000000 *vs* 0110011 is : 5

class D: 1010101 *vs* 1011101 is : 1

Since the codeword of class D has smallest Hamming distance 1, we assign the test instance to class D. The Hamming distance can be calculated using either of those two ways that we mentioned before.

4.2.2 Inverse Hamming decoding

Inverse Hamming decoding (Escalera & Pujol, 2010) is another popular decoding method. It is defined as follows: Let Δ be the matrix composed by the Hamming decoding measure between the codewords. Δ can be inverted to find the vector containing the N individual class likelihood function by means of:

$IHD(x, y_i) = max(\Delta^{-1}D^T)$ where,

$\Delta(i_1, i_2) = HD(y_{i1}, y_{i2})$, and

D is the vector of Hamming decoding values of the test codeword x for each of the base codewords y_i.

Escalera and Pujol state that, in practical situation, the behaviour of the inverse Hamming decoding method is very close to the behaviour of the Hamming decoding strategy.

4.2.3 Euclidean decoding

Euclidean Decoding (Escalera & Pujol, 2010) is another well-known decoding strategy. This measure is defined as follows:

$ED(x, y_i) = \sqrt{\sum_{j=1}^{n}(x^j - {y_i}^j)^2}$

It measures the Euclidean distance between two code vectors, It also behaves similarly to the Hamming distance.

4.3 Exhaustive encoding method

We have talked about several encoding methods already, e.g. one-vs-rest and one-vs-one. In this section, we consider a powerful method: the exhaustive method. The exhaustive method builds a codeword matrix that contains all possible unique codewords.

Suppose that there are k classes. The number of columns in an exhaustive ECOC is $n = 2^{k-1} - 1$. This means that the length of the codeword is n. Each column represents one base learner. Table 4.1 is an example of exhaustive method. Table 4.1 is an example of the exhaustive method.

It is quite simple to generate the matrix. We know that there should be k rows and n columns. We first build n binary strings of length k. The value of the binary strings starts from 1(in decimal). For example, suppose we have

$k = 4$ classes, then we have

$n = 2^{k-1} - 1 = 7$

The exhaustive matrix should have 4 rows and 7 columns. We write 7 binary strings of length 4 whose decimal value starts from 1 (in decimal). We have:

string 1:	0001
string 2:	0010
string 3:	0011
string 4:	0100
string 5:	0101
string 6:	0110
string 7:	0111

With these 7 strings matrix in hand, we can easily create our exhaustive matrix. All we need to do is to turn the matrix 90 °C clock wise. We then have our exhaustive matrix:

$$\begin{array}{ccccccc}
1&2&3&4&5&6&7\\
\hline
0&0&0&0&0&0&0\\
1&1&1&1&0&0&0\\
1&1&0&0&1&1&0\\
1&0&1&0&1&0&1
\end{array}$$

This process is simple and straightforward. In an exhaustive matrix, each row represents one codeword for one class as in other output-code-based approaches. As we mentioned before for k class datasets, we have k rows and $2^{k-1} - 1$ columns (number of base learners).

Generating an exhaustive ECOC does not require a very intelligent codeword matrix generation design. The process is very easy to follow. In practice, it is a strong method because it uses the maximum number of possible unique base learners. That means it learns all possible binary combination problems.

The exhaustive method is a great approach to use in ECOC-based classification. However, the codeword matrix becomes very large when the number of classes increases. This is because the number of columns is $2^{k-1} - 1$ and will increase exponentially quickly. As a result, the time that is consumed for training the base learners becomes a big issue.

If the number of classes is very large, it is quite impossible to run the experiment. Based on experience with the experiments presented in this thesis, the number of classes can go up to about 8. In this case, we have 127 base learners. Even though exhaustive ECOC work very well, we need to consider another strategy to handle a situation where there are many classes.

4.4 Random method

Because of the limitation of running the exhaustive method, we can use a very simple, so-called "random" method to cover situations that the exhaustive method cannot handle. In this section, we are going to look at one of the random methods that have been used in this thesis. We have used two random methods in this thesis. One is WKEA's existing method and the other is a newly developed one. We can treat the one in WEKA as a purely random method. It simply creates a random code matrix by setting bits based on unbiased coin tosses. In this section, we focus on our newly method that generates an optimized pre-defined matrix.

Instead of using a matrix that is generated purely randomly, we can measure the quality of the codeword matrix before using it. We call this method pre-defined random method. In this method, the number of columns is defined as twice the number of classes ($N = k \times 2, k$ is the number of classes) and the same as in WEKA's completely random method. The strings are composed of 0s and 1s. With the codeword matrix in hand, we measure its qualities. We do this for many randomly generated matrices. Finally, we choose the best codeword matrix for that particular number of classes. This is the "pre-defined" matrix for that number of classes that is then used in the experiments.

A very important aspect of this method is the measurement of quality. We aim to maximize the minimum Hamming distance between all possible pairs of rows and columns. The Hamming distance between rows is a priority because we wish to keep the codewords as separate as possible for different classes. Column separation is important for decorrelating the classifiers.

Table 4.2 shows an example of pre-defined random codeword matrix. In this example, there are 2 classes in original dataset (A and B); we apply clustering to it using 3 clusters. Therefore we have 6 classes in our new training dataset. The

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_10	f_11	f_12
class A'_1	1	0	0	0	1	0	0	0	1	0	0	0
class A'_2	0	0	0	1	1	1	0	0	1	0	0	0
class A'_3	0	0	0	0	1	1	1	0	1	0	0	0
class B'_1	0	0	0	1	1	0	0	1	1	0	0	0
class B'_2	0	1	0	0	1	0	0	0	0	1	1	0
class B'_3	0	0	1	0	0	1	1	0	1	0	0	0

Table 4.2: An example of a pre-defined random matrix

codeword matrix has 12 columns($N = 6 \times 2$). Thus we randomly generate 6 strings of length 12. We do this many times and choose the best one of the matrices that are generated.

The algorithm that we use to find the best matrix is as follows:

- 1. Define the number of rows$k$ and columns N. The number of rows is the number of classes (k) and the number of columns can be calculated using $N = k \times 2$.
- 2. Initialize a variable ($bestScore = 0$) to keep the score for the best matrix.
- 3. Iterate from i to I, where I is a user specified constant. For each iteration:
- 4. Generate a codeword matrix with k strings of length N.
- 5. Measure the quality of the codeword and compute its *score*.
- 6. If ($score > bestScore$)

  i. $bestScore = score$

  ii. save the matrix
- 7. Repeat steps 4, 5 and 6 until we reach the end of the loop.

The quality measurement of a codeword is based on the minimum Hamming distance between all possible pairs of rows and columns. In practice, we mainly focus on row separation. The Hamming distance between columns is calculated only if the minimum Hamming distance between rows equals to the saved score. In that case, it is used as a tie breaker.

The advantage of using the pre-defined random method is that we can handle large numbers of classes. This is because we have a linear number of columns $2 \times k$ rather than exponential growth 2^k in the exhaustive method when the number of classes k increases. Since the pre-defined codeword matrix is smaller than the one in the exhaustive method when k is greater than 4 ($k > 4$), the running time of the pre-defined method is less.

However, smaller size codeword matrices normally have worse accuracy as we will also see in the experiments. The pre-defined method is recommended to be used in situations where it is not possible to run exhaustive methods, i.e. $k > 5$. Nevertheless, when the number of classes gets very large, there an issue arises with the random method, namely that it is very difficult to find a good codeword matrix using random search as it is proposed in this thesis. We use the example in Figure 4.2 again. There are $2^{12 \times 6}$ possible codeword matrices because there are 12×6 bits that can be 0s or 1s. Each combination of 0s and 1s represents one matrix. Therefore, even with modern computers, it is not possible to run all possible matrices when k is getting large. What we do is that we let the computer run for a fix length of time and output the best matrix.

Instead of using more computers to speed up the process of producing more matrices in a period, we can parallelize the program. In particular we can move the process of generating the codeword matrix and measuring the quality of a matrix into a Graphics Processing Unit (GPU).

4.5 GPU-Optimized codeword matrices

The randomly generated codeword matrices are purely independent. This is a perfect situation for parallelization because we can generate matrices parallel to speed up the process. There are many ways that we can parallelize the computing process. In this section, we discuss the GPU programming and the usage of GPU programming for finding code matrices.

The model of GPU programming is to use a CPU and GPU together. The sequential part of the application runs on the CPU and the computationally-intensive part is accelerated by the GPU. It is known that many applications can then run much faster because the high-performance GPU can boost performance. There are many high-level languages that can be chosen to express the parallelism for the GPU programming, such as C, C++ or driver APIs such as OpenCL. The actual program of the GPU-Optimize process used in this thesis is written in JavaCL, which is a wrapper of OpenCL.

4.5.1 OpenCL

OpenCL is an open industry standard platform of programming a heterogeneous collection of CPUs and GPUs. The platform model for OpenCL is defined in Figure 4.2. The model consists of a Host (CPU core) connected to one or more OpenCL devices (GPU cores). Each GPU core is divided into one or more compute units that are further divided into one or more processing elements. The computation is executed within the processing elements. This is the platform model of OpenCL.

There are two execution parts of an OpenCL program in the execution model (Figure 4.3): Kernel and Host program. The kernel executes on he GPU devices and the host program executes on the CPU. The host program defines the context for the kernels and manages the execution process. When a kernel is submitted,

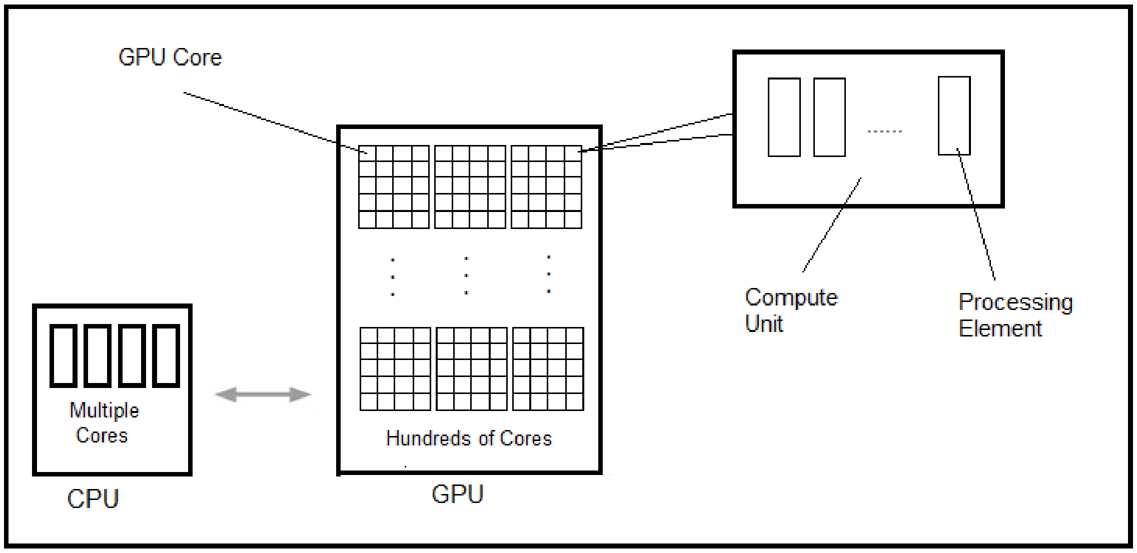


Figure 4.2: OpenCL platform model

an index space is defined. The kernel instance is called work-item. The work-items are organized into work-groups with a work-group ID. Each work-item is assigned a unique local ID within a work-group. These two IDs define a work-item uniquely.

We can specify the size of the work-group and the number of work-groups we wish to use to execute our kernel function program. The selection of these two values affects the execution speed of the program. It is difficult to find the best options and different graphics cards normally have different behaviour. However, we can tune the work-group size and the number of work-groups manually based on observed runtime .

With GPU programming, the CPU and the GPU work together and they need to communicate with each other. There are four different types of memory regions that a kernel function has excess to.

- 1. Global memory: All work-items in all work-groups and the host program have read/write access to this memory region.

- 2. Constant memory: All work-items in all work-groups only have read

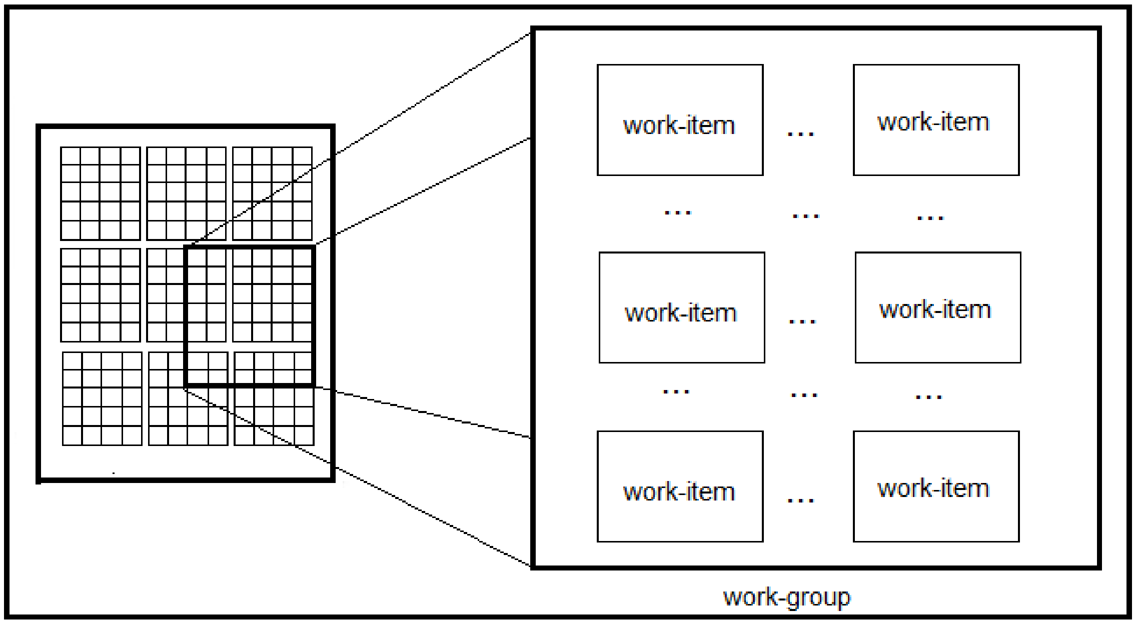


Figure 4.3: OpenCL execution model

access to this memory region and only the host program has read/write access to it.

- 3. Local memory: It is a memory region local to a work-group. Only the work-items in that work-group have read/write access to. There is no access for the host program.

- 4. Private memory: It is a memory region private to a work-item. Only that work-item has read/write access to it but not the host program.

The time cost of visiting a memory region from a kernel is very different. Visiting global memory is slowest, then constant and local memory, private is the fastest. In other words, we want to use private memory if possible. For our problem, where we generate a matrix and calculate its scores, we can ask one kernel to execute thousands of matrices at a time and only write the best one to global memory to let the CPU read it. Our goal of using the GPU is to find a good codeword matrix

```
Kernel:
        typedef struct  ulong a, b, c  randomState;

        unsigned long random(randomState ∗r){
                unsigned long old = r− > b;
                r− >b = r− >a ∗1103515245 + 12345;
                r− >a = (˜old ˆ(r− >b >> 3)) - r− >c++;
                return r− >b;
        }

        void seedRandom(randomState *r, ulong seed){
                r− >a = seed;
                r− >b = 0;
                r− >c = 362436;
        }
```

Figure 4.4: A snippet of kernel code for random number generation

in as little time as possible. That means we generate as many matrices as possible in a given period of time.

4.5.2 JavaCL

JavaCL wraps OpenCL in a nice Java API and it makes OpenCL available to the Java platform. This is important because the code developed for this thesis is integrated into WEKE software. With JavaCL, we embed code for the GPU in a Java program and we can take advantages of its powerful API.

One of the difficulties of using JavaCL is that there is no existing random number generation method defined in the kernel function. We have to implement our own pseudo random number generation method. The method are used in our kernel function is shown in Figure 4.4.

We can write the host program in Java directly using the JavaCL API. For the kernel function, we write the program in a "*C*-like" code as a Java String, and then use the JavaCL API to compile the string into a kernel function. Figure 4.5 shows

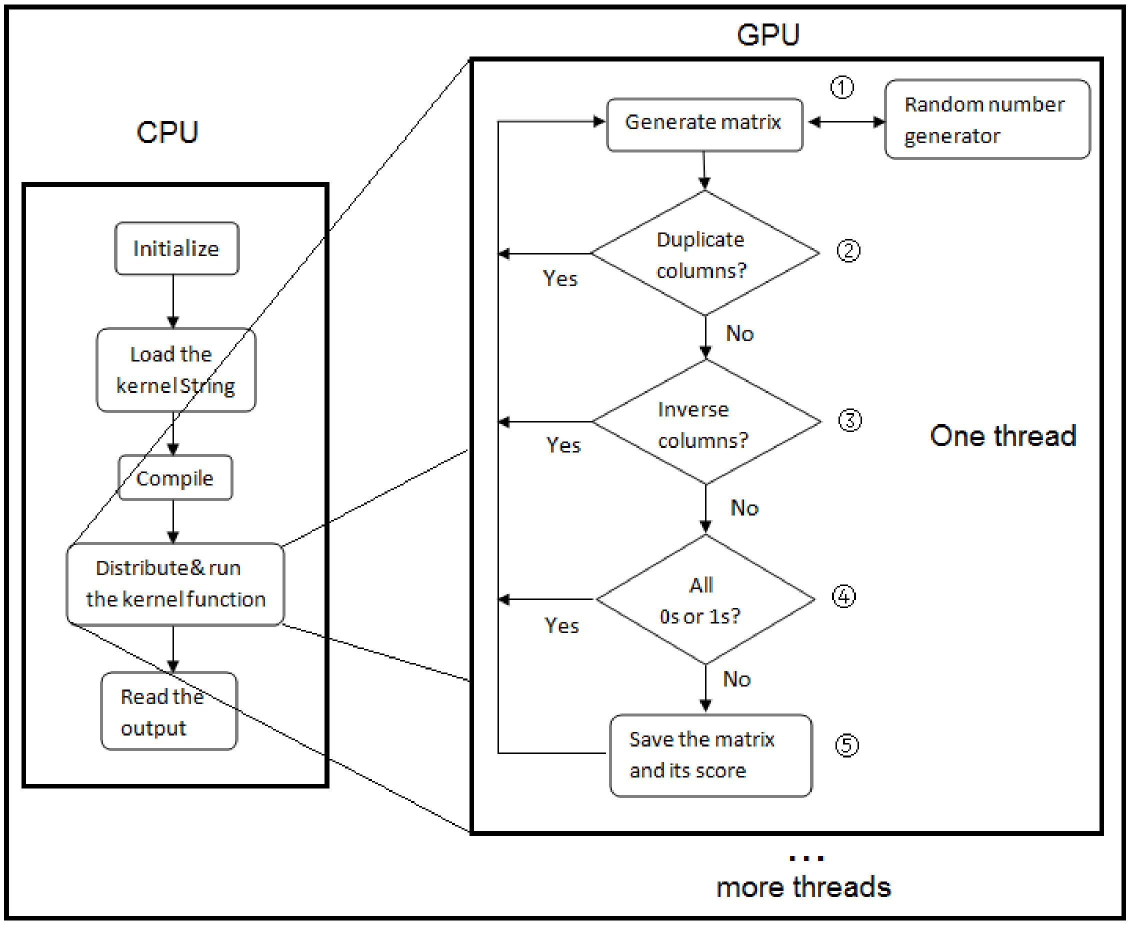


Figure 4.5: Overview of the program flow chart

the overview of the process.

CPU part

The program is written in Java using JavaCL. What follows are the main steps of the part of the program that runs on CPU which are also shown in Figure 4.5:

- 1. Initialize: Set the environment for the kernel function, e.g. initialize the work-group size and the number of work-items in one work-group.
- 2. Load the kernel string: the kernel function is written in a "*C*-like" string. The kernel function used for finding good matrices is shown in Figure B.1

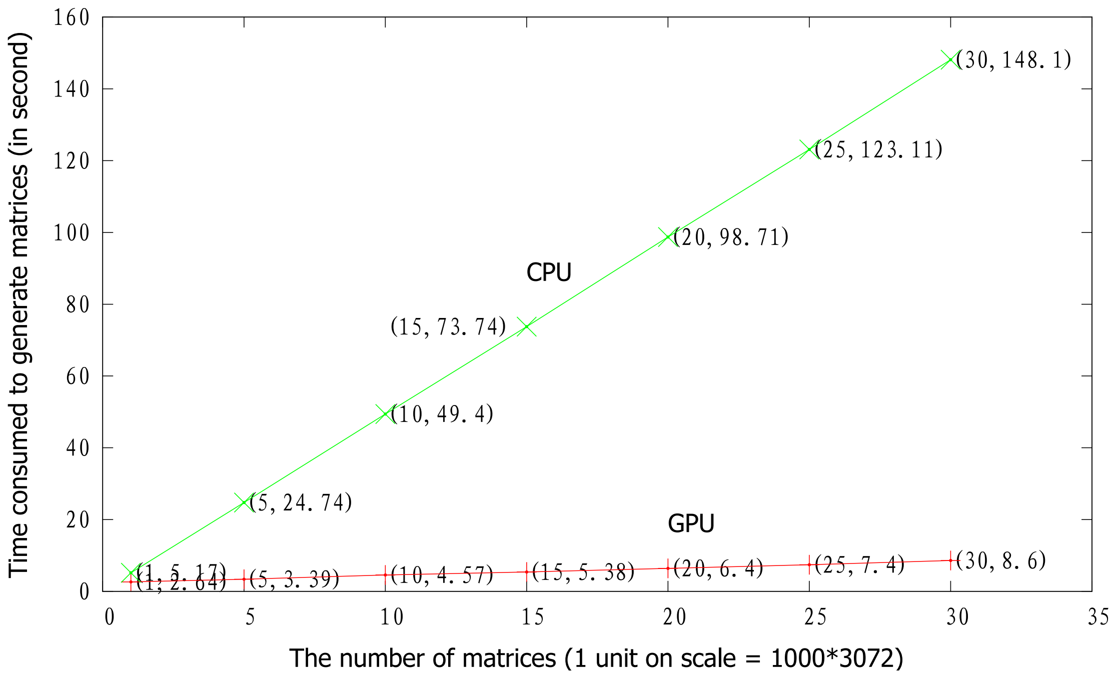


Figure 4.6: GPU vs CPU

(Appendix B).

- 3. Compile: The GPU cannot process a Java string directly so that we need to compile the Java string into a "*C*-like" kernel function using JavaCL.

- 4. Distribute & run the kernel: once we have our kernel function ready, we push the kernel function to the GPU using the JavaCL API.

- 5. Read the output: After all kernels have successfully finished, we read their output form the global memory.

This process can be done as many times as we wish. Figure 4.6 shows the snippet of JavaCL code that we used in a situation where we used 48 work-groups and 32 work-items in one work-group. The complete JavaCL code can be found in Figure B.2 (Appendix B). Note that the kernel function is compiled using JavaCL by the CPU and the actual function then runs on the GPU.

GPU part

We now take a closer look at GPU part in Figure 4.5. There are several steps that are required for finding a good matrix:

- 1. Generate a random matrix. In this step, the random number generation function defined in Figure 4.4 is used.

- 2. Duplicate columns check: we need to validate if there are any columns that are the same. Having the same columns in one matrix means two or more dichotomizers are the same (and make exactly the same errors).

- 3. Inverse columns check: This is necessary to check if there are columns that are the inverse of each other. This is because we do not want to have two dichotomizers that learn the opposite classification of each other (again, the errors would be perfectly correlated).

- 4. Check if all 0s or 1s: It is obvious that we do not want to have these columns in our matrix because they cannot be used learn anything.

- 5. Save the matrix: after we validate steps 2, 3, and 4, we save the matrix we found in this iteration if it achieves a higher score than the best matrix found in previous iterations.

We can continue iterating steps 1,2,3,4 and 5 N times. N is a specified constant number. Note this is just one thread running on one work-item in the GPU. The example used in Figure 4.7 has $48 \times 32 = 2048$ threads, which means 2048 threads are running at the same time. This is why we have a very high performance with GPU programming.

Comparison of performance

We have discussed GPU programming and we know that the GPU can handle thousands of threads smoothly. In this section, we give a short comparison of speed using GPU and CPU.

Figure 4.6 shows a curve based on both GPU-generated and CPU-generated results. The x axis has number of matrices we generated and the y axis has the time consumed to generate those matrices. Because of the way the experiment was set up, we use 3073×1000 as one unit in the x axis. From the figure, we can see that GPU is more than 10 times faster than the CPU. Note also that the GPU is less efficient if there are fewer jobs that need to be done. That is why there is not much difference when generating 3073×1000 matrices (The left-most points on the plot). Note that for the GPU-based results a straightforward translation of the C-like kernel code to Java was performed, so that the results are as directly comparable as possible.

4.5.3 A short summary of the GPU-based experiments

Programming the GPU is quite complex, in particular it is quite difficult to debug. To simplify the problem by using some existing frameworks, JavaCL is used in our experiment. It is worth while using the GPU in a situation where we have a very computationally-intensive problem. The matrix generation problems considered here are independent of each other. By parallelism, we can improve the computing speed more than 10 times. This means that we can get our job done in one days time with the GPU while it takes 10 days to finish with the CPU. The matrices that were found using GPU-based random search are listed in Table C.1 (Appendix C).

```
JavaCL:
        // create context
        CLContext context = JavaCL.createBestContext();
        // define the work the number of tasks.
        int task = 48*32;
        CLQueue queue = context.createDefaultQueue();
        // initialise the length of the output buffer
        int size = 10;

        // define buffers.
        CLLongBuffer score = context.
                createLongBuffer(CLMem.Usage.InputOutput, size);
        CLLongBuffer memIn = context.
                createIntBuffer(CLMem.Usage.InputOutput, workgroup);
        CLLongBuffer memOut = context.
                createLongBuffer(CLMem.Usage.InputOutput, workgroup);

        // get the kernel that we wish to push to the GPU
        String kernelFunction = getKernelFunction();
        try {
            // create kernel function using given string
            CLProgram program = context.createProgram(kernelFunction)
                .build();
            // push the kernel function to the GPU and run it
            IntBuffer a = memIn.map(queue, MapFlags.Write);
            CLKernel kernel = program.
                createKernel("calculate", score, memIn, memOut);
        } catch (CLBuildException e) {
            e.printStackTrace();
        }
```

Figure 4.7: A short snippet of JavaCL code

Chapter 5

Empirical Results

In the previous chapters, we have discussed different encoding and decoding methods for Error-Correcting Output Codes, and how to use a clustering approach to turn two-class problems into multi-class problems. This chapter presents empirical results on prediction performance for those strategies. First, we introduce the dataset that were used in our experiment as well as how these dataset were produced. Secondly, we will give a comparison of the classification models that were built using different base learners, such as AdaBoost, Bagging and RandomForest. We also compare the performance between exhaustive codes and the END method for multi-class classification. Finally, the performance of the "pre-defined" strategy and WEKA's random code method are discussed. All experiments were run on distributed remote machines using the WEKA Experimenter. Note that C4.5 decision tree is implemented in WEKA as J48 and the WEKA version used is version 3.6.

5.1 Datasets

Let us briefly review the datasets that were used in our experiment. The original dataset collection has 33 datasets and some of these datasets have more than 2 classes originally. For testing, where we want apply ECOCs on two-class datasets, we need to transfer the multi-class data into two-class data first. Because the Euclidean distance measure is used in our algorithms, when clustering is done, all no-class nominal attributes were turned into numeric ones. Therefore, there are two steps to pre-process our datasets: generating two-class datasets and turning nominal attributes to numeric ones.

5.1.1 Generating two-class datasets

One of the filters in WEKA called $MergeTwoValues$ was used to merge classes. It is located in the package $weka.filters.unsupervised.attribute$. This filter was used manually, because the aim was to keep the class distribution of the datasets as balanced as possible. For example, if the class distribution in dataset T is:

class A : 50 instances
class B : 35 instances
class C : 30 instances

We merge class B and class C and we end up with the following two-class dataset T':

class A: 50 instances
class B & C: 65 instances.

We only apply this filter on multi-class datasets. Two-class datasets were unchanged. After pre-processing the datasets manually, all datasets were turned to two-class ones.

5.1.2 Nominal to binary conversion

Another filter in WEKA, called $NominalToBinary$ that is also located in the package $weka.filters.unsupervised.attribute$, was used to transfer nominal attributes to numeric attributes. This is a very straightforward method to use. We can directly apply this filter on a nominal dataset to produce a numeric dataset by creating a binary numeric presence/absence indicator for each value of a nominal attribute.

After using the two filters $MergeTwoValues$ and $NominalToBinary$, we end up with 33 numeric two-class datasets. These 33 datasets are used for presenting the performance of difference algorithms in what follows.

5.2 Comparison of EOCOs with different base learners

In this section, we compare the performance of different base learners using ECOCs based on the exhaustive method with that same base learner algorithm applied directly to the datasets. We also want to examine the accuracy of classification models that use different number of clusters.

5.2.1 AdaBoost

In Table 5.1, the column " Original AdaBoost" represents the accuracy of the AdaBoost algorithm applied directly with C4.5 decision trees (J48 in WEKA). The other columns show the results for different numbers of clusters in the clustering based scheme used in conjunction with the ECOC algorithm with the exhaustive method. AdaBoost with J48 was here used as the base learner for the ECOC-based scheme. The result shows that the accuracy of the ECOC models using the exhaustive method is better than using AdaBoost directly. Compared with AdaBoost, 2 clusters per class yield 5 significant improvements and 0 significant degradations; 3 clusters yield 9 significant wins and 1 significant loss; and 4 clusters yield 10 significant improvements and 1 significant loss.

Dataset	Original AdaBoost	2 clusters	3 clusters	4 clusters
mfeat	93.81	95.17 ○	97.30 ○	97.99 ○
mushroom	100.00	100.00	100.00	100.00
mfeat	92.66	93.25	94.69 ○	95.03 ○
optdigits	97.72	98.07 ○	98.67 ○	98.88 ○
nursery	100.00	100.00	100.00	100.00
kr-vs-kp	99.58	99.56	99.59	99.63
ionosphere	93.05	93.13	94.16	93.99
lymphography	84.75	85.56	85.46	86.55
labor	85.10	86.33	88.57	89.63
vehicle	96.99	97.29	97.75	98.00
waveform	86.17	86.69 ○	88.57 ○	89.36 ○
vote	95.54	96.02	96.25	95.95
sonar	79.13	81.42	85.27 ○	86.70 ○
sick	98.95	98.96	98.95	98.94
splice	97.25	97.29	97.70 ○	98.00 ○
spambase	95.27	95.54	96.02 ○	96.32 ○
horse-colic.ORIG	69.91	70.16	70.76	71.01
credit-rating	85.03	85.22	86.39	86.61
horse-colic	82.44	83.36	84.34	84.23
cylinder-bands	78.85	81.80 ○	85.07 ○	86.07 ○
german-credit	71.97	73.93 ○	75.18 ○	75.55 ○
breast-cancer	68.08	70.24	71.33	71.73
balance-scale	87.25	86.74	86.35	86.14
bridges-version1	81.66	82.74	82.85	84.85
wisconsin-breast-cancer	96.08	96.00	96.43	96.74
heart-statlog	78.59	79.33	80.33	80.67
hayes-roth	76.50	75.06	75.59	75.12
hypothyroid	99.62	99.64	99.64	99.68
hepatitis	82.55	82.15	84.02	84.79
ecoli	95.29	95.53	96.16	96.30
pima-diabetes	71.69	72.73	74.07	74.66 ○
haberman	66.93	68.78	67.62	67.75
flags	73.93	72.85	73.57	73.01

○, ● statistically significant improvement or degradation

Table 5.1: Comparison of (a) AdaBoost and (b) multi-clustering with ECOCs using AdaBoost as the base learner, with 2, 3 and 4 clusters (i.e. 4, 6, and 8 classes in the transformed datasets)

The result also indicates that the exhaustive method has better performance with larger numbers of clusters: assume that x is the number of clusters and y is the number of significant wins, then we can obtain the curve shown in Figure 5.1.

5.2.2 Bagging

Let us now consider the effect on another ensemble learning algorithms, namely Bagging. In Table 5.2, the column "Original Bagging" represents the accuracy of the Bagging algorithm applied directly with C4.5 decision trees as the base learner. The other columns show the results for the different numbers of clusters used in the ECOC method with exhaustive coding, using Bagging as its base learner. Note that

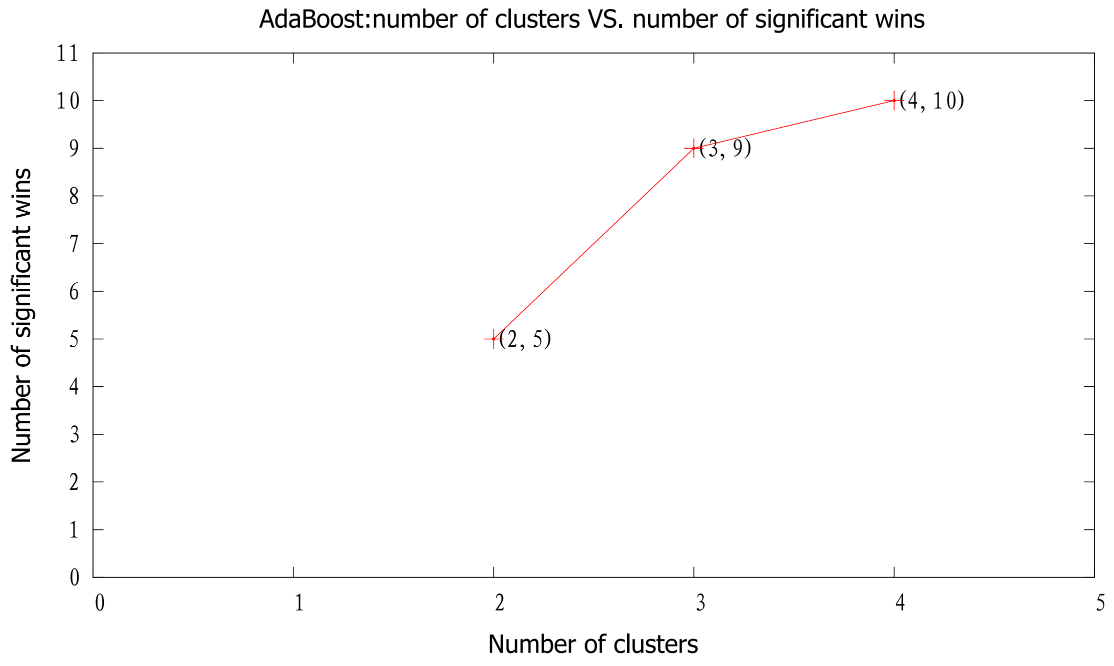


Figure 5.1: AdaBoost: number of clusters vs. number of significant wins

C4.5 is the base learner of Bagging in the multi-clustering approach.

The empirical results in Table 5.2 show that the multi-clustering algorithm with Bagging as its base learner is also better than when we apply Bagging directly to the datasets. With 2 clusters, there are 2 statistically significant improvements, there are 5 statistically significant improvements with 3 clusters and there are 7 statistically significant improvements with 4 clusters. There is no any statistically significant degradation for the multi-clustering algorithm with Bagging as its base learner. The result also shows that the accuracy of the models increases as the number of clusters gets larger. Figure 5.2 draws the learning curve, where the x axis has the number of clusters and the y axis has the number of significant wins.

5.2.3 Random forest

Random forest are another powerful ensemble learning approach. Table 5.3 shows the result of WEKA "Original RandomForest" and multi-clustering with Random-

Dataset	Original Bagging	2 clusters	3 clusters	4 clusters
mfeat	87.77	91.42 ○	95.25 ○	96.31 ○
mushroom	100.00	100.00	100.00	100.00
mfeat	92.01	92.25	93.72 ○	94.32 ○
optdigits	96.62	97.17 ○	98.15 ○	98.53 ○
nursery	100.00	100.00	100.00	100.00
kr-vs-kp	99.35	99.31	99.27	99.32
ionosphere	92.17	92.37	93.08	93.31
lymphography	82.93	83.60	84.63	84.71
labor	82.20	87.47	87.17	86.80
vehicle	95.59	95.87	96.50	96.96 ○
waveform	86.65	86.81	88.32 ○	88.80 ○
vote	96.30	96.62	96.66	96.57
sonar	78.51	79.79	82.45	83.21
sick	98.84	98.87	98.77	98.78
splice	97.57	97.51	97.71	97.81
spambase	94.32	94.48	94.82 ○	95.04 ○
horse-colic.ORIG	69.54	69.32	69.97	68.86
credit-rating	86.06	85.68	86.16	86.26
horse-colic	85.64	85.75	85.37	85.45
cylinder-bands	77.57	79.35	81.15	81.44 ○
german-credit	74.29	74.86	75.42	75.63
breast-cancer	71.16	72.20	72.60	72.88
balance-scale	87.44	87.33	88.13	88.72
bridges-version1	81.87	81.60	84.40	84.75
wisconsin	96.07	95.90	96.31	96.77
heart-statlog	80.59	80.07	79.59	80.41
hayes-roth	79.99	79.70	79.98	80.39
hypothyroid	99.63	99.61	99.54	99.51
hepatitis	80.60	81.46	82.23	82.22
ecoli	95.57	96.01	96.43	96.54
pima-diabetes	75.65	75.43	76.00	75.47
haberman	73.08	73.93	73.44	73.75
flags	73.09	73.88	74.25	73.31

○, ● statistically significant improvement or degradation

Table 5.2: Comparison of (a) Bagging and (b) multi-clustering with ECOCs using Bagging as the base learner, with 2, 3 and 4 clusters (i.e. 4, 6, and 8 classes in the transformed dataset)

Forest as its base learner, again using exhaustive ECOCs. In this experiment, we set 100 as the number of trees to be built in the random forest (the WEKA default is 10, which is generally too small). From the result, we can see that RandomForest applied directly to the two-class data is better than multi-clustering. With multi-clustering, we have 7 significant degradations with 2 clusters and 6 significant degradations with 3 clusters.

From the results shown in Table 5.3, unlike Adaboost or Bagging, RandomForest is not a suitable baser learner for the multi-clustering algorithm. It is instructive to set the number of trees in RandomForest to be 10 when this experiment is performed the multi-clustering algorithm does not reduce the accuracy of the classification

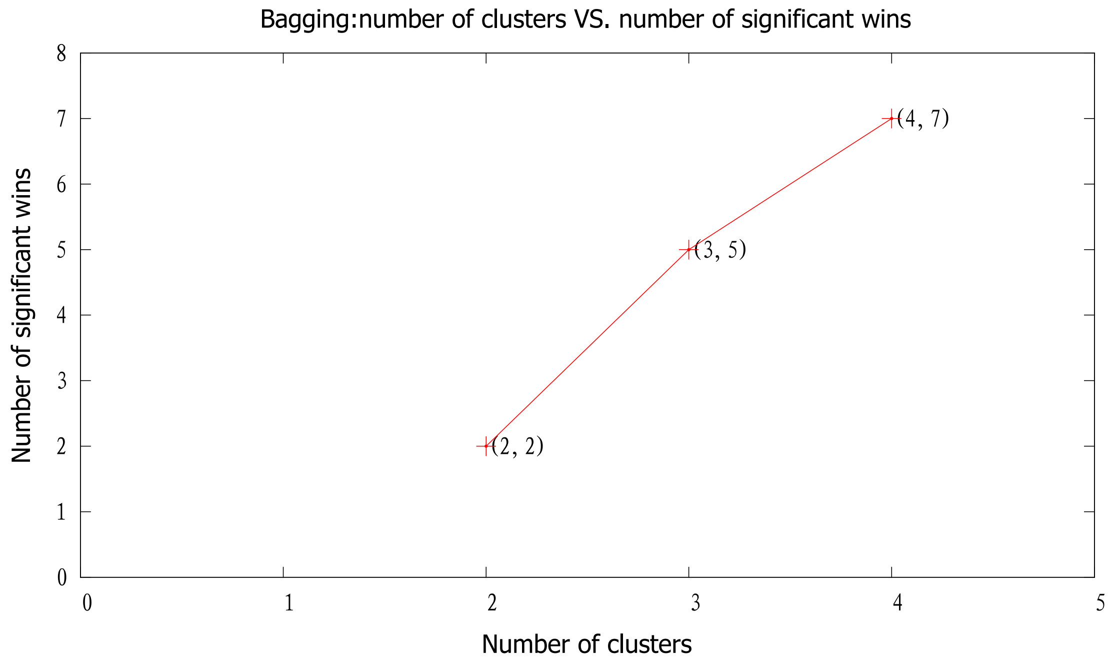


Figure 5.2: Bagging: number of clusters vs. number of significant wins

models. Hence, when sufficiently many trees are used in a random forest, clustering plus error-correction does not improve performance and is sometimes detrimental.

5.3 Exhaustive codes vs ENDs

We have discussed the performance of the exhaustive method with AdaBoost, Bagging and RandomForest as the base learners. In this section, we examine the performance of the exhaustive method versus the END algorithm discussed in Section 2.1.4, which is also applied using the multi-clustering approach. We will consider how the accuracy is affected along with changes resulting from the number of clusters. Note that all experiment results in this section are based on using AdaBoost as the base learner, with J48 is the base learner of AdaBoost.

Dataset	Original RandomForest	2 cluster	3 clusters
mfeat	96.90	95.62 ●	96.28
mushroom	100.00	100.00	100.00
mfeat	94.45	93.35 ●	94.39
optdigits	98.57	97.97 ●	98.18 ●
nursery	100.00	100.00	100.00
kr-vs-kp	99.27	99.07	98.90 ●
ionosphere	93.48	93.45	93.68
lymphography	87.05	84.44	85.38
labor	87.87	85.90	89.10
vehicle	97.08	96.86	97.02
trains	61.00	49.00	42.00
waveform	88.78	88.87	88.66
vote	96.52	95.79	95.79
sonar	84.63	84.15	84.57
sick	98.41	98.07 ●	98.00 ●
splice	96.72	96.01	94.99 ●
spambase	95.68	94.44 ●	94.97 ●
horse-colic.ORIG	70.49	70.17	70.84
credit-rating	87.35	87.12	87.33
horse-colic	86.00	85.24	84.23
cylinder-bands	82.80	83.39	83.07
german-credit	76.38	75.19	75.46
breast-cancer	71.94	73.70	73.12
balance-scale	87.55	87.33	88.29
bridges-version1	86.53	82.49	84.30
wisconsin-breast-cancer	96.58	95.32 ●	96.71
heart-statlog	82.26	80.81	79.89
hayes-roth	76.50	74.20	74.68
hypothyroid	99.63	99.12 ●	99.06 ●
hepatitis	84.08	80.95	81.94
ecoli	95.95	95.66	96.30
pima-diabetes	76.01	75.47	75.69
haberman	69.29	69.10	68.94
flags	72.24	70.09	72.34

∘, ● statistically significant improvement or degradation

Table 5.3: Comparison of (a) RandomForest and (b) multi-clustering with ECOCs using RandomForest as the base learner, with 2, 3 clusters (i.e. 4, and 6 classes in the transformed dataset)

2 clusters

Table 5.4 shows a comparison of the exhaustive correction method and the END algorithm in a situation with 2 clusters per class. Based on this result, the END algorithm is better than the exhaustive method because the END method has 4 statistically significant wins and no significant loss. It has higher accuracy on 24 out of 33 dataset.

Dataset	Exhaustive	END	
balance-scale	86.74	86.70	
breast-cancer	70.24	70.53	
wisconsin-breast-cancer	96.00	96.23	
bridges-version1	82.74	83.05	
horse-colic	83.36	82.87	
horse-colic.ORIG	70.16	70.25	
credit-rating	85.22	85.75	
german-credit	73.93	74.68	
cylinder-bands	81.80	85.07	
pima-diabetes	72.73	73.60	
ecoli	95.53	95.89	
flags	72.85	73.37	
haberman	68.78	67.02	
hayes-rotht	75.06	75.60	
heart-statlog	79.33	79.63	
hepatitis	82.15	81.30	
hypothyroid	99.64	99.59	
ionosphere	93.13	93.33	
kr-vs-kp	99.56	99.61	
labor	86.33	88.83	
lymphography	85.56	85.38	
mfeat	93.25	94.09	
mfeat	95.17	97.46	○
mushroom	100.00	99.99	
nursery	100.00	100.00	
optdigits	98.07	98.57	○
sick	98.96	98.86	
sonar	81.42	83.55	
spambase	95.54	95.94	
splice	97.29	97.89	○
vehicle	97.29	97.33	
vote	96.02	95.79	
waveform	86.69	88.48	○

○, ● statistically significant improvement or degradation

Table 5.4: Exhaustive codes vs. END (2 clusters per class in the multi-clustering approach)

3 clusters

When we increase the number of clusters per class from 2 to 3, the accuracy of the classification models using the exhaustive correction method improves dramatically. Compared with END, there is no longer any significant degradation, instead 20 out of 33 classification models have higher accuracy than those models based on the END method. The empirical results are shown in Table 5.5.

4 clusters

If we keep increasing the number of clusters per class from 3 to 4, the performance of the exhaustive method is improved again. From the results shown in Table 5.6,

Dataset	Exhaustive	END
mfeat	97.30	97.75
mushroom	100.00	100.00
mfeat	94.69	94.72
optdigits	98.67	98.65
nursery	100.00	100.00
kr-vs-kp	99.59	99.56
ionosphere	94.16	93.33
lymphography	85.46	86.00
labor	88.57	87.67
vehicle	97.75	97.45
waveform	88.57	88.85
vote	96.25	96.00
sonar	85.27	82.92
sick	98.95	98.78
splice	97.70	97.93
spambase	96.02	96.03
horse-colic.ORIG	70.76	70.76
credit-rating	86.39	86.10
horse-colic	84.34	83.58
cylinder-bands	85.07	85.28
german-credit	75.18	74.74
breast-cancer	71.33	70.71
balance-scale	86.35	86.94
bridges-version1	82.85	83.20
wisconsin-breast-cancer	96.43	96.53
heart-statlog	80.33	79.74
hayes-roth	75.59	75.18
hypothyroid	99.64	99.57
hepatitis	84.02	83.50
ecoli	96.16	95.77
pima-diabetes	74.07	73.98
haberman	67.62	66.84
flags	73.57	73.42

○, ● statistically significant improvement or degradation

Table 5.5: Exhaustive codes vs. END (3 clusters per class in the multi-clustering approach)

we can see that there is 1 significant improvement with 4 clusters. Taking a closer look at the result, there are only two models based on the END approach that have higher accuracy now.

Based on what we obtained in the experiments, we can summarise a message namely that the exhaustive correction method benefits from having more clusters per class. Note that similar results can be obtained using Bagging as the base learner (see Table A.1 in Appendix A). However, we have a computational limitation when running the exhaustive method. This is because the number of columns in codeword matrix grows exponentially. In a situation with 4 clusters per class, we have $4 \times 2 = 8$ new classes (recall that our original dataset has 2 classes). That

Dataset	Exhaustive	END	
mfeat	97.99	97.79	
mushroom	100.00	100.00	
mfeat	95.03	94.83	
optdigits	98.88	98.77	
nursery	100.00	100.00	
kr-vs-kp	99.63	99.50	
ionosphere	93.99	93.45	
lymphography	86.55	85.13	
labor	89.63	87.73	
vehicle	98.00	97.59	
waveform	89.36	88.79	
vote	95.95	95.79	
sonar	86.70	84.46	
sick	98.94	98.67	•
splice	98.00	97.93	
spambase	96.32	95.99	
horse-colic.ORIG	71.01	69.73	
credit-rating	86.61	86.14	
horse-colic	84.23	83.20	
cylinder	86.07	84.91	
german-credit	75.55	74.21	
breast-cancer	71.73	71.50	
balance-scale	86.14	86.62	
bridges-version1	84.85	82.35	
wisconsin-breast-cancer	96.74	96.54	
heart-statlog	80.67	79.52	
hayes-roth	75.12	75.51	
hypothyroid	99.68	99.53	
hepatitis	84.79	82.12	
ecoli	96.30	96.06	
pima-diabetes	74.66	73.71	
haberman	67.75	66.34	
flags	73.01	72.65	

∘, • statistically significant improvement or degradation

Table 5.6: Exhaustive codes vs. END (4 clusters per class in the multi-clustering approach)

means we have $2^8 - 1 = 127$ columns and 8 rows in our exhaustive error correcting output code matrix. When increasing the number of clusters to 5, we end up with $2^{(5\times2)} - 1 = 255$ columns and $5 \times 2 = 10$ rows in the codeword matrix, which is just too big to run. Therefore, all our exhaustive correction results are based at most 4 clusters per class.

5.4 Exhaustive codes vs Random codes

As we mentioned, there is a limitation in using the exhaustive correction method. As an alternative, we can use a random coding matrix instead. In this section, we give a comparison of the exhaustive method and the random method. We also summarise a performance trend for the random method illustrating and behaviour when the number of clusters changes. Note that all experiment results in this section are based on using AdaBoost with C4.5 as the base learner.

Instead of comparing exhaustive with random pair by pair like we did when comparing exhaustive vs END, we put all the results with the comparison to the random method in one table. Table 5.7 shows a comparison of the random method using different numbers of clusters and exhaustive method with 2 clusters (i.e. 4 classes in the transformed problem). It is obvious that all classification models of the random method are worse than those of the exhaustive method. The results also show that the accuracy decreases as the number of clusters increases with the random method.

From the result shown in Table 5.7, we can summarise that the random method is worse than the exhaustive method. Moreover, it does not improve the performance when the number of clusters increases instead it may reduce accuracy. Therefore, one should only consider using the random method in a situation where the exhaustive method is not applicable. The advantage of using the random method is that the time that is consumed for training the base learner is less. This is because there are only $2 \times k$ columns and k rows in the random codeword matrix used in WEKA (see Section 4.4) while there are $2^k - 1$ columns and k rows in the exhaustive one. Recall that k is the number of classes times the number of clusters. However, this small number of columns is most likely the reason why the random method normally produces less accurate classification models.

5.5 Evaluating "pre-defined" codes

From we have observed, we can say that the random method does not perform well. However, running the random method is much faster than the exhaustive method

Dataset	Exh(2)	Ran(5)	Ran(10)	Ran(3)	Ran(2)	Ran(4)
mfeat	95.17	90.72 ●	87.58 ●	89.84 ●	91.16	89.07 ●
mushroom	100.00	96.03	92.51 ●	94.75	95.29	92.65 ●
mfeat	93.25	86.70 ●	86.46 ●	87.31	87.32	86.00 ●
optdigits	98.07	92.38 ●	90.60 ●	91.66	91.76	91.37 ●
nursery	100.00	92.06 ●	91.39 ●	94.27	96.80	93.50 ●
kr-vs-kp	99.56	93.05 ●	91.75 ●	94.46	94.85	93.97 ●
ionosphere	93.13	85.38 ●	86.47 ●	84.95	87.02	85.57
lymphography	85.56	78.36	78.44	80.46	81.81	77.50
labor	86.33	83.03	82.63	84.10	80.60	79.97
vehicle	97.29	91.09 ●	89.54 ●	91.45	94.03	91.23 ●
trains	66.00	62.00	66.00	59.00	64.00	63.00
waveform	86.69	80.59 ●	80.74 ●	81.42	82.33	80.87 ●
vote	96.02	88.89 ●	88.36 ●	89.43	92.85	89.77 ●
sonar	81.42	75.70	76.73	76.69	77.12	75.49
sick	98.96	91.80	90.81 ●	95.43	97.74	92.18
splice	97.29	90.28 ●	89.86 ●	90.06 ●	91.55	86.42 ●
spambase	95.54	88.75	88.63 ●	88.50	90.31	87.94
horse-colic.ORIG	70.16	67.75	67.74	68.37	68.55	69.05
credit-rating	85.22	80.16	81.26 ●	79.74	82.70	79.86
horse-colic	83.36	77.06 ●	77.03 ●	79.78	79.82	77.20 ●
cylinder-bands	81.80	77.98	76.91	77.94	77.63	78.72
german-credit	73.93	70.64	69.84 ●	70.25	70.92	70.55
breast-cancer	70.24	67.56	68.05	67.02	67.34	67.68
balance-scale	86.74	82.51	81.38 ●	80.70 ●	83.28	80.91 ●
bridges-version1	82.74	77.41	80.30	76.90	78.91	77.85
wisconsin-breast-cancer	96.00	90.29	89.86 ●	90.66	93.93	92.22
heart-statlog	79.33	75.56	75.15	75.04	76.74	74.19
hayes-roth	75.06	74.15	73.98	72.52	71.66	69.21
hypothyroid	99.64	95.69	92.50 ●	93.29	96.74	94.13
hepatitis	82.15	75.97	78.53	76.94	77.40	76.60
ecoli	95.53	89.12 ●	89.23 ●	90.40	91.19	89.78 ●
pima-diabetes	72.73	69.44	69.15	69.44	70.75	69.60
haberman	68.78	65.91	63.13	65.69	65.59	64.49
flags	72.85	68.69	68.51	68.78	72.02	68.84

○, ● statistically significant improvement or degradation

Table 5.7: Exhaustive vs. Random codes in multi-clustering (AdaBoost + C4.5 as the base learner)

based on our experiments. The “pre-defined” approach (discussed in Section 4.4) is a method that has the same size of codeword matrix as the random method and in theory it could have similar performance as the exhaustive method. In this section we look at the performance of the “pre-defined” matrices that we have found using GPU-based calculation in comparison with the random method.

Table 5.8 shows the performance of the exhaustive method, the random method and the “pre-defined” method with 4 clusters per class. Compared with the exhaustive correction method, the random method has 18 statistically significant degradations and the “pre-defined” method has only 8 statistically significant degradations. Even though the “pre-defined” method is worse than the exhaustive method, the

Dataset	Exhaustive	Random	"pre-defined"
mfeat	97.99	89.07 ●	95.53 ●
mushroom	100.00	92.65 ●	100.00
mfeat	95.03	86.00 ●	93.39 ●
optdigits	98.88	91.37 ●	98.12 ●
nursery	100.00	93.50 ●	100.00
kr-vs-kp	99.63	93.97 ●	99.47
ionosphere	93.99	85.57	93.17
lymphography	86.55	77.50 ●	84.40
labor	89.63	79.97	88.50
vehicle	98.00	91.23 ●	96.96 ●
trains	65.00	63.00	61.00
waveform	89.36	80.87 ●	86.07 ●
vote	95.95	89.77 ●	94.98
sonar	86.70	75.49 ●	81.70
sick	98.94	92.18	98.87
splice	98.00	86.42 ●	95.34 ●
spambase	96.32	87.94	95.43 ●
horse-colic.ORIG	71.01	69.05	69.78
credit-rating	86.61	79.86 ●	85.51
horse-colic	84.23	77.20 ●	83.58
cylinder-bands	86.07	78.72 ●	83.19 ●
german-credit	75.55	70.55 ●	73.68
breast-cancer	71.73	67.68	71.20
balance-scale	86.14	80.91	86.40
bridges-version1	84.85	77.85	83.49
wisconsin-breast-cancer	96.74	92.22	95.97
heart-statlog	80.67	74.19	80.48
hayes-roth	75.12	69.21	74.91
hypothyroid	99.68	94.13	99.57
hepatitis	84.79	76.60	82.61
ecoli	96.30	89.78 ●	95.44
pima-diabetes	74.66	69.60 ●	73.61
haberman	67.75	64.49	67.25
flags	73.01	68.84	72.82

○, ● statistically significant improvement or degradation

Table 5.8: Exhaustive vs. Random vs. "pre-defined" with 4 clusters per class (AdaBoost + C4.5 as the base learner)

performance of its classification models improves noticeably compared with the random method.

Table 5.9 shows the results of the "pre-defined" method with different numbers of clusters. It appears that generally the accuracy decreases as the number of clusters increases. But this is not always the case. For example, the "pre-defined" codeword matrix with 10 clusters is better than those for 6, 7, 8 and 9 clusters. It appears that there is no clear pattern. The reason may be that these matrices are found using optimization based on random search. The performance depends on the quality of the matrix that is found.

Dataset	# 4	# 5	# 6	# 7	# 8	# 10	# 9
balance-scale	86.40	86.43	86.58	82.66 ●	86.38	86.49	86.43
breast-cancer	71.20	70.30	70.30	67.00	69.45	67.37	66.98
wisconsin-breast-cancer	95.97	95.18	94.86	87.44 ●	94.88	95.98	95.14
bridges-version1	83.49	82.18	80.75	76.99	82.46	79.24	79.16
horse-colic	83.58	82.68	80.70	78.66 ●	81.81	80.25	78.03 ●
horse-colic.ORIG	69.78	70.75	70.45	67.34	69.42	69.46	68.88
credit-rating	85.51	84.68	84.29	81.23 ●	83.67	84.74	83.13
german-credit	73.68	72.97	72.99	70.09 ●	72.87	71.57	69.55 ●
cylinder-bands	83.19	82.02	81.96	79.96	80.43	81.48	79.76
pima-diabetes	73.61	72.99	72.10	68.81	72.62	71.57	69.45 ●
ecoli	95.44	95.09	94.32	91.39 ●	93.19	94.28	92.23 ●
flags	72.82	72.21	71.31	68.14	70.84	72.27	69.56
haberman	67.25	68.21	68.01	62.46 ●	67.52	65.16	65.63
hayes-roth	74.91	75.34	74.76	73.04	75.28	76.15	76.76
heart-statlog	80.48	79.00	78.78	73.96	78.78	78.78	77.63
hepatitis	82.61	81.79	80.16	81.05	80.22	81.37	81.09
hypothyroid	99.57	99.49	99.36	87.08 ●	99.29 ●	99.18	99.07 ●
ionosphere	93.17	93.00	91.23	89.34	90.29	92.31	90.26
kr-vs-kp	99.47	99.41	99.32	92.27 ●	99.17	99.23	99.00 ●
labor	88.50	85.67	84.70	83.87	84.60	85.03	85.47
lymphography	84.40	80.74	79.49	81.02	82.07	83.47	82.37
mfeat	93.39	92.96	91.86 ●	88.45 ●	90.86 ●	91.68 ●	90.30 ●
mfeat	95.53	94.58	94.21	89.77 ●	92.59 ●	93.76 ●	92.66 ●
mushroom	100.00	100.00	100.00	92.89 ●	99.99	100.00	100.00
nursery	100.00	100.00	100.00	95.38 ●	99.98	100.00	99.99
optdigits	98.12	97.68	97.28 ●	91.93 ●	96.31 ●	97.25 ●	96.50 ●
sick	98.87	98.70	98.58	86.79 ●	98.59	98.45 ●	98.38 ●
sonar	81.70	81.85	80.21	79.68	78.16	82.18	77.37
spambase	95.43	94.99	94.47 ●	90.35	94.42 ●	94.61 ●	93.88 ●
splice	95.34	94.01 ●	92.38 ●	93.12 ●	93.66 ●	94.95	93.23 ●
trains	61.00	63.00	69.00	68.00	64.00	66.00	58.00
vehicle	96.96	96.65	95.48 ●	87.42 ●	95.89	95.34	94.08 ●
vote	94.98	95.00	94.23	89.14 ●	93.86	93.91	92.25 ●
waveform	86.07	84.30 ●	82.53 ●	84.16 ●		85.20	82.84 ●

○, ● statistically significant improvement or degradation

Table 5.9: The “pre-defined” method with different numbers of clusters

5.6 A summary of the experimental results

In this chapter, we have shown the experimental results of different methods. Overall, ECOCs with the exhaustive method is the best error correction method we have examined. When used with multi-clustering, it performs very well with other ensemble learners except the RandomForest classifier. It works best with AdaBoost as its base learner, but also exhibits some improvements using Bagging. The disadvantage of using the exhaustive method is that the time consumed for training the base learners can be very long. Due to the feature of the exhaustive method, we are only able to have 4 clusters per class on binary-class datasets. In this case, there are 127 columns and 8 rows in the codeword matrix. Other encoding methods are worse than exhaustive in term of accuracy, but the time consumed is much less. We have found that the “pre-defined” method is much better than the random method. However, the “pre-defined” method is not very stable, and the performance of this method really depends on the quality of the matrices that were found in the GPU-based random search.

Chapter 6

Conclusions and future work

In the following, we present some conclusions based on the results obtained and point out opportunities for future work.

6.1 Conclusions

This thesis focused on investigating an approach that can be used to apply multi-class techniques based on output codes to two-class datasets. The goal was to determine empirically whether this approach can yield higher accuracy classification models.

Enabling the application of multi-class techniques was achieved by turning two-class datasets into multi-class ones. This is done using clustering techniques that were discussed in Chapter 3. In that chapter, we discussed clustering techniques and focussed in particular on k-Means. k-Means is the technique that was used in the experiments, where we have observed how accuracy is affected by using different values of k .

With the resulting multi-class datasets in hand, different encoding and decoding methods were used in examining the performance of the output-code-based multi-class methods. We discussed several encoding methods in Chapters 2 and 4, such as 1-vs-1, 1-vs-all and the ECOC method. Different decoding methods, e.g., Hamming decoding, inverse Hamming decoding and Euclidean decoding, were also discussed in Chapter 4.

Exhaustive, random and "pre-defined" coding are the main encoding methods that were used in the experiment. For simplicity, all experiments were conducted using Hamming distanced based decoding. For finding good "pre-defined" matrices, GPU-based optimized random search has been used. This was discussed in detail in Chapter 4.

Based on the empirical result obtained in chapter 5, we can say that our new algorithm, which applies multi-class strategies on two-class data using clustering techniques, does improve accuracy for some base learners, when compare to applying them directly to two-class datasets, but not for all of them. AdaBoost and Bagging are the algorithms whose performance is increased significantly in several cases. However, the new algorithm does not perform well with RandomForest. Due to the larger number of trees used in a RandomForest ensemble the reader might wonder whether the new algorithm simply produced better results with AdaBoost because there were not enough boosted trees in the ensemble. To clear the doubt, we have performed an additional supplementary experiment where we have set the number of boosting iterations to 1000. The results are shown in Table A.2 (Appendix A), and we can see the new algorithm still improves the accuracy of AdaBoost models. Let us now consider the main findings of this thesis.

Exhaustive coding is the best coding method considered

Exhaustive coding is the best encoding method considered in the experiments, although the END method performs better for small numbers of clusters per class. Exhaustive coding uses all the possible unique bit columns in the codeword matrix. For a k-class problem, the number of bit columns is given by $2^k - 1$. Suppose the minimum Hamming distance of row separation is d, then we can correct up to $\frac{d-1}{2}$ bit errors. In fact, it is easy to see that d is 2^{k-2} for exhaustive codes. Therefore, for k-class problems, we can correct up to $\frac{2^{k-2}-1}{2}$ (rounded down) bit errors. To be able to correct at least 1 bit error ($\frac{2^{k-2}-1}{2} \geqslant 1$), it is required that $k > 3$. In order to be able to use ECOCs in a meaningful way, we need to satisfy $k > 3$, which means we have to turn two-class or three-class datasets into multi-class ones.

"Pre-defined" codes are better than random ones

The random coding method was used because the exhaustive coding method has its limitations: it cannot handle large numbers of classes (k) due to the time that is consumed for training the base learners. Rather than only consider using purely random coding, we also used random search to find good matrices and used these good matrices in the "pre-defined" method. The "pre-defined" coding method has higher error-correcting abilities than the completely random method as we discussed in Chapter 4.

GPU-based search for matrices achieves a large speed-up

To boost the speed of finding good matrices, we have used JavaCL to parallelize the computing process on a GPU. As mentioned in Chapter 4, by parallelizing the process, we can speed up more than 10 times the time required for running on the CPU. This is due to the fact that the random search for good matrices can be easily parallelized because generation and evaluation of a series of matrices can be performed in an independent thread. As an aside, we should also mention again that GPU programming is quite difficult to debug since the kernel function runs on the GPU. It is not straightforward to output a variable value, and use a command such as "System.out.println()" in Java.

6.2 Future work

The approach that we have evaluated in this thesis can be considered successful to some extent. However, due to time constraints, we have not had the chance to pursue all avenues considered. We would like to point out opportunities for future work.

Encoding methods

We have investigated quite several encoding methods and found that the "pre-defined" approach works better than the random one, but worse than the exhaustive one. However, the matrices used in the "pre-defined" coding method are not necessarily the best ones possible. We could potentially find better matrices by investing more compute time in random search. Another possible way to find a good matrices is that we may be able to discover an algebraic approach to calculate a good matrix directly. Such an approach, if efficient, would be a valuable contribution for such a problem.

Decoding methods

To decode, we have discussed 3 strategies: Hamming decoding, inverse Hamming decoding and Euclidean decoding. There are other state-of-the-art decoding strategies could be considered in future work. They are:

- Attenuated Euclidean decoding (Escalera & et al., 2007),
- Loss-based decoding with linear and exponential loss-functions (Allwein & Shapire, 2000),
- Probabilistic decoding (Passerini & Pontil, 2004),
- Laplacian decoding (Escalera & et al., 2006),

- Pessimistic β-density distribution decoding (Escalera & et al., 2006),
- Linear loss-weighted with discrete and continuous output of the classifier (Escalera & et al., 2008), and
- Exponential loss-weighted with discrete and continuous output of the classifier (Escalera & et al., 2008).

It would be interesting to investigate whether any of these other decoding methods could further increase the benefit of the methods investigated in this thesis.

Appendix A

Additional results

A.0.1 Bagging empirical results

A.0.2 AdaBoost with 1000 Boosting iteration trees

Dataset	Exh(2)	Exh(3)	Exh(4)	END(4)	END(3)	END(2)
balance-scale	87.33	88.13	88.72	88.87	89.10	88.11
breast-cancer	72.20	72.60	72.88	72.11	72.39	72.39
wisconsin-breast-cancer	95.90	96.31	96.77	96.70	96.57	96.41
bridges-version1	81.60	84.40	84.75	81.48	81.97	82.31
horse-colic	85.75	85.37	85.45	85.05	85.21	85.26
horse-colic.ORIG	69.32	69.97	68.86	67.98	67.61	68.86
credit-rating	85.68	86.16	86.26	86.74	86.46	86.29
german-credit	74.86	75.42	75.63	73.89	74.73	74.92
cylinder-bands	79.35	81.15	81.44	81.31	81.13	81.13
pima-diabetes	75.43	76.00	75.47	74.89	75.73	75.28
ecoli	96.01	96.43	96.54	96.13	96.48	96.34
flags	73.88	74.25	73.31	73.62	73.06	73.96
haberman	73.93	73.44	73.75	72.62	72.32	73.27
hayes-roth	79.70	79.98	80.39	79.45	78.76	80.23
heart-statlog	80.07	79.59	80.41	79.56	79.63	79.63
hepatitis	81.46	82.23	82.22	82.03	82.73	82.45
hypothyroid	99.61	99.54	99.51	99.37 •	99.41 •	99.48
ionosphere	92.37	93.08	93.31	92.82	92.79	92.99
kr-vs-kp	99.31	99.27	99.32	99.04	99.13	99.19
labor	87.47	87.17	86.80	85.10	84.40	87.63
lymphography	83.60	84.63	84.71	83.99	83.72	84.61
mfeat	92.25	93.72 ◦	94.32 ◦	94.28 ◦	93.98 ◦	93.30 ◦
mfeat	91.42	95.25 ◦	96.31 ◦	96.04 ◦	95.89 ◦	95.34 ◦
mushroom	100.00	100.00	100.00	100.00	100.00	100.00
nursery	100.00	100.00	100.00	100.00	100.00	100.00
optdigits	97.17	98.15 ◦	98.53 ◦	98.28 ◦	98.10 ◦	97.91 ◦
sick	98.87	98.77	98.78	98.54	98.61	98.73
sonar	79.79	82.45	83.21	81.73	83.18	81.97
spambase	94.48	94.82	95.04 ◦	95.13 ◦	95.03 ◦	94.81
splice	97.51	97.71	97.81	97.63	97.69	97.66
trains	53.00	51.00	51.00	47.00	44.00	44.00
vehicle	95.87	96.50	96.96 ◦	96.75	96.77	96.16
vote	96.62	96.66	96.57	96.32	96.43	96.20
waveform	86.81	88.32 ◦	88.80 ◦	88.86 ◦	88.75 ◦	88.24 ◦

◦, • statistically significant improvement or degradation

Table A.1: Exhaustive vs END with Bagging as their base learner

Dataset	Exh(2)	Exh(3)	Exh(4)	END(4)	END(3)	END(2)
balance-scale	87.33	88.13	88.72	88.87	89.10	88.11
breast-cancer	72.20	72.60	72.88	72.11	72.39	72.39
wisconsin-breast-cancer	95.90	96.31	96.77	96.70	96.57	96.41
bridges-version1	81.60	84.40	84.75	81.48	81.97	82.31
horse-colic	85.75	85.37	85.45	85.05	85.21	85.26
horse-colic.ORIG	69.32	69.97	68.86	67.98	67.61	68.86
credit-rating	85.68	86.16	86.26	86.74	86.46	86.29
german-credit	74.86	75.42	75.63	73.89	74.73	74.92
cylinder-bands	79.35	81.15	81.44	81.31	81.13	81.13
pima-diabetes	75.43	76.00	75.47	74.89	75.73	75.28
ecoli	96.01	96.43	96.54	96.13	96.48	96.34
flags	73.88	74.25	73.31	73.62	73.06	73.96
haberman	73.93	73.44	73.75	72.62	72.32	73.27
hayes-roth	79.70	79.98	80.39	79.45	78.76	80.23
heart-statlog	80.07	79.59	80.41	79.56	79.63	79.63
hepatitis	81.46	82.23	82.22	82.03	82.73	82.45
hypothyroid	99.61	99.54	99.51	99.37 ●	99.41 ●	99.48
ionosphere	92.37	93.08	93.31	92.82	92.79	92.99
kr-vs-kp	99.31	99.27	99.32	99.04	99.13	99.19
labor	87.47	87.17	86.80	85.10	84.40	87.63
lymphography	83.60	84.63	84.71	83.99	83.72	84.61
mfeat	92.25	93.72 ○	94.32 ○	94.28 ○	93.98 ○	93.30 ○
mfeat	91.42	95.25 ○	96.31 ○	96.04 ○	95.89 ○	95.34 ○
mushroom	100.00	100.00	100.00	100.00	100.00	100.00
nursery	100.00	100.00	100.00	100.00	100.00	100.00
optdigits	97.17	98.15 ○	98.53 ○	98.28 ○	98.10 ○	97.91 ○
sick	98.87	98.77	98.78	98.54	98.61	98.73
sonar	79.79	82.45	83.21	81.73	83.18	81.97
spambase	94.48	94.82	95.04 ○	95.13 ○	95.03 ○	94.81
splice	97.51	97.71	97.81	97.63	97.69	97.66
trains	53.00	51.00	51.00	47.00	44.00	44.00
vehicle	95.87	96.50	96.96 ○	96.75	96.77	96.16
vote	96.62	96.66	96.57	96.32	96.43	96.20
waveform	86.81	88.32 ○	88.80 ○	88.86 ○	88.75 ○	88.24 ○

○, ● statistically significant improvement or degradation

Table A.2: Exhaustive vs END with Bagging as their base learner

Appendix B

Source code

B.0.3 Kernel function

B.0.4 JavaCL code

Program 1 Kernel function that runs on the GPU for finding good matrices

```
private String getKernelFunction() {
String src =

// random number generation function
" typedef struct { ulong a, b, c } random_state; \n"

    + " unsigned long random(random_state *r){ \n"
    + "  unsigned long old = r->b;\n"
    + "  r->b = r->a * 1103515245 + 12345;\n"
    + "  r->a = (~old ^ (r->b >> 3)) - r->c++;\n"
    + "  return r->b;\n"
+ " }\n"

+ " void seed_random(random_state *r, ulong seed){ \n"
+ "  r->a = seed;\n"
+ "   r->b = 0;\n"
+ "    r->c = 362436;\n"
+ "}\n"
// END random number generation

// kernel function starts here
+ "__kernel void calculate(__global long* score, __global int* input,
 __global long* output)\n"
+ "{\n"
+ " __private int id = get_global_id(0);\n"
// assume there are 16 rows and 16x2 = 32 columns
+ " int row = 4; \n"
// we use one long to represent one row.
+ " long bestMatrix[4];\n"
+ " long tempMatrix[4];\n"
```

Program 2 Kernel function, continued

```
// the largest number that 32(the number of columns) bit binary string
// can represent max = 2^(column) -1 ; in this example, 2^32 -1
+ " long max = 255;\n"

// initialization
+ " int bestScore = 0;\n"
+ " int bestID = -1;\n"
+ " for (int g =0 ; g < row; g++) { \n"
+ "  bestMatrix[g] = 0;\n"
+ " }\n"

// input[0] and input[1] are random seeds that passed in from host program
+ "  int xoff=input[0];\n"
+ "  int yoff=input[1];\n"
+ "  int x = get_global_id(0) + xoff*get_global_size(0); \n"
+ "  int y = get_global_id(1) + yoff*get_global_size(1);\n"
+ "  random_state randstate;\n"
+ "  seed_random(&randstate, x + y*640);\n"

// assume that we wish to generate one 1000 matrix per run.
+ " for(long run =0 ; run< 1000; run++){ \n"
// generate one random matrix that represented using row number of long
// numbers
+ "  for (int g =0 ; g < row; g++) { \n"
+ "  long value = random(&randstate)&max;\n"
+ "  tempMatrix[g] = value;\n"
+ "  } \n"

// calculate current matrix score
+ "  int tempScore= 100;\n"
+ "  for(int i =0 ; i < row-1; i++){ \n"
+ "  for(int j = i+1; j < row; j++) {\n"
+ "  long rowDiff = tempMatrix[i]^tempMatrix[j]; \n"
+ "  int numberOfOnes =0; \n"
+ "  while(rowDiff!=0){\n"
+ "  long d = rowDiff%2;\n"
+ "  if(d==1){\n"
+ "  numberOfOnes++;\n"
+ "  }\n"
+ "  rowDiff/=2;\n"
+ "   }\n"
+ "  if(numberOfOnes < tempScore){\n"
```

Program 3 Kernel function, continued

```
+ "  tempScore = numberOfOnes;\n"
+ "  }\n"
+ "  }\n"
+ "  }\n"

// check if it is valid matrix if current matrix score is large the
// one we have got highest score.
+ "  if (tempScore > bestScore){ \n"
// Restructure the matrix, transfer the 16 long numbers into 32 long
// numbers, which should represent the same matrix
+ "  long columns[8]; \n"
+ "  for(int i =0 ; i < 2*row ; i++){\n "
+ "  long tempValue =0; \n"
+ " for(int j=0; j < row; j++){ \n"
+ "  long temp = tempMatrix[j] >> i & 1; \n"
+ " if(temp==1){\n"
+ " long lValue =1; \n"
+ " for(int time =0; time< j ; time ++) {\n"
+ " lValue*=2;\n"
+ "  }\n"
+ "  tempValue+=lValue;\n"
+ "  }\n"
+ "  }\n"
+ "  columns[i] = tempValue; \n"
+ "  }\n"


// validate the matrix
+ "  bool isOk = true;\n"
// check if there are any same column
+ "  for (int i = 0; i < 2*row - 1; i++) { \n"
+ "  for (int j = i + 1; j < 2*row; j++) {"
+ "  if ((columns[i] ^ columns[j]) == 0) { \n"
+ "  isOk = false;\n"
+ "  break;\n"
+ " }\n"
+ "  if (!isOk) {\n"
+ "  break;\n"
+ "  }\n"
+ "  }\n"
+ "  }\n"
```

Program 4 Kernel function, continued

```
// check if any columns that inverse of others
+ "  if (isOk) {\n"
+ " long lValue =1; \n"
+ " for(int time =0; time< row ; time ++) {\n"
+ " lValue*=2;\n"
+ "  }\n"
+ "  long sum = lValue-1; \n"
+ "  for (int i = 0; i < row*2 - 1; i++) {\n"
+ "  for (int j = i + 1; j < row*2; j++) {\n"
+ "  if (((columns[i] & columns[j]) == 0)&&
((columns[i] | columns[j]) == sum)) {\n"
+ "  isOk = false;\n"
+ "  break;\n"
+ "  }\n"
+ "  if (!isOk) {\n"
+ "  break;\n"
+ "  }\n"
+ "  }\n"
+ "  }\n"
+ "  }\n"

// check if any column has only 0s or 1s
+ "  if (isOk) {\n"
+ " long lValue =1; \n"
+ " for(int time =0; time< row ; time ++) {\n"
+ " lValue*=2;\n"
+ "  }\n"
+ "  long maxValue = lValue-1; \n"
+ "  for (int i = 0; i < row*2; i++) {\n"
+ "  if (columns[i] == 0 && columns[i] == maxValue) {\n"
+ "  isOk = false;\n"
+ "  break;\n"
+ "  }\n"
+ "  }\n"
+ "  }\n"

// save the matrix if the matrix is valid and its score is higher
// than saved score
+ "  if (isOk && (tempScore > bestScore)) {\n"
+ "      bestScore = tempScore;\n"
+ " bestID = id;\n"
```

Program 5 Kernel function, continued

```
+ "  for(int i =0 ; i < row; i++) {\n"
+ "  bestMatrix[i] = tempMatrix[i];\n"
+ "  }\n"
+ "  }\n"

+ "  }\n" // END |if (tempScore > bestScore)|

+ " }\n"// END FOR

// after 100 generations, we compare the best matrix we found in
// current kernel with the matrix in global memory. if current is
// better, write to global memory
// the BestID just shows which kernel found the best matrix.
+ " if (bestScore > score[0]) {\n"
+ " score[0] = bestScore;\n"
+ " score[1] = bestID;\n"
+ "  for(int i =0 ; i < row; i++) {\n"
+ "  score[i+5] = bestMatrix[i];\n"
+ "  }\n"
+ " }\n"
+ " output[id] = bestScore;\n"

+ "}\n" // END kernel function
;

return src;

}
```

Program 6 JavaCL code: used to initialize OpenCL and push kernel to GPU

```
  package fz41.matrix;

/**
 * @author fanhua
 */

import java.io.BufferedWriter;
import java.io.FileWriter;
import java.nio.IntBuffer;
import java.nio.LongBuffer;
import java.util.Random;
import com.nativelibs4java.opencl.CLBuildException;
import com.nativelibs4java.opencl.CLContext;
import com.nativelibs4java.opencl.CLEvent;
import com.nativelibs4java.opencl.CLIntBuffer;
import com.nativelibs4java.opencl.CLKernel;
import com.nativelibs4java.opencl.CLLongBuffer;
import com.nativelibs4java.opencl.CLMem;
import com.nativelibs4java.opencl.CLMem.MapFlags;
import com.nativelibs4java.opencl.CLProgram;
import com.nativelibs4java.opencl.CLQueue;
import com.nativelibs4java.opencl.JavaCL;

import fz41.model.CLFunction;

/**
 *
 * 1. initialize OpenCL;
 * 2. passes the data to CL kernel and asks the kernel do jobs;
 * 3. get the output from kernel;
 * 4. measure the running time;
 */
public class Matrix {

public Matrix(int size, int row) {
// give extra 5 column to store data.
this.scoreSize = 5 + row;
this.row = row;
this.size=size;
this.fileName = "Matrix_"+row+".txt";
initializeOpenCL(size);
}
```

Program 7 JavaCL code, continued

```
/**
 * This method is called from within the constructor to initialize the
 * OpenCl.
 */
private void initializeOpenCL(int size) {

context = JavaCL.createBestContext();
// we have found that 48 has a very good performance
tasks = 48 * size;
queue = context.createDefaultQueue();

score = context.createLongBuffer(CLMem.Usage.InputOutput, scoreSize);
memIn = context.createIntBuffer(CLMem.Usage.InputOutput, 2);
memOut = context.createLongBuffer(CLMem.Usage.InputOutput, tasks);
String src = f.getKernelFunction();

try {
program = context.createProgram(src).build();
a = memIn.map(queue, MapFlags.Write);
kernel = program
.createKernel("calculate",score, memIn, memOut);
} catch (CLBuildException e) {
e.printStackTrace();
}
}

/**
 * call the kernel; and
 * copy the output to input after each generation.
 */
private void runCL() {
System.out.flush();
int seed1 = random.nextInt();
int seed2 = random.nextInt();
System.out.println("Seed 1: " + seed1 + "  seed 2: " + seed2);

a.put(new int[]{seed1, seed2});
CLEvent readCompletion = memIn.unmap(queue, a);
readCompletion.waitFor();

CLEvent kernelCompletion = kernel.enqueueNDRange(queue,
new int[] { 48*size },
new int[] { size });
```

Program 8 JavaCL code, continued

```
kernelCompletion.waitFor();
queue.finish();
a.clear();
}

/**
 * where we start the openCL
 */
public void gpu() {
long initTime = System.nanoTime();
long bestScore = 0;
// we have set 25 as the number iteration, this can be any valid long
//number.
for (long a = 0; a < 25 ; a++) {

long start = System.nanoTime();
runCL();

LongBuffer scoreArray = score.read(queue);
long end = System.nanoTime();
double time = (end-start)/(1000.0*1000*1000);


long score = scoreArray.get(0);

if(score>bestScore){
//save result
double estimatedTime = (end-initTime)/(1000.0*1000*1000);
bestScore = score;
saveResult(scoreArray, a , estimatedTime);
}

for(int i =0 ; i < scoreSize; i ++){
System.out.println(scoreArray.get(i));
}

}

}

private void saveResult(LongBuffer scoreArray,
long iteration, double estimatedTime){
```

Program 9 JavaCL code, continued

```
try {
FileWriter fstring = new FileWriter(fileName, true);
BufferedWriter out = new BufferedWriter(fstring);
out.write("--------iteration : "+iteration+"---------\n");
for(int i =0 ; i < scoreSize; i ++){
out.write(scoreArray.get(i)+"\n");
}
out.write("Estimated time since start : "+estimatedTime +" seconds\n");

out.close();
} catch (Exception ex) {
System.err.println("error : " + ex.getMessage());
}
}



private String fileName ="";

private int scoreSize =0;

CLFunction f = new CLFunction();
private IntBuffer a;
private CLContext context;
private int tasks;
private CLQueue queue;
private CLLongBuffer score;
private CLLongBuffer memOut;
private CLIntBuffer memIn;
private int row = 0;
private Random random  = new Random();
private int size ;
private CLProgram program;
private CLKernel kernel;

}
```

Appendix C

Good matrices found by GPU

Note that one number (need to transfer to binary) represents one row, for example:

$182_{10} = 10110110_2$

$255_{10} = 11111111_2$

$152_{10} = 10011000_2$

$25_{10} = 00011001_2$

Therefore, numbers 182, 255, 152, 25 represent one matrix :

10110110

11111111

10011000

00011001

Bellows are the matrices (We only list large matrices here):

C.0.5 Matrix size: 12×24

8664803

96530

5835814

13047560

2942910

825465

7273026

11864917

8125117

12757886

7636353

12309704

Estimated time to find: 32158.721219 seconds

C.0.6 Matrix size: 14×28

minimum Hamming distance: 12

40543007

69514957

41289963

149721148

22005475

215248786

175760456

238230937

161876940

135778782

221808294

245449506

93891933

12668213

Estimated time to find : 140.019364 seconds

C.0.7 Matrix size: 16×32

minimum Hamming distance: 13

1135148457

3913886139

2131870117

2566833819

1543223507

4062729143

272077046

3772597717

723921409

1790695039

3559555533

753524552

554096469

2792292418

2649787190

2053628174

Estimated time to find : 10.139268 seconds

C.0.8 Matrix size: 18×36

minimum Hamming distance: 15

63826692481

53590797047

44711116888

8585570706

10042523215

56143186757

21164629500

48219449016

50165972334

13846725727

11631432034

64497631871

7762800352

4341169205

13111620545

10169565433

24876528079

18030679336

Estimated time to find : 9754.886285 seconds

C.0.9 Matrix size: 20×40

minimum Hamming distance: 16

838074193360

920840877031

729696079223

1053710409777

572157676988

10169727070

340557433364

230383396006

438348780197

255144386478

248993409851

882712786505

719610259815

330989526787

943242158876

806522257994

865494084236

207504686289

881487067269

26916319561

Estimated time to find: 49.109626 seconds

C.0.10 Matrix size: 22×44

minimum Hamming distance: 18

1756526480523

```
4790075684140

6217052350383

12927012866931

17072415891006

4907058674010

1866966592416

2941471454606

13340108721274

13273220211165

3668278467632

4122735031473

15257647012949

7874091086967

13219003267818

10866642165978

14486708659378

3662173893766

5079114503675

5451849111491

11167797120852

6932088334777

Estimated time to find : 75500.267253 seconds
```

C.0.11 Matrix size: 24×48

minimum Hamming distance: 19

```
159162892404724

9980684591297

175278664684711

96978137619845

63082043235454

46267428365978

90313613051590

226880412406248

174792761239734

86486476873474

144239136935197

112335360353729

109806592517792

198205374483843

43721350498615

226826934211673

23040175974177

150308518129991

182677232452222

116324277856894

252016349089436

7091744825187

83152682012399

280326589509196

Estimated time to find : 31.173693 seconds
```

C.0.12 Matrix size: 26×52

minimum Hamming distance: 21

77104675753898

219347532513154

89976172103721

266329959101075

96287838216514

122756080478101

174931506177239

186722419041015

269459605623704

147317097641292

216623853511478

56286907620804

157432919781827

11302438117760

60231565523688

167945453604057

176770638591630

86277298154672

104307140856435

186418071264280

111716425614241

172868276485736

70982871283707

38863492159807

Estimated time to find : 257.107396 seconds

C.0.13 Matrix size: 28×56

minimum Hamming distance: 22

18038432589466759

34777901281365257

55292730450849481

62194133028861178

70848512679439999

31413930819139359

56360459930752983

4701323216890709

70486235868413168

55004786109432461

13102564031842818

29028090152900057

18184915187877616

7235374170769229

49566841106063414

2471703258541951

57828362962509468

24128433066569304

19841427932348188

61864310472144923

7098918954403049O

37763660835922952

55212408773523060

61142796619580088

22868330697293746

37672610728569334

19420569079909411

43365289407162056

Estimated time to find : 196.873369 seconds

C.0.14 Matrix size: 30×60

minimum Hamming distance: 24

186742836881620287

927718617525950555

391541814579880871

909416904262238043

414820338076454549

113719769810751023

862903777581494535

1025400882138007367

1129325489460507770

707585768445257896

65696989655954976

920366522863485434

471067541415193174

```
133602640893489708

166929403000544408

588146158585013231

776547771697751636

728946605358773927

164838716489069179

1006498514726631366

949645032845639258

713957603141681707

15368405268005396

118572268096046117

49344768036018973

502865304162139954

871995372474307764

123173407095994450

667561222622954707

2175682158292260

Estimated time since start : 37724.820602 seconds
```

Bibliography

Allwein, E. & Shapire, R. (2000). Reducing multiclass to binary: A unifying approach for margin classifiers. In *Journal of Machine Learning Research* (pp. 113–141).

Aly, M. (2005). Survey on multiclass classification methods. Technical report, California Institute of Technology.

Barnhill, S. & Vapnik, V. (2002). Gene selection for cancer classification using support vector machines. In *Machine Learning* (pp. 389–442).

Breiman, L. (1996). *Bagging predictors, Machine Learning 24: 123-140* (2 Ed.). Morgan Kaufmann.

Breiman, L. & et al. (1984). Classification and Regression Trees. Technical report, Wadsworth International Group.

Dietterich, T. G. & Bakiri, G. (1995). Solving multiclass learning problems vis error-correcting ouput code. In 2 (Ed.), *Journal of Artificial Intelligence Research* (pp. 263–286).

Escalera, S. & et al. (2006). Decoding of ternary error correcting output codes. In *n: CIARP, vol. 4225* (p. 753763).

Escalera, S. & et al. (2007). Boosted landmarks of contextual descriptors and forest-ecoc: A novel framework to detect and classify objects in clutter scenes. In *Pattern Recognition Lett* (pp. 1759–1768).

Escalera, S. & et al. (2008). Loss-weighted decoding for error-correcting output codes. In *In: Internat. Conf. on Computer Vision Theory and Applications* (pp. 117–122).

Escalera, S. & Pujol, O. (2010). Error-correcting ouput code library. In Ong, C. S. (Ed.), *Journal of Machine Learning Research 11* (pp. 661–664). Bellaterra, Barcelona, Spain.

Frank, E. & Kramer, S. (2004). Ensembles of nested dichotomies for multi-class problems. In *In Proc 21st International Conference on Machine Learning* (pp. 305–312). Banff, Canada: ACM Press.

Freund, Y. & Schapire, R. E. (1995). A Decision-Theoretic Generalization of on-Line Learning and an Application to Boosting . Technical report.

Hamming, R. W. (1950). Error detecting and error correcting codes. In *Bell System Technical Journal 29* (pp. 147–160).

Ho, T. (1995). Random decision forest. In *3rd Int'l Conf. on Document Analysis and Recognition* (p. 278282).

L., D. & et al. (1996). A probabilistic theory of pattern recog- nition. New York: Springer-Verlag.

Passerini, A. & Pontil, M. (2004). New results on error correcting output codes of kernel machines. In *IEEE Trans. Neural Networks 15 (1)* (pp. 45–54).

Quinlan, J. R. (1993). *C4.5: Programs for Machine Learning.* Morgan Kaufmann.

Rifkin, R. & Klautau, A. (2004). Parallel networks that learn to pro- nounce english text. In *Journal of Machine Learning Research* (p. 101141).

Sejnowski, T. & Rosenberg, C. R. (1987). Parallel networks that learn to pronounce english text. In 1 (Ed.), *Complex Syst* (pp. 145–168).

Witten, I. H. & Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques* (2 Ed.). San Francisco, CA: Morgan Kaufmann.