Exploiting Code Redundancies in ECOC

Sang-Hyeun Park, Lorenz Weizsäcker, and Johannes Fürnkranz

Knowledge Engineering Group, TU Darmstadt, Germany
{park,lorenz,juffi}@ke.tu-darmstadt.de

Abstract. We study an approach for speeding up the training of error-correcting output codes (ECOC) classifiers. The key idea is to avoid unnecessary computations by exploiting the overlap of the different training sets in the ECOC ensemble. Instead of re-training each classifier from scratch, classifiers that have been trained for one task can be adapted to related tasks in the ensemble. The crucial issue is the identification of a schedule for training the classifiers which maximizes the exploitation of the overlap. For solving this problem, we construct a classifier graph in which the nodes correspond to the classifiers, and the edges represent the training complexity for moving from one classifier to the next in terms of the number of added training examples. The solution of the Steiner Tree problem is an arborescence in this graph which describes the learning scheme with the minimal total training complexity. We experimentally evaluate the algorithm with Hoeffding trees, as an example for incremental learners where the classifier adaptation is trivial, and with SVMs, where we employ an adaptation strategy based on adapted caching and weight reuse, which guarantees that the learned model is the same as per batch learning.

1 Introduction

Error-correcting output codes (ECOC) [5] are a well-known technique for handling multiclass classification problems, i.e., for problems where the target attribute is a categorical variable with $k > 2$ values. Their key idea is to reduce the k-class classification problem to a series of n binary problems, which can be handled by a 2-class classification algorithm, such as a SVM or a rule learner. Conventional ECOC always use the entire dataset for training each of the binary classifiers. Ternary ECOC [1] are a generalization of the basic idea which allows to train the binary classifiers on subsets of the training examples. For example, pairwise classification [8, 9], which trains a classifier for each pair of classes, is a special case of this framework.

For many common general encoding techniques, the number of binary classifiers may exceed the number of classes by several orders of magnitude. This allows for greater distances between the code words, so that the mapping to the closest code word is not compromised by individual mistakes of a few classifiers. For example, for pairwise classification, the number of binary classifiers is quadratic in the number of classes. Thus, the increase in predictive accuracy comes with a corresponding increase in computational demands at classification time. In previous work [11], we focused on fast ECOC decoding methods, which tackled this problem. For example, for the special case of pairwise classification, the quadratic complexity can be reduced to $O(k \log k)$ in practice.

In this paper, we focus on the training phase, where overlaps of training instances in highly redundant codes are reduced without altering the models. This is done by identifying shared subproblems in the ensemble, which need to be learned only once, and by rescheduling the binary classification problems so that these subproblems can be reused as often as possible. This approach is obviously feasible in conjunction with incremental base learners, but its main idea is still applicable for the more interesting case when SVMs are used as base learners, by reusing computed weights of support vectors from related subproblems and applying an adapted ensemble caching strategy.

At first, we will briefly recapitulate ECOC with an overview of typical code designs and decoding methods (section 2) before we discuss their redundancies and an algorithm to exploit them in Section 3. The performance of this algorithm is then evaluated for Hoeffding trees and for SVMs as base classifiers (Section 4). Finally, we will conclude and elaborate on possible future directions.

2 Error-Correcting Output Codes

Error-correcting output codes [5] are a well-known technique for converting multi-class problems into a set of binary problems. Each of the k original classes receives a code word in $\{-1, 1\}^n$, thus resulting in a $k \times n$ coding matrix M. Each of the n columns of the matrix corresponds to a binary classifier where all examples of a class with $+1$ are positive, and all examples of a class with -1 are negative. Ternary ECOC [1] are an elegant generalization of this technique which allows 0-values in the codes, which correspond to ignoring examples of this class.

As previously mentioned, the well known *one-against-one* and *one-against-all* decomposition schemes for multiclass classification are particular codes within the framework of ECOC. Other well-known general codes include:

Exhaustive Ternary Codes cover all possible classifiers involving a given number of classes l. More formally, a (k, l)-exhaustive ternary code defines a ternary coding matrix M, for which every column j contains exactly l values, i.e., $\sum_{i \in K} |m_{i,j}| = l$. Obviously, in the context of multiclass classification, only columns with at least one positive ($+1$) and one negative (-1) class are meaningful. These codes are a straightforward generalization of the exhaustive binary codes, which were considered in the first works on ECOC [5], to the ternary case. Note that $(k, 2)$-exhaustive codes correspond to pairwise classification.

In addition, we define a cumulative version of exhaustive ternary codes, which subsumes all (k, i)-codes with $i = 2 \dots l$, so up to a specific level l. In this case, we speak of (k, l)-*cumulative exhaustive codes*. For a dataset with k classes, (k, k)-cumulative exhaustive codes represent the set of all possible binary classifiers.

Random Codes are randomly generated codes, where the probability distribution of the set of possible symbols $\{-1, 0, 1\}$ can be specified. The zero probability parameter $r_{zp} \in [0, 1]$, specifies the probability for the zero symbol, $p(\{0\}) = r$, whereas the remainder is equally subdivided to the other symbols: $p(\{1\}) = p(\{-1\}) = \frac{1-r}{2}$. This type of code allows to control the degree of *sparsity* of the ECOC matrix. In accordance with the usual definition, we speak of *random dense codes* if $r_{zp} = 0$, which relates to binary ECOC.

3 Redundancies within ECOC

3.1 Code Redundancy

Many code types specify classifiers which share a common code configuration. For instance, in the case of exhaustive cumulative k-level codes, we can construct a *sub-classifier* by setting some $+1$ bits and some -1 bits of a specified classifier to zero. Clearly, the resulting classifier is itself a valid classifier that occurs in the ECOC matrix of this cumulative code. Furthermore, every classifier f of length $l < k$, is subclassifier of exactly $2 \cdot (n - l)$ classifiers with length $l + 1$, since there are $n - l$ remaining classes and each class can be specified as positive or negative. Such redundancies also occur frequently in random codes with a probability of the zero-symbol smaller than 0.5, and therefore also in the special case of random dense codes, where the codes consists only of $+1$ and -1 symbols. On the other hand, the widely used one-against-one code has no code redundancy, and the redundancy of the one-against-all code is very low.

In general, the learning of a binary classifier is independent of the explicit specification, which class of instances is regarded as positive and which one as negative. So, from a learning point of view, the classifier specified by a column $\boldsymbol{m}_i = (m_{1i}, \ldots, m_{ki})$ is equivalent to $-\boldsymbol{m}_i$.

Formally, code redundancy can be defined as follows:

Definition 1 (Code Redundancy). *Let f_i and f_j be two classifiers and $(m_{1i}, \ldots, m_{ki})$ and $(m_{1j}, \ldots, m_{kj})$ their corresponding (ternary) ECOC columns. We say f_i and f_j are $p-$*redundant*, if for $a \in \{1 \ldots k\}$,*

$$p = \max(\#\{a \,|\, m_{ai} = m_{aj}, m_{ai} \neq 0\}, \#\{a \,|\, m_{ai} = -m_{aj}, m_{ai} \neq 0\})$$

Let $d = \max(d_H(\boldsymbol{m}_i, \boldsymbol{m}_j), d_H(-\boldsymbol{m}_i, \boldsymbol{m}_j))$, where d_H is the Hamming distance. Two classifiers f_i and f_j are p-redundant, if and only if $k - p = d - \#\{a \in \{1 \ldots k\} \,|\, m_{ai} = 0 \wedge m_{aj} = 0\}$. Thus, in essence, classifier redundancy is the opposite of Hamming distance except that bit positions with equal zero values are ignored. For convenience, similarly to the symmetric difference of sets, we denote for two classifiers f_i and f_j the set of classes which are only involved in one of their code configurations m_i and m_j as $f_i \triangle f_j$. More precisely, $f_i \triangle f_j = \{c_a \,|\, a \in \{1 \ldots k\} \wedge |m_{ai}| + |m_{aj}| = 1\}$. In addition, we speak of a *specified* classifier, if there exists a corresponding code-column in the given ECOC matrix.

3.2 Exploitation of Code Redundancies

Code redundancies can be trivially exploited by incremental base learners, which are capable of extending an already learned model on additional training instances. Then, repeated iterations over the same instances can be avoided, since shared subclassifiers only have to be learned once. The key issue is to find a training protocol that maximizes the use of such shared subclassifiers, and therefore minimizes the redundant computations. Note that the subclassifiers do not need to be specified classifiers, i.e., they do not need to correspond to a class code in the coding matrix.

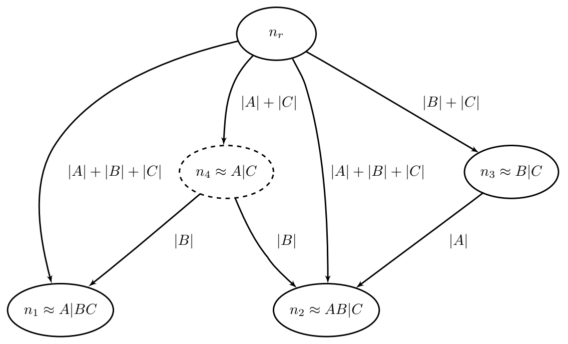

Fig. 1. A sample training graph. Three classifiers $f_1 = A|BC$, $f_2 = AB|C$ and $f_3 = B|C$ are specified. The non-specified classifier $f_4 = A|C$ is added because it is the maximal common subclassifier of f_1 and f_2. For each edge $e_{ij} = (n_i, n_j)$ the weights depict the training effort for learning classifier f_j based on classifier f_i ($|A|$ is the number of training instances of class A).

This task may be viewed as a graph-theoretic problem. Let $G = (V, E)$ be a weighted directed graph with $V = \{n_r\} \cup \{f_i\} \cup \{f_s\}$, i.e., each classifier f_i and each possible subclassifier f_s are in the set of nodes V. Furthermore, the special *root* node n_r is connected to every other node $n_i \in V$ with the directed edge (n_r, n_i). Besides, for each two non-root nodes n_i and n_j, there exists a directed edge (n_i, n_j), if and only if n_i is subclassifier of n_j. The weight of these edges is $f_j \bigtriangleup f_i$. For all edges (n_r, n_i), which are incident to the root node, the weight is the number of training instances involved in classifier f_i.

To elaborate, incident edges to the root node depict classifiers which are learned by batch learning. All other edges (n_i, n_j), which are edges between two (sub)-classifiers, represent incremental learning steps. Based on the learned model of classifier f_i, the remaining training instances of $f_j \bigtriangleup f_i$ are used to learn classifier f_j. The multiple possible paths to one particular classifier represents the possible ways to learn it. Each of these paths describe a different partitioning of training costs, represented by the number of edges (number of partitions) and edge weights (size of the partitions). Considering only one classifier, the cost for all paths are identical. But, by considering that paths of different redundant classifiers can overlap, and that shared subpaths are trained only once, the total training cost can be reduced. Another view at this graph is the following: every subgraph of G which is an arborescence consisting of all specified classifiers is a *valid* scheme for learning the ensemble, in the sense that it produces exactly the specified set of classifiers.

In this context, our optimization problem is to find a minimum-weight subgraph of G including all classifier nodes f_i, which relates to minimizing the processed training instances for the set of specified classifiers and therefore total training complexity of the ECOC ensemble. Note, this problem is known in graph theory as *Steiner problem in a directed graph*, which is NP-hard [14].

Figure 1 shows an example of such a *training graph* for a 3-class problem, where three classifiers $f_1 = A|BC$, $f_2 = AB|C$ and $f_3 = B|C$ are specified by a given

Algorithm 1 Training Graph Generation

```
Require: ECOC Matrix $\boldsymbol{M} = (m_{i,j}) \in \{-1,0,1\}^{k \times n}$, binary classifiers $f_1, \ldots, f_n$
 1: $V = \{n_r\}, E = \varnothing$
 2: for each $f_i$ do
 3:     $V = V \cup \{n_i\}$                                   # Integration of all classifiers
 4:     $e_{ri} = (n_r, n_i)$
 5:     $w(e_{ri}) = I(f_i)$
 6:     $E = E \cup \{e_{ri}\}$
 7: end for
 8: for $l = k$ downto 2 do
 9:     $F = \{n \in V \backslash \{n_r\} \mid \mathsf{length}(n) \geq l\ ,\ \mathsf{seen}(n) = 0\}$     # level-wise subclassifier generation
10:     for each pair $(n_i, n_j) \in F \times F$ with $i \neq j$ do
11:         $n_s = \mathsf{intersection}(n_i, n_j)$          # generate shared subclassifier of $f_i$ and $f_j$
12:         if $n_s$ is valid then
13:             if $n_s \notin V$ then
14:                 $V = V \cup \{n_s\}$                       # classifier is new
15:                 $e_{rs} = (n_r, n_s)$
16:                 $w(e_{rs}) = I(f_s)$
17:                 $E = E \cup \{e_{rs}\}$
18:             end if
19:             $e_{si} = (n_s, n_i)\ ,\ e_{sj} = (n_s, n_j)$
20:             $w(e_{si}) = I(f_s \bigtriangleup f_i)$
21:             $w(e_{sj}) = I(f_s \bigtriangleup f_j)$
22:             $E = E \cup \{e_{si}, e_{sj}\}$
23:         end if
24:     end for
25:     $\forall n \in F.\mathsf{seen}(n) = 1$                  # mark as processed, see also note in text
26: end for
27: return $G = (V, E, w)$
```

ECOC matrix. A, B, C are symbol representatives for classes and $A|B$ describes the binary classifier which discriminates instances of class A against B. The standard training scheme, which learns each classifier separately, can be represented as a subgraph $G_1 \subseteq G$ consisting of $V_1 = \{n_r, n_1, n_2, n_3\}$ and $E_1 = \{e_{r1}, e_{r2}, e_{r3}\}$. This scheme uses $2|A| + 3|B| + 3|C|$ training instances in total. An example where fewer training instances are needed is $G_2 = (V_1, E_2)$ with $E_2 = \{e_{r1}, e_{r3}, e_{32}\}$, which exploits that classifier f_2 can be incrementally trained from f_3, resulting in training costs $2|A| + 2|B| + 2|C|$. Another alternative is to add a non-specified classifier $f_4 = A|C$ to the graph, resulting in $G_3 = (V, E_3)$ with $E_3 = \{e_{r4}, e_{41}, e_{42}, e_{r3}\}$ with training costs $|A| + 3|B| + 2|C|$. It is easy to see that either G_2 or G_3 is the optimal Steiner Tree in this example and that both process fewer training examples than the standard scheme. Whether G_2 or G_3 is optimal, depends on whether $|A| > |B|$.

Since the optimal solution is in general hard to compute, we use a greedy approach. We first have to generate the training graph. Then, we iteratively remove local non-optimal edges, starting from the leaf nodes (specified classifiers) up to the root. Both methods are described in detail in the following subsections.

Generation of Training Graph We consider an algorithm which is particularly tailored for exhaustive and exhaustive cumulative codes. Let F be the set of all classifiers f of a specific length l, which is successively decreased from k down to 2. For each pair $(f_i, f_j) \in F \times F$ the maximal common subclassifier f_s is determined and eventually integrated into the graph. Then, these classifiers are marked as processed ($\mathsf{seen}(f) = 1$) and are not considered in the following steps of the generation algorithm. Level l is decreased and the algorithm repeats. The processed classifiers can be ignored, because for the systematic codes (exh. and exh. cumulative) all potential subclassifiers can be constructed using its immediate subclassifiers. This algorithm does not find all edges for random codes or general codes, but only for their inherent systematic code structures. For the sake of efficiency and also considering that we employ a greedy Steiner Tree Solving procedure afterwards, we neglect this fact.

A pseudo code is given in Algorithm 1. Note that there, the set F is populated with classifiers of length greater equal than l instead of exactly l, considering the special case that there can be multiple levels with zero classifiers or only one classifier. Also, classifiers should only be flagged as *processed* if they were actually checked at least once. The complexity of this version is exponential, but it will be later reduced to quadratic in combination with the greedy Steiner Tree algorithm.

In the beginning, for each specified classifier f_i a corresponding node n_i is generated in the graph and connected with the root node by the directed edge (n_r, n_i). In the main loop, which iterates over $l = k$ down to 2, for each pair (f_i, f_j) of classifiers of length l the maximum common subclassifier f_s is determined. If it is valid (i.e., it is non-zero and contains at least one positive and one negative class), two cases are possible:

- a corresponding node to f_s already exists in the tree: f_i and f_j are included to the set of childs of f_s, that means, two directed edges e_{si} and e_{js} with weights $I(f_i \triangle f_s)$, $I(f_j \triangle f_s)$ respectively are created, where $I(.)$ denotes the total number of training instances for a given code configuration.
- There exists no corresponding node to the subclassifier f_s: f_s is integrated into the tree by creating a corresponding node and by linking it to the root node with edge e_{rs} of weight $I(f_s)$. In addition, the same steps as in the first case are applied.

Greedy Computing of Steiner Trees A Steiner Tree is, essentially, a minimum spanning tree of a graph, but it may contain additional nodes (which, in our case, correspond to unspecified classifiers). Minimizing the costs is equivalent to minimizing the total number of training examples that are needed to train all classifiers at the leaf of the tree from its root. As mentioned previously, we tackle this problem in a greedy way.

Let f_i be a specified classifier and E_i the set of incident incoming edges. We compute the minimum-weight edge and remove all other incoming edges. The outgoing node of this minimum edge is stored to repeat the process on this node afterwards, e.g. by adding it into a FIFO-queue. This is done until all classifiers and connected subclassifiers have been processed. Note, some subclassifiers are never processed, since all outgoing edges may have been removed. A pseudocode of this simple greedy approach is depicted in algorithm 2. In the following, we will refer to it as the *min-redundant training scheme*, and to the calculated approximate Steiner Tree as $\hat{G}$.

This greedy approach can be combined with the generation method of the training graph, such that the resulting Steiner Graph is identical and such that the overall complexity is reduced to polynomial time. Recall the first step of the generation method: all pairs of classifiers of length k are checked for common subclassifiers and eventually integrated into G. After generating $O(n^2)$ subclassifiers, for each classifier f_i (of

Algorithm 2 Greedy Steiner Tree Computation

```
Require: Training Graph $G = (V, E, w)$, binary classifiers $f_1, \ldots, f_n$
 1: let $Q$ be an empty FIFO-queue
 2: $\hat{V} = \varnothing, \hat{E} = \varnothing$
 3: for each $f_i$ do
 4:     $Q$.push($n_i$) , $\hat{V} = \hat{V} \cup \{n_i\}$
 5: end for
 6: while !$Q$.isEmpty() do
 7:     $n_i = Q$.pop()
 8:     $(n_x, n_i) = \text{argmin}_{(n_a, n_i) \in E}\, w((n_a, n_i))$
 9:     $\hat{E} = \hat{E} \cup (n_x, n_i)$ , $\hat{V} = \hat{V} \cup \{n_x\}$
10:     if $n_x \neq n_r$ then
11:         $Q$.push($n_x$)
12:     end if
13: end while
14: return $\tilde{G} = (\hat{V}, \hat{E}, w)$
```

length k) the minimal incoming edge[1] is marked. All unmarked edges and also the corresponding outgoing nodes, if they have no other child, are removed. In the next step of the iteration, $l = k - 1$, the number of nodes with length $l - 1$ are now at most n, since only maximally n new subclassifiers were included into the graph G. This means, for each level, $O(n^2)$ subclassifiers are generated, where the generation/checking of a subclassifier has cost of $\Theta(k)$, since we have to check k bits. So, in total, each level costs $O(n^2 \cdot k)$ operations. And, since we have k levels, the total complexity is $O(k^2 \cdot n^2)$. The implementation of the combined greedy method is straight-forward, so we omit a pseudocode and we will refer to it as GSTEINER.

3.3 Incremental Learning with Training Graph

Given a Steiner Tree of the training graph, learning with an incremental base learner is straight-forward. The specific training scheme is traversed in preorder depth-first-manner, i.e. at each node, the node is first evaluated and then its subtrees are traversed in left-to-right order. Starting from the root node, the first classifier f_1 is learned in batch mode. In the next step, if f_1 has a child, i.e. f_1 is subclassifier of another classifier f_2, f_1 is copied and incrementally learned with instances of $f_2 \bigtriangleup f_1$, yielding classifier f_2 and so on.

After the learning process, all temporary learned classifiers, which served as subclassifiers and are not specified in the ECOC matrix, are removed, and the prediction phase of the ECOC ensemble remains the same.

In this paper, we use *Hoeffding Trees* [6] as an example for an incremental learner. It is a very fast incremental decision tree learner with anytime properties. One of its main features is that its prediction is guaranteed to be asymptotically nearly identical to that of a corresponding batch learned tree. We used the implementation provided in the Massive Online Analysis Framework [2].

[1] The weights of the edges are identical to the corresponding ones in the fully generated training graph, since it only depends on the total number of training instances, computable by the code configuration of the subclassifier, and not on the actual partitioning.

3.4 SVM Learning with Training Graph

While incremental learners are obvious candidates for our approach to save training time, the problem actually does not demand full incrementality because we always add batches of examples corresponding to different classes to the training set. Thus, the incremental design of a training algorithm might retard the training compared to an algorithm that can naturally incorporate larger groups of additional instances. Therefore, we decided to study the applicability of this approach to a genuine batch learner, and selected the Java-implementation of LIBSVM [4]. The adaption of this base learner consists of two parts: First, the previous model (subclassifier) is used as a starting point for the successor model in the training graph, and second, the caching strategy is adapted to this scenario.

Reuse of Weights A binary SVM model consists of a weight vector $\boldsymbol{\alpha}$ containing the weights α_i for each training instance $(\boldsymbol{x}_i, y_i)$ and a real-valued threshold b. The latter is derived from $\boldsymbol{\alpha}$ and the instances without significant costs. The weights α are obtained as the solution of a quadratic optimization problem with a quadratic form $\alpha^T(\boldsymbol{y}^T K \boldsymbol{y})\alpha$ that incorporates the inputs through pairwise evaluations $K_{ij} = k(\boldsymbol{x}_i, \boldsymbol{x}_j)$ of the kernel function k. The first component to speed up the training is to use the weights α of the parent model as start values for optimizing the child weights $\bar{\alpha}$. That is, we set $\bar{\alpha}_i = \alpha_i$, if instance i belongs to the parent model and $\bar{\alpha}_i = 0$ otherwise.

The mutual influence of different instances on their respective weights is twofold. There is a local mutual influence due to the fact that an instance can stand in the shadow of another instance closer to the decision boundary. And there is a weaker, global mutual influence that also takes effect on more unrelated instances communicating through the error versus regularization trade-off in the objective.

If we add additional instances to the training set we might expect that there is only a modest alternation of the old weights, because many of the new instances will have little direct effect on the local influence among previous instances. On the other hand, if the new instances do interfere with some subsets of the previous instances, the global influence can strongly increase as well. In any case, we are more interested in the question whether the parent initialization of the weights does speed up the optimization step.

Cache Strategy It is well-known that caching of kernel evaluations provides significant speed-up for the learning with SVMs [10]. LIBSVM uses a *Least-Recently-Used* (LRU) Cache, which stores columns of the matrix K respective its signed variant $Q = \boldsymbol{y}^T K \boldsymbol{y}$. Since we use an ensemble of classifiers which potentially overlap in terms of their training instances and therefore also in their matrices K, it is beneficial to replace their local caches, which only keep information for each individual classifier, with an ensemble cache, which allows to transfer information from one classifier to the next.

Typically, each classifier receives a different subset of training instances $T_l \subset T$, specified by its code configuration. In order to transfer common kernel evaluations K_{ab} from classifier f_i to another classifier f_j, the cached columns have to be transformed, since they can contain evaluations of irrelevant instances. Each K_{ab} has to be removed, if instances a or b are not contained in the new training set and also the possible change in the ordering of instances has to be considered in the columns. The main difficulty is the implementation of an efficient mapping of locally used instance ids to the entire training set and its related transformation steps, otherwise, the expected speed-up of an ensemble caching strategy is undone.

Two ensemble cache strategies were evaluated, which are based on the local cache implementation of LIBSVM. The first one reuses *nearly all* reusable cached kernel evaluations from one classifier to another. For each classifier, two mapping tables $m_a(.)$ and $m_o(.)$ are maintained, where m_a associates each local instance number with its corresponding global instance number in order to have a unique addressing used in the transformation step. The table m_o is the mapping table from the previous learning phase. Before using the old cache for the learning of a new classifier, all cached entries are marked (as to be converted). During querying of the cache two cases can occur:

– **a cached column is queried:** If the entry is marked, the conversion procedure is applied. Using the previous and actual mapping table m_o, m_a, the column is transformed to contain only kernel evaluations for relevant instances, which can be done in $O(|m_o| \cdot \log |m_o|)$. Missing kernel evaluations are marked with a special symbol, which are computed afterwards. In addition, the mark is removed.
– **an uncached column is queried:** If the free size of the cache is sufficient, the column is computed and normally stored. Otherwise, beforehand, the least recently used entry is repeatedly removed until the cache has sufficient free space.

Since the columns are converted only on demand, unnecessary conversions are avoided and their corresponding entries are naturally replaced by new incoming kernel evaluations due to the LRU strategy. But, this tradeoff has the disadvantage that kernel evaluations that have been computed and cached at some point earlier may have to be computed again if they are requested later. The marked entries are carried maximally only over two iterations, otherwise it would be necessary for each additional iteration to carry another mapping table. We denote this ensemble caching method as *Short-Term Memory* (STM). One beneficial feature is the compatibility to any training scheme, in particular to the standard and the min-redundant training scheme.

The second ensemble caching method is particularly tailored to the use with a min-redundant training scheme. It differs from the previous one only in its transformation step. Recall that the learning phase traverses the subgraph in preorder depth-first manner. That means that during the learning procedure only the following two cases can occur: either the current classifier f_i is the child of a subclassifier f_j, or the current classifier is directly connected with the root node.

This information can be used for a more efficient caching scheme. For the first case, the set of training instances of f_i is superset of f_j, i.e. $T_j \subseteq T_i$. That means, $|T_j|$ rows and columns can be reused and also importantly without any costly transforming method. The columns and rows have to be simply trimmed to size $|T_j|$ for the reuse in the current classifier. Trimming is sometimes necessary, since they can contain further kernel evaluations from previously learned sibling nodes, i.e. nodes which share the same subclassifier f_j. So, the cache for the current classifier is prepared by removing Q_{ab} with $a > |T_j| \vee b > |T_j|$. In the second case, we know beforehand that no single kernel evaluation can be reused in the actual classifier. So, the cache is simply cleared. We denote this ensemble cache method as *Semi-Local* (SL) cache.

4 Experimental Evaluation

4.1 Experimental Setup

As we are primarily concerned with computational costs and not with predictive accuracy, we applied pre-processing based on all available instances instead of building a pre-processing model on the training data only. First, missing values were replaced by the average or majority value for numeric or ordinal attributes respectively. Second, all numeric values were normalized, such that the values lie in the unit interval.

Our experiment consisted of following parameters and parameter ranges:

- **6+2 multiclass classification datasets** from the UCI repository [7], where 6 relatively small datasets in terms of instances (up to ca. 4000) were used in conjunction with LIBSVM and two large-scale datasets, $pokerhand$ and $covtype$ consisting of $581,012$ and $1,025,010$ instances, were used with Hoeffding Trees. The number of classes lie in the range between 4 and 11.
- **3 code types**: exhaustive $k-$level codes, exhaustive cumulative $k-$level codes, random codes of up to length 500 with $k = 3, 4$ and $r_{zp} = 0.2, 0.4$
- **2 learn methods**: min-redundant and standard training scheme
- **2 base learners**: incremental learner Hoeffding Trees and batch learner LIBSVM (no parameter tuning, RBF-kernel) for which following parameters were evaluated:
  - **3 cache methods**: two ensemble cache methods, namely STM and SL, and the standard local cache of LIBSVM
  - **4 cache sizes**: 25%, 50%, 75%, 100% of the number of total kernel evaluations

All experiments with LIBSVM were conducted with 5-fold cross-validation and for Hoeffding Trees a training-test split of 66% to 33% was used. The parameters of the base learners were not tuned, because we were primarily interested in their computational complexity.[2]

4.2 Hoeffding Trees

Table 1 shows a comparison between the standard training scheme and the greedy computed min-redundant scheme with respect to the total amount of training instances. It shows that even with the suboptimal greedy procedure a significant amount of training instances can be saved. In this evaluation, the worst case can be observed for dataset $covtype$ with 3-level exhaustive codes, for which the ratio to the standard training scheme is 22%. In absolute numbers, this relates to processing 3.8 million training instances instead of 17.2 million. In summary, the improvements range from 78% to 98% or in other words, 4 to 45 times less training instances are processed.

Table 2 shows the corresponding total training time. It shows that the previous savings w.r.t. the number of training instances do not transfer directly to the training time. One reason is that the constant factor in the linear complexity of Hoeffding Trees regarding the number of training instances decreases for increasing number of training

[2] Tuning of the SVM parameters of the base learners can be relevant here because it may affect the effectiveness of reusing and caching of models. However, this would add additional complexity to the analysis of total cost and was therefore omitted to keep the analysis simple.

Table 1. Total number of processed training instances of standard and min-redundant training scheme. The italic values show the ratio of both. The datasets $pokerhand$ and $covtype$ consist of $581,012$ and $1,025,010$ instances respectively, from which 66% was used as training instances.

dataset	standard	min-redundant	standard	min-redundant
EXHAUSTIVE CUMULATIVE CODES				
	$l = 3$		$l = 4$	
$pokerhand$	79,151,319	9,429,611 (*0.119*)	476,937,435	10,479,451 (*0.022*)
$covtype$	19,556,868	3,807,748 (*0.195*)	73,242,388	5,354,720 (*0.073*)
EXHAUSTIVE CODES				
	$l = 3$		$l = 4$	
$pokerhand$	73,062,756	9,429,591 (*0.129*)	397,786,116	10,478,523 (*0.026*)
$covtype$	17,256,060	3,796,818 (*0.220*)	53,685,520	5,191,055 (*0.097*)
RANDOM CODES				
	$r_{zp} = 0.4$		$r_{zp} = 0.2$	
$pokerhand$	258,035,711	10,205,330 (*0.040*)	311,051,271	8,990,547 (*0.029*)
$covtype$	153,519,616	6,744,692 (*0.044*)	95,483,532	5,300,005 (*0.056*)

Table 2. Training time in seconds. This table shows training performances for the standard and the min-redundant learning scheme. The italic values shows the ratio of both.

dataset	standard	min-redundant	standard	min-redundant
EXHAUSTIVE CUMULATIVE CODES				
	$l = 3$		$l = 4$	
$pokerhand$	261.27	127.33 (*0.487*)	1530.06	542.57 (*0.355*)
$covtype$	118.70	40.89 (*0.344*)	463.09	93.71 (*0.202*)
EXHAUSTIVE CODES				
	$l = 3$		$l = 4$	
$pokerhand$	236.52	131.12 (*0.554*)	1337.00	522.18 (*0.391*)
$covtype$	101.50	34.65 (*0.341*)	330.97	83.58 (*0.253*)
RANDOM CODES				
	$r_{zp} = 0.4$		$r_{zp} = 0.2$	
$pokerhand$	896.41	356.43 (*0.398*)	1089.99	537.11 (*0.493*)
$covtype$	1106.48	157.61 (*0.142*)	695.84	107.12 (*0.154*)

instances. Furthermore, some overhead is incurred for copying the subclassifiers before each incremental learning step. In total, exploiting the redundancies yields a run-time reduction of about $44.6\% - 85.8\%$.

The running-time for GSTEINER (constructing the graph and greedily finding the Steiner tree, without evaluation of the classifiers) is depicted in Table 3. For the *systematic* code types, exhaustive and its cumulative version, the used time is in general negligible compared to the total training time. The only exception is for dataset $pokerhand$ with random codes and $r_{zp} = 0.2$: About 106 seconds were used and contributes therefore one-fifth to the total training time in this case.

4.3 LibSVM

Table 4 shows a comparison of training times between LIBSVM and its adaptions with weight reusing and ensemble caching strategies. M1 and M2 use the standard training scheme, where M1 is standard LIBSVM with local cache and M2 uses the ensemble caching strategy STM. M3 and M4 utilize a min-redundant training scheme with STM and SL respectively. The underlined values depict the best value for each dataset and code-type combination. The results confirm that the weight reuse and ensemble caching

Table 3. GSTEINER running time in seconds.

	EXH. CUMULATIVE		EXHAUSTIVE		RANDOM	
	$l = 3$	$l = 4$	$l = 3$	$l = 4$	$r_{zp} = 0.4$	$r_{zp} = 0.2$
$pokerhand$	0.82	4.63	4.56	3.57	22.09	105.97
$covtype$	0.24	3.01	0.14	0.17	0.67	0.52

Table 4. Training time in seconds using a cache size of 25%.

	optdigits	page-blocks	segment	solar-flare-c	vowel	yeast
			EXHAUSTIVE CUMULATIVE CODES			
			$l = 3$			
M1	92.28 ± 0.36	8.73 ± 0.19	6.56 ± 0.05	3.47 ± 0.07	5.80 ± 0.02	5.43 ± 0.03
M2	80.70 ± 0.37	8.32 ± 0.37	6.00 ± 0.03	4.30 ± 0.08	4.90 ± 0.02	5.62 ± 0.02
M3	76.93 ± 0.60	6.90 ± 0.18	6.94 ± 0.05	3.13 ± 0.16	6.28 ± 0.04	5.77 ± 0.03
M4	53.37 ± 0.40	2.93 ± 0.27	4.19 ± 0.05	1.70 ± 0.25	3.51 ± 0.01	2.98 ± 0.02
			$l = 4$			
M1	833.12 ± 14.98	24.66 ± 0.43	33.98 ± 0.21	18.61 ± 0.35	47.61 ± 0.08	40.42 ± 0.09
M2	666.02 ± 1.54	21.19 ± 0.80	28.69 ± 0.14	22.94 ± 0.52	36.72 ± 0.08	41.19 ± 0.11
M3	680.75 ± 8.23	18.30 ± 0.51	36.91 ± 0.39	15.08 ± 1.71	51.61 ± 0.15	41.79 ± 0.10
M4	410.44 ± 6.08	5.32 ± 0.53	17.18 ± 0.13	8.59 ± 1.27	25.26 ± 0.06	22.01 ± 0.10
			EXHAUSTIVE CODES			
			$l = 3$			
M1	87.42 ± 0.35	7.63 ± 0.39	6.02 ± 0.03	3.17 ± 0.05	5.51 ± 0.03	5.11 ± 0.02
M2	75.28 ± 0.29	6.76 ± 0.12	5.48 ± 0.03	3.95 ± 0.07	4.58 ± 0.03	5.28 ± 0.01
M3	75.61 ± 1.04	7.09 ± 0.27	6.91 ± 0.04	3.13 ± 0.14	6.25 ± 0.03	5.83 ± 0.05
M4	53.13 ± 0.39	2.90 ± 0.21	4.13 ± 0.03	1.71 ± 0.25	3.48 ± 0.02	3.00 ± 0.02
			$l = 4$			
M1	735.76 ± 9.63	15.31 ± 0.49	27.13 ± 0.31	15.14 ± 0.28	41.78 ± 0.09	34.99 ± 0.08
M2	570.69 ± 1.93	12.72 ± 0.45	22.76 ± 0.13	18.72 ± 0.42	31.92 ± 0.06	35.73 ± 0.06
M3	646.6 ± 11.98	16.39 ± 0.44	34.24 ± 0.36	14.69 ± 1.59	49.75 ± 0.10	41.09 ± 0.10
M4	397.79 ± 5.07	4.76 ± 0.46	15.88 ± 0.09	8.45 ± 1.17	24.55 ± 0.10	21.71 ± 0.06
			RANDOM CODES			
			$r_{zp} = 0.4$			
M1	1654.0 ± 22.6	25.7 ± 1.1	156.5 ± 1.7	34.7 ± 1.5	37.5 ± 0.6	46.9 ± 1.2
M2	1424.4 ± 32.8	24.3 ± 0.5	162.9 ± 0.8	46.1 ± 1.9	39.7 ± 0.7	52.1 ± 1.3
M3	1609.2 ± 44.3	22.6 ± 0.3	190.6 ± 3.8	39.9 ± 5.4	65.8 ± 2.3	79.1 ± 2.0
M4	1378.8 ± 34.4	5.7 ± 0.3	140.6 ± 3.0	25.9 ± 3.7	57.1 ± 2.5	64.5 ± 2.4
			$r_{zp} = 0.2$			
M1	2634.6 ± 59.5	10.2 ± 0.3	123.0 ± 0.9	48.2 ± 2.0	49.6 ± 0.4	67.2 ± 1.2
M2	2281.7 ± 29.6	8.6 ± 0.5	129.7 ± 1.4	63.2 ± 3.1	53.0 ± 0.4	74.1 ± 1.3
M3	3049.0 ± 48.3	12.7 ± 0.2	157.9 ± 1.4	57.6 ± 13.3	153.0 ± 2.0	157.5 ± 2.1
M4	2594.0 ± 64.8	3.6 ± 0.2	128.5 ± 2.4	39.1 ± 9.4	144.6 ± 1.7	144.0 ± 2.2

techniques can be used to exploit code redundancies for LIBSVM. For exhaustive codes and its cumulative variant, M4 dominates all other approaches and achieves an improvement of $31.4\% - 78.4\%$ of the training time. However, the results for random codes are not so clear.

For the datasets $vowel$ and $yeast$ both methods employing the min-redundant training schemes (M3 and M4) use significantly more time. This can be explained with the relative expensive cost for generating and solving the Steiner Tree in these cases, as depicted in Table 5 (89 and 52 sec for vowel and yeast). Contrary to the the results on $optdigits$, for these datasets the tree generation and solving has a big impact on the total training time. Nevertheless, this factor is decreasing for increasing number of instances, since the complexity of GSTEINER only depends on k and n. Besides, based on the results with various cache sizes, which we omit due to space restrictions (we refer to [12] for all results), the cache size has a greater impact on the training time for random

Table 5. GSTEINER running time for random codes in seconds.

	optdigits	page-blocks	segment	solar-flare-c	vowel	yeast
$r_{zp} = 0.4$	8.93	< 0.01	0.12	0.50	15.10	8.56
$r_{zp} = 0.2$	53.60	< 0.01	0.12	1.26	89.34	52.80

Table 6. Training time in seconds of random codes using a cache size of 75%.

	optdigits	page-blocks	segment	solar-flare-c	vowel	yeast
			RANDOM CODES, CACHE=75%			
			$r_{zp} = 0.4$			
M1	1603.4 ± 22.2	25.8 ± 0.5	153.7 ± 1.5	34.3 ± 1.5	36.0 ± 0.6	45.6 ± 1.1
M2	1317.4 ± 16.3	23.0 ± 0.4	136.9 ± 1.0	45.1 ± 1.9	35.9 ± 0.6	51.2 ± 1.3
M3	1364.6 ± 53.6	22.4 ± 0.2	148.7 ± 1.3	35.8 ± 4.5	60.9 ± 2.0	82.6 ± 2.2
M4	1162.6 ± 27.6	5.5 ± 0.3	70.3 ± 0.4	7.9 ± 0.8	42.8 ± 2.0	27.4 ± 1.2
			$r_{zp} = 0.2$			
M1	2507.0 ± 33.8	10.3 ± 0.3	119.8 ± 1.2	47.6 ± 2.0	47.6 ± 0.4	65.3 ± 1.2
M2	1826.2 ± 21.3	8.5 ± 0.6	98.7 ± 0.6	61.3 ± 3.1	44.3 ± 0.4	70.6 ± 1.2
M3	2093.7 ± 38.6	12.4 ± 0.2	116.9 ± 0.8	51.0 ± 11.0	139.9 ± 1.8	163.7 ± 2.6
M4	1632.5 ± 40.2	3.9 ± 0.1	56.8 ± 0.3	10.0 ± 1.6	118.6 ± 1.6	87.7 ± 2.2

codes than for the systematic ones. Table 6 shows as an example the performance for random codes with a cache size of 75%. Notice the reduction of the training time for the different methods in comparison to Table 4, where a cache size of 25% was used. M4 achieves the best efficiency increase and by subtracting the time for generating and solving the tree, M4 dominates again all other methods.

Table 7 shows the number of optimization iterations of LIBSVM, which can be seen as an indicator of training complexity. The ratio values are averaged over all datasets and show that the reuse of weights in the pseudo-incremental learning steps lead to a reduction of optimization iterations. Once again, the effect on the ensemble caching strategy can be seen in Table 8, showing a selection of the results, here for cache sizes 25% and 75%. The first column of each block describes the number of kernel evaluation calls. The consistent reduction for min-redundant schemes M3 and M4 is accredited to the weight-reusing strategy. Except for random codes with $r_{zp} = 0.2$ and cache size=25% all methods using an ensemble cache strategy (M2, M3 and M4) outperform the baseline of LIBSVM with a local cache. Among these three methods, M3 and M4 both outperform M2 in absolute terms, but not relative to the number of calls. For the special case (random codes, $r_{zp} = 0.2$, M3, M4), one can again see the increased gain of a bigger cache size for the min-redundant training schemes.

Even though all ensemble caching strategies almost always outperform the baseline in terms of hit-miss measures, the corresponding time complexities of Table 4 show that only M4, which uses a min-redundant training scheme and the SL caching strategy, is reliably reducing the total training time. The rather costly transformation cost of STM is the cause for the poor performance of M2 and M3.

5 Related Work

In [3], an efficient algorithm for cross-validation with decision trees is proposed, which also exploits training set overlaps, but focuses on a different effect, namely that in this case the generated models tend to be similar, such that often identical test nodes are

Table 7. Comparison of LIBSVM Optimization iterations. The values show the ratio of optimization iterations of a min-redundant training scheme with weight reusing to standard learning.

EXH. CUMULATIVE		EXHAUSTIVE		RANDOM	
$l = 3$	$l = 4$	$l = 3$	$l = 4$	$r_{zp} = 0.4$	$r_{zp} = 0.2$
0.673	0.576	0.768	0.745	0.701	0.773

Table 8. Cache efficiency and min-redundant training scheme impact: averaged mean ratio values of kernel evaluation calls (first column) and actual computed kernel evaluations (second column) to the baseline: standard LIBSVM (M1). The values of M1 are set to 1 and the following values describe the ratio of corresponding values of M2, M3 and M4 to M1.

	EXH.CUMULATIVE				EXHAUSTIVE				RANDOM			
	$l = 3$		$l = 4$		$l = 3$		$l = 4$		$r_{zp} = 0.4$		$r_{zp} = 0.2$	
	CACHE = 25%											
M2	1.00	0.68	1.00	0.61	1.00	0.66	1.00	0.60	1.00	0.83	1.00	0.84
M3	0.78	0.56	0.71	0.52	0.87	0.63	0.88	0.67	0.84	0.83	0.95	1.01
M4	0.78	0.56	0.71	0.51	0.87	0.63	0.88	0.65	0.84	0.83	0.95	1.00
	CACHE = 75%											
M2	1.00	0.59	1.00	0.48	1.00	0.56	1.00	0.44	1.00	0.64	1.00	0.56
M3	0.78	0.43	0.71	0.34	0.87	0.47	0.88	0.42	0.84	0.41	0.95	0.47
M4	0.78	0.44	0.71	0.32	0.87	0.48	0.88	0.41	0.84	0.42	0.95	0.49

generated in the decision tree during the learning process. This approach is not applicable here, since during the incremental learning steps, the inclusion of new classes may lead to significant model changes. Here, a genuine incremental learner or in the case of LIBSVM different approaches are necessary. However, the main idea, to reduce redundant computations is followed also here.

Pimenta et al. [13] consider the task of optimizing the size of the coding matrix so that it balances effectivity and efficiency. Our approach is meant to optimize efficiency for a given coding matrix. Thus, it can also be combined with their approach if the resulting *balanced* coding matrix is code-redundant.

6 Conclusion

We studied the possibility of reducing the training complexity of ECOC ensembles with highly redundant codes such as exhaustive cumulative, exhaustive and random codes. We proposed an algorithm for generating a so-called training graph, in which edges are labeled with training cost and nodes represent (sub-)classifiers. By finding an approximate Steiner Tree of this graph in a greedy manner, the training complexity can be reduced without changing the prediction quality. An initial evaluation with Hoeffding Trees, as an example for an incremental learner, yielded time savings in the range of 44.6% to 85.8%. Subsequently, we also demonstrated how SVMs can be adapted for this scenario by reusing weights and by employing an ensemble caching strategy. With this approach, the time savings for LIBSVM ranged from 31.4% to 78.4%. In general, we can expect higher gains for incremental base learners whose complexity grows more steeply with the number of training instances. The presented approach is useful for all considered high-redundant code types, and also for random codes, for which the impact of the GSTEINER algorithm decreases with increasing training instances. In addition, the generation of a min-redundant training scheme could be seen as a pre-processing step, such that it is not counted or only counted once for the total training time of an ECOC ensemble, because it is reusable and independent of the base learner.

However, this approach has its limitations. GSTEINER can be a bottleneck for problems with a high class count, since its complexity is $O(n^2 \cdot k^2)$ and the length n for common code types such as exhaustive codes grow exponentially in the number of classes k. And, this work considers only highly redundant code types, which are not unproblematic. First, usually in conjunction with ECOC ensembles, one prefers diverse classifiers, which are contrasting the redundant codes in our sense. The more shared code configurations exist in an ensemble, the less independent are its classifiers. Secondly, these codes are not as commonly used as the low-redundant decompositions schemes one-against-all and one-against-one.

Acknowledgments This work was supported by the *German Science Foundation (DFG)*.

References

1. Allwein, E.L., Schapire, R.E., Singer, Y.: Reducing multiclass to binary: A unifying approach for margin classifiers. J. Mach. Learn. Res. (JMLR) **1** (2000) 113–141
2. Bifet, A., Holmes, G., Kirkby, R., Pfahringer, B.: MOA: Massive Online Analysis http://sourceforge.net/projects/moa-datastream/. J. Mach. Learn. Res. (JMLR) (2010)
3. Blockeel, H., Struyf, J.: Efficient algorithms for decision tree cross-validation. J. Mach. Learn. Res. (JMLR) **3** (2003) 621–650
4. Chang, C.-C., Lin, C.-J.: LIBSVM: a library for support vector machines. (2001) Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.
5. Dietterich, T.G., Bakiri, G.: Solving multiclass learning problems via error-correcting output codes. J. Artif. Intell. Res. (JAIR) **2** (1995) 263–286
6. Domingos, P., Hulten, G.: Mining high-speed data streams. In: KDD, Boston, MA, USA, ACM (2000) 71–80
7. Frank, A., Asuncion, A.: UCI machine learning repository (2010)
8. Friedman, J.H.: Another approach to polychotomous classification. Technical report, Department of Statistics, Stanford University, Stanford, CA (1996)
9. Fürnkranz, J.: Round robin classification. J. Mach. Learn. Res. (JMLR) **2** (2002) 721–747
10. Joachims, T.: Making large-scale SVM learning practical. In Schölkopf, B., Burges, C., Smola, A., eds.: Advances in Kernel Methods - Support Vector Learning. MIT Press, Cambridge, MA (1999) 169–184
11. Park, S.-H., Fürnkranz, J.: Efficient decoding of ternary error-correcting output codes for multiclass classification. In Buntine, W.L., Grobelnik, M., Mladenić, D., Shawe-Taylor, J., eds.: ECML/PKDD-09. Volume Part II., Bled, Slovenia, Springer-Verlag (2009) 189–204
12. Park, S.-H., Weizsäcker, L., Fürnkranz, J.: Exploiting code-redundancies in ECOC for reducing its training complexity using incremental and SVM learners. Technical Report TUD-KE-2010-06, TU Darmstadt (July 2010)
13. Pimenta, E., Gama, J., Carvalho, A.: Pursuing the best ecoc dimension for multiclass problems. In Wilson, D., Sutcliffe, G., eds.: FLAIRS Conference, AAAI Press (2007) 622–627
14. Wong, R.: A dual ascent approach for steiner tree problems on a directed graph. Mathematical Programming **28**(3) (oct 1984) 271–287