Fast Evolutionary Maximum Margin Clustering

Fabian Gieseke FABIAN.GIESEKE@CS.TU-DORTMUND.DE
Faculty of Computer Science, TU Dortmund, Germany

Tapio Pahikkala TAPIO.PAHIKKALA@UTU.FI
Turku Centre for Computer Science, Department of Information Technology, University of Turku, Finland

Oliver Kramer OLIVER.KRAMER@CS.TU-DORTMUND.DE
Faculty of Computer Science, TU Dortmund, Germany

Abstract

The maximum margin clustering approach is a recently proposed extension of the concept of support vector machines to the clustering problem. Briefly stated, it aims at finding an optimal partition of the data into two classes such that the margin induced by a subsequent application of a support vector machine is maximal. We propose a method based on stochastic search to address this hard optimization problem. While a direct implementation would be infeasible for large data sets, we present an efficient computational shortcut for assessing the "quality" of intermediate solutions. Experimental results show that our approach outperforms existing methods in terms of clustering accuracy.

1. Introduction

The task of clustering a given set of objects into groups of "similar" items is one of the most investigated problems in both data mining and machine learning and can be applied to many real-world situations such as computer vision, information retrieval, marketing, and many others (Jain & Dubes, 1988).

Recently, a new clustering technique called *maximum margin clustering* (MMC) has been proposed by Xu *et al.* (2005). This technique can be seen as an extension of support vector machines (SVM) (Vapnik, 1998) to unsupervised learning scenarios: Having no class labels at hand, the aim is to find a partition of the objects into two classes such that the margin computed by a subsequent application of a support vector machine is maximal. Experimental results show that this technique often outperforms common clustering methods with respect to the accuracy. However, applying the approach requires solving a non-convex integer problem, which is, due to its combinatorial nature, a difficult task.

Aiming at practical solutions, Zhang *et al.* (2007) proposed a method which is based on iteratively applying a SVM to improve an initial "guess" obtained by a k-means (Hartigan & Wong, 1979) preprocessing step. Our method is inspired by this approach, i.e. starting with an initial set of (random) candidate solutions, we iteratively update the quality of these candidates by means of a stochastic search heuristic. Our key contribution is a computational shortcut for assessing the quality of a candidate solution. More precisely, we depict how to efficiently update some auxiliary information and, based on this information, how to compute the quality of a candidate in linear time. Compared to standard methods for evaluating the quality, this constitutes a runtime reduction by a quadratic factor and will make our approach capable of testing a *massive* amount of candidate solutions. Our experimental results show that our approach yields a better clustering accuracy than conventional techniques in most cases.

Notations. We use $[n]$ to denote the set $\{1, \ldots, n\}$. Further, the set of all $n \times m$ matrices with real coefficients is denoted by $\mathbb{R}^{n \times m}$. Given a matrix $M \in \mathbb{R}^{n \times m}$, we denote the element in the i-th row and j-th column by $[M]_{i,j}$. For two sets $R = \{i_1, \ldots, i_r\} \subseteq [n]$ and $S = \{k_1, \ldots, k_s\} \subseteq [m]$ of indices, we use M_{RS} to denote the matrix that contains only the rows and columns of M that are indexed by R and S, respectively. Moreover, we set $M_{R,[m]} = M_R$. At last, we use y_i to denote the i-th coordinate of a vector $\mathbf{y} \in \mathbb{R}^n$.

Appearing in *Proceedings of the 26th International Conference on Machine Learning*, Montreal, Canada, 2009.

2. Maximum Margin Clustering

2.1. Standard Formulation

Given a training set $T = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$ with patterns $\mathbf{x}_i$ belonging to a set X and class labels $y_i \in Y = \{-1, +1\}$, the aim of the SVM learning process is to find a hyperplane $f(\mathbf{x}) = \langle \mathbf{w}, \Phi(\mathbf{x}) \rangle + b$ in a *feature space* $\mathcal{H}_0$ induced by a *feature mapping* $\Phi : X \to \mathcal{H}_0$. Both the feature map and the feature space stem from a *kernel function* $k : X \times X \to \mathbb{R}$ with $k(\mathbf{x}_i, \mathbf{x}_j) = \langle \Phi(\mathbf{x}_i), \Phi(\mathbf{x}_j) \rangle$. This leads to the optimization problem

$$\begin{aligned} \underset{\mathbf{w} \in \mathcal{H}_0, \xi \in \mathbb{R}^n, b \in \mathbb{R}}{\text{minimize}} \; & \frac{1}{2}||\mathbf{w}||^2 + C \sum_{i=1}^{n} \xi_i \\ \text{s.t. } & y_i(\langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle + b) \geq 1 - \xi_i, \; \xi_i \geq 0, \end{aligned} \tag{1}$$

where the ξ_i values are called *slack variables* and where $C > 0$ is a manually chosen constant.

The maximum margin clustering approach aims at finding a partition of the patterns such that the margin yielded by a subsequent application of a SVM is maximal. Originally, this task was formulated in terms of the following optimization problem (Xu et al., 2005):

$$\begin{aligned} \underset{\mathbf{y} \in \{-1,+1\}^n, \mathbf{w}, \xi, b}{\text{minimize}} \; & \frac{1}{2}||\mathbf{w}||^2 + C \sum_{i=1}^{n} \xi_i \\ \text{s.t. } & y_i(\langle \mathbf{w}, \Phi(\mathbf{x}_i) \rangle + b) \geq 1 - \xi_i, \; \xi_i \geq 0, \\ \text{and } & -l \leq \sum_{i=1}^{n} y_i \leq l, \end{aligned} \tag{2}$$

where $C > 0$ and $l \geq 0$ are manually chosen constants.

2.2. Least-Squares Variant

The concept of support vector machines can also be considered to be a special case of regularization problems of the form

$$\inf_{f \in \mathcal{H}} \; \frac{1}{n} \sum_{i=1}^{n} L\big(y_i, f(\mathbf{x}_i)\big) + \lambda ||f||_{\mathcal{H}}^2, \tag{3}$$

where $\lambda > 0$ is a fixed real number, $L : Y \times \mathbb{R} \to [0, \infty)$ is a *loss function* measuring the "quality" of the prediction function f on the training set, and $||f||_{\mathcal{H}}^2$ is the squared norm in a *reproducing kernel Hilbert space* (RKHS) $\mathcal{H} \subseteq \mathbb{R}^X = \{f : X \to \mathbb{R}\}$ induced by a kernel function. Using the so-called *hinge loss* $L_{hinge}(y, t) = \max\{0, 1 - yt\}$ with $y \in \{-1, +1\}$ leads to the support vector machine approach depicted above (Schölkopf et al., 2001; Steinwart & Christmann, 2008).[1]

[1] In the latter formulation, the offset b is omitted. From the theoretical as well as from the practical point of view, the offset term yields no known advantages for more complex kernel functions like the Gaussian RBF kernel (Rifkin, 2002; Steinwart & Christmann, 2008). Furthermore, when using a linear kernel, a regularized bias effect can be obtained by adding a dimension of 1's to the data.

Central for our approach is the *square-loss* $L_{LS}(y, t) = (y - t)^2$ leading to

$$\inf_{f \in \mathcal{H}} \; \frac{1}{n} \sum_{i=1}^{n} \big(y_i - f(\mathbf{x}_i)\big)^2 + \lambda ||f||_{\mathcal{H}}^2 \tag{4}$$

(Rifkin et al., 2003; Suykens & Vandewalle, 1999). By the representer theorem (Schölkopf et al., 2001), any minimizer $f^* \in \mathcal{H}$ of (4) has the form

$$f^*(\cdot) = \sum_{i=1}^{n} c_i k(\mathbf{x}_i, \cdot) \tag{5}$$

with appropriate coefficients $\mathbf{c} = (c_1, \ldots, c_n)^t \in \mathbb{R}^n$. Hence, by using $||f^*||_{\mathcal{H}}^2 = \mathbf{c}^t K \mathbf{c}$, where $K \in \mathbb{R}^{n \times n}$ is the (symmetric) kernel matrix with entries $[K]_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j)$, we can rewrite optimization problem (4) as

$$\underset{\mathbf{c} \in \mathbb{R}^n}{\text{minimize}} \, J(\mathbf{y}, \mathbf{c}) = \frac{1}{n}(\mathbf{y} - K\mathbf{c})^t(\mathbf{y} - K\mathbf{c}) + \lambda \mathbf{c}^t K \mathbf{c}. \tag{6}$$

Our approach is based on the replacement of the hinge loss by its least-squares variant, i.e. instead of solving problem (2) we aim at finding a solution for

$$\begin{aligned} \underset{\mathbf{y} \in \{-1,+1\}^n, \mathbf{c} \in \mathbb{R}^n}{\text{minimize}} \; & \frac{1}{n}(\mathbf{y} - K\mathbf{c})^t(\mathbf{y} - K\mathbf{c}) + \lambda \mathbf{c}^t K \mathbf{c} \\ \text{s.t.} \quad & -l \leq \sum_{i=1}^{n} y_i \leq l. \end{aligned} \tag{7}$$

2.3. Related Work

The first attempts at coping with problem (2) consisted in relaxing the definition to form semidefinite programming problems (Xu et al., 2005; Valizadegan & Jin, 2007). However, these approaches resort to solving SDP problems, which is computationally expensive. The iterative method was the first approach capable of dealing with large data sets (Zhang et al., 2007). One of their key insights consisted in the replacement of the hinge loss by several other loss functions including the square-loss. The cutting plane algorithm is one of the most recent techniques (Zhao et al., 2008a). It is based on constructing a sequence of successively tighter relaxations of (2) and each of the intermediate tasks is solved using the so-called constrained concave-convex procedure. In addition to these methods, extensions to multiclass scenarios have been proposed in the literature (Xu & Schuurmans, 2005; Zhao et al., 2008b).

The MMC technique is related to the concept of *Transductive Support Vector Machines (TSVM)* (Vapnik, 1998) and the corresponding optimization problem is approached by several heuristics including genetic algorithms (Silva et al., 2005; Adankon & Cheriet, 2007). Another related technique is given by Bach and Harchaoui (2008). Their method also uses the square-loss instead of the hinge loss. Based upon this replacement, they show how to solve the resulting problem by approximating a discrete set of equivalence matrices by a convex set (Bach & Harchaoui, 2008).

3. Fast Evolutionary Approach

In this section, we outline our evolutionary approach for optimization problem (7). Evolutionary algorithms are inspired by biological evolution (Beyer & Schwefel, 2002): They start with an initial set of candidate solutions, called *population*. For each candidate, also named *individual*, the *fitness* measuring the quality is computed. After the initialization phase the iteration over all *generations* is started. For each generation, some individuals of the current population are *mutated* and the fitness values for the mutated individuals are computed. At the end of each generation, the population is updated by selecting the best performing candidates. The key insight of our approach is the way we obtain the fitness values. More precisely, we show how to compute the fitness values for a mutated individual based on some auxiliary information given for the parental individual. Further, we describe how to efficiently update this auxiliary information such that it is available for further generations. This computational shortcut greatly reduces the runtime of our approach.

3.1. Evolutionary Optimization Approach

The framework of our evolutionary approach is given by Algorithm 1. The starting point is the population $\mathcal{P}_0 = \{\mathbf{y}_1, \ldots, \mathbf{y}_\mu\} \subseteq \{-1, +1\}^n$ consisting of μ randomly generated individuals. Each of these individuals (along with a corresponding vector $\mathbf{c} \in \mathbb{R}^n$) constitutes a possible solution for optimization problem (7). Throughout our algorithm, we ensure that only *valid* individuals are created, i.e. individuals $\mathbf{y}$ fulfilling the balance constraint $-l \leq \sum_{i=1}^{n} y_i \leq l$. In Step 2, the fitness $F(\mathbf{y})$ is computed for each of the initial individuals, where

$$F(\mathbf{y}) = \underset{\mathbf{c} \in \mathbb{R}^n}{\text{minimize}}\, J(\mathbf{y}, \mathbf{c}). \tag{8}$$

Note that, in our case, individuals with a lower fitness represent better solutions. The iteration over all generations $1, \ldots, \tau$ is started in Step 4. For each generation t, we randomly select ν parental individuals to produce mutated individuals. Each of these mutated individuals is created by flipping $s = \max\{1, n/(t+1)\}$ coordinates of the parental individual.[2] After the computation of the fitness values for the mutated individuals, all resulting individuals are sorted upwards by their fitness values yielding a sorted sequence $\mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_{\mu+\nu}}$. Finally, the population $\mathcal{P}_t$ is updated by selecting the μ individuals with the best fitness values. When generation τ is reached, the best individual along with its corresponding vector $\hat{\mathbf{c}}$ is returned.

Algorithm 1 Evolutionary Optimizer

Require: Training set $T = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\} \subset X$, constants $C > 0$, $l \geq 0$, and μ, ν, $\tau \in \mathbb{N}$.
Ensure: $(\hat{\mathbf{y}}, \hat{\mathbf{c}}) \in \{-1, +1\}^n \times \mathbb{R}^n$
1: Initialize $\mathcal{P}_0 = \{\mathbf{y}_1, \ldots, \mathbf{y}_\mu\} \subseteq \{-1, +1\}^n$
2: Compute the fitness $F(\mathbf{y}_j)$ for each $\mathbf{y}_j \in \mathcal{P}_0$
3: $t = 0$
4: **while** $t \leq \tau$ **do**
5: **for** $i = 1$ **to** ν **do**
6: Randomly select parent $\mathbf{y} \in \mathcal{P}_t$
7: Generate valid mutated individual $\mathbf{y}_{\mu+i}$
8: Compute fitness $F(\mathbf{y}_{\mu+i})$
9: **end for**
10: Compute sorted sequence $\mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_{\mu+\nu}}$
11: $\mathcal{P}_{t+1} = \{\mathbf{y}_{i_1}, \ldots, \mathbf{y}_{i_\mu}\}$
12: $t = t + 1$
13: **end while**
14: Compute solution $\hat{\mathbf{c}}$ for $\text{minimize}_{\mathbf{c} \in \mathbb{R}^n} J(\mathbf{y}_{i_1}, \mathbf{c})$
15: **return** $(\mathbf{y}_{i_1}, \hat{\mathbf{c}})$

3.2. Fast Recurrent Fitness Computations

For fixed $\mathbf{y}$, the function $J(\mathbf{y}, \cdot)$ is convex and differentiable. Thus, the computation of the fitness $F(\mathbf{y}_{\mu+i})$ in Step 8 can be performed by solving $\frac{d}{d\mathbf{c}} J(\mathbf{y}, \mathbf{c}) = \mathbf{0}$ with respect to $\mathbf{c}$ which leads to

$$\mathbf{c}^* = G\mathbf{y}, \tag{9}$$

with $G = (K + \lambda n I)^{-1}$, where $I \in \mathbb{R}^{n \times n}$ denotes the identity matrix (Rifkin, 2002). This requires inverting a $n \times n$ matrix which takes $\mathcal{O}(n^3)$ time. Assuming that G is stored in memory, a fitness value $F(\bar{\mathbf{y}})$ for a new individual $\bar{\mathbf{y}}$ can be obtained in $\mathcal{O}(n^2)$ time. Unfortunately, the quadratic runtime would still render our approach infeasible.

Our key contribution is the following computational shortcut for obtaining the fitness values: Fix a parental individual $\mathbf{y} \in \mathcal{P}_t$ as selected in Step 6 and observe that a mutated individual $\bar{\mathbf{y}} = \mathbf{y}_{\mu+i}$ differs from $\mathbf{y}$ in

[2] If a flip violates the balance constraint, a coordinate with a reverse label is selected and both labels are swapped.

only a *small* number s of coordinates.[3] Further, by substituting (9) into (6), the value $F(\bar{\mathbf{y}})$ becomes

$$F(\bar{\mathbf{y}}) = \frac{1}{n}(\bar{\mathbf{y}} - KG\bar{\mathbf{y}})^t(\bar{\mathbf{y}} - KG\bar{\mathbf{y}}) + \lambda \bar{\mathbf{y}}^t GKG\bar{\mathbf{y}}. \quad (10)$$

Now, let $K = V\Lambda V^t$ be the eigendecomposition of the kernel matrix, where $V \in \mathbb{R}^{n\times n}$ contains the eigenvectors of K and where the diagonal matrix Λ contains the corresponding eigenvalues. Using the latter decomposition of the kernel matrix, we can write G as $G = V\widetilde{\Lambda}V^t$, where $\widetilde{\Lambda} = (\Lambda + \lambda nI)^{-1}$. Moreover, using $G = G^t$, the fitness value can then be reformulated as

$$\begin{aligned} F(\bar{\mathbf{y}}) &= \frac{1}{n}(\bar{\mathbf{y}} - KG\bar{\mathbf{y}})^t(\bar{\mathbf{y}} - KG\bar{\mathbf{y}}) + \lambda \bar{\mathbf{y}}^t GKG\bar{\mathbf{y}} \\ &= \bar{\mathbf{y}}^t \left(\frac{1}{n}I - \frac{2}{n}KG + \frac{1}{n}GKKG + \lambda GKG\right)\bar{\mathbf{y}} \\ &= 1 + \bar{\mathbf{y}}^t V \underbrace{\left(\frac{1}{n}\Lambda^2\widetilde{\Lambda}^2 - \frac{2}{n}\Lambda\widetilde{\Lambda} + \lambda\widetilde{\Lambda}\Lambda\widetilde{\Lambda}\right)}_{=:D} V^t\bar{\mathbf{y}}. \end{aligned}$$

Furthermore, let $S = \{k_1, \ldots, k_s\} \subseteq [n]$ denote the set of indices such that $\bar{\mathbf{y}}$ differs from $\mathbf{y}$ for all coordinates $k \in S$ and agrees for all coordinates $k \notin S$. Then,

$$\bar{\mathbf{y}}^t V = \mathbf{y}^t V + (\bar{\mathbf{y}}_S - \mathbf{y}_S)^t V_S, \quad (11)$$

can be computed in $\mathcal{O}(sn)$ time if $\mathbf{y}^t V \in \mathbb{R}^n$ is precomputed. As the remaining multiplications can each be performed in $\mathcal{O}(n)$ time, it follows that the overall runtime for computing $F(\bar{\mathbf{y}})$ is $\mathcal{O}(sn)$, assuming that $\mathbf{y}^t V$ and the diagonal matrix $D \in \mathbb{R}^{n\times n}$ are available.

Theorem 1 *Assuming that $\mathbf{y}^t V \in \mathbb{R}^n$ and $D \in \mathbb{R}^{n\times n}$ are precomputed and stored in memory, one can compute the fitness $F(\bar{\mathbf{y}})$ of a mutated individual with $\frac{1}{2}\sum_{j=1}^n |y_j - \bar{y}_j| = s$ in $\mathcal{O}(sn)$ time.*

The latter result permits an efficient implementation of Algorithm 1, which is based on iteratively updating the auxiliary information $\mathbf{y}^t V$ along with the shortcut for computing the fitness: The diagonal matrix D and the auxiliary information $\mathbf{y}_1^t V, \ldots, \mathbf{y}_\mu^t V$ for the initial population $\mathcal{P}_0$ can be obtained in $\mathcal{O}(n^3 + \mu n^2)$ time. Given this auxiliary information, the fitness value for each $\mathbf{y} \in \mathcal{P}_0$ can be computed in $\mathcal{O}(n)$ time. For each generation, the auxiliary information can be updated in $\mathcal{O}(\nu sn)$ time. Moreover, due to Theorem 1, a fitness computation in Step 8 can be performed in $\mathcal{O}(sn)$ time. Hence, taking into account that Step 14 takes $\mathcal{O}(n^3)$ time, the overall runtime of Algorithm 1 is $\mathcal{O}(n^3 + \mu n^2 + \tau\nu sn)$. Furthermore, the algorithm obviously needs $\mathcal{O}(n^2 + \mu n) = \mathcal{O}(n^2)$ space to store all matrices, the auxiliary information, and the individuals.

To sum up, for reasonable values for the parameters τ, μ, and ν, the *overall runtime* of our algorithm is essentially the same as training a *single* classifier via (9).

3.3. Kernel Matrix Approximation

The computational shortcut described above already offers a promising runtime reduction compared to the direct $\mathcal{O}(n^2)$ time approach for computing the fitness values. However, when dealing with large data sets, the cubic runtime for obtaining the diagonal matrix D as well as the quadratic space consumption for storing the kernel matrix form bottlenecks. In this section, we propose a way to shorten both drawbacks.

One way to deal with the above bottlenecks consists in approximating the kernel matrix: Fix an individual $\mathbf{y}$ and consider optimization problem (6). Here, the first term corresponds to the loss evaluated over all training patterns and the second term is the regularizer penalizing "complex" solutions. Now assume that only a subset of the coefficients $c_1, \ldots, c_n$ in (5) is allowed to be nonzero. More formally, let $R = \{i_1, \ldots, i_r\} \subseteq [n]$ be a subset of indices such that only $c_{i_1}, \ldots, c_{i_r}$ are nonzero in (5). Then, we search for minimizers $\hat{f}(\cdot) = \sum_{j=1}^r c_{i_j} k(\mathbf{x}_{i_j}, \cdot) \in \mathcal{H}$ solving

$$\begin{aligned} \hat{F}(\mathbf{y}) &= \underset{\hat{\mathbf{c}}\in\mathbb{R}^r}{\text{minimize}} \frac{1}{n}\big(\mathbf{y} - (K_R)^t\hat{\mathbf{c}}\big)^t\big(\mathbf{y} - (K_R)^t\hat{\mathbf{c}}\big) \\ &\quad + \lambda\hat{\mathbf{c}}^t K_{RR}\hat{\mathbf{c}}. \end{aligned} \quad (12)$$

Note that, by applying this approximation scheme, the loss is still evaluated over all training patterns. The benefit of this approach consists in a runtime and space reduction, i.e. the solution for (12) can be obtained in $\mathcal{O}(nr^2)$ time and $\mathcal{O}(nr)$ space (Rifkin, 2002) via

$$\hat{\mathbf{c}}^* = (K_R(K_R)^t + \lambda n K_{RR})^{-1} K_R\mathbf{y}. \quad (13)$$

In the remainder of this section, we show that the above approximation scheme can be integrated into our approach for obtaining the fitness values: The optimal prediction vector $\hat{\mathbf{f}}^* = (K_R)^t\hat{\mathbf{c}}^* \in \mathbb{R}^n$ for a mutated individual $\bar{\mathbf{y}}$ can be written as

$$\hat{\mathbf{f}}^* = (K_R)^t \frac{1}{\lambda}(K_{RR})^{-1} K_R\mathbf{u}^* = \frac{1}{\lambda}\widehat{K}\mathbf{u}^* = \widehat{K}\widehat{G}\bar{\mathbf{y}}$$

using

$$\widehat{K} = (K_R)^t(K_{RR})^{-1}K_R, \;\; \widehat{G} = \big(\widehat{K} + \lambda nI\big)^{-1},$$

$$\mathbf{u}^* = \lambda\big(\widehat{K} + \lambda nI\big)^{-1}\bar{\mathbf{y}}, \;\; \hat{\mathbf{c}}^* = \frac{1}{\lambda}(K_{RR})^{-1}K_R\mathbf{u}^*,$$

[3] The number s decreases rapidly. Also, the stochastic search works well with $s = 1$ for all generations; for ease of notation, we therefore assume s to be a small constant for the further analysis.

refer to the thesis of Rifkin (2002) for the derivation of the last equation.[4] Thus, using

$$(\hat{\mathbf{c}}^*)^t K_{RR}(\hat{\mathbf{c}}^*) = \left(\frac{\mathbf{u}^*}{\lambda}\right)^t \widehat{K} \left(\frac{\mathbf{u}^*}{\lambda}\right) = \bar{\mathbf{y}}^t \widehat{G}\widehat{K}\widehat{G}\bar{\mathbf{y}}, \quad (14)$$

we can rewrite $\hat{F}(\bar{\mathbf{y}})$ for the mutated individual $\bar{\mathbf{y}}$ as

$$\hat{F}(\bar{\mathbf{y}}) \quad = \quad \frac{1}{n}(\bar{\mathbf{y}} - \widehat{K}\widehat{G}\bar{\mathbf{y}})^t(\bar{\mathbf{y}} - \widehat{K}\widehat{G}\bar{\mathbf{y}}) + \lambda \bar{\mathbf{y}}^t \widehat{G}\widehat{K}\widehat{G}\bar{\mathbf{y}}.$$

The latter representation shows that applying the approximation approach corresponds to exchanging the matrices K and G by $\widehat{K}$ and $\widehat{G}$ in (10). The remaining parts of the computational shortcut leading to Theorem 1 can essentially be carried over directly: Let $(K_{RR})^{-1} = BB^t$ be the Cholesky decomposition of $(K_{RR})^{-1}$, where $B \in \mathbb{R}^{r\times r}$ is a lower triangular matrix. Then, we can write $\widehat{K}$ as $\widehat{K} = (K_R)^t B B^t K_R$. Further, let $(K_R)^t B = \widehat{V}\widehat{\Sigma}\widehat{U}^t$ be the reduced singular value decomposition[5] of $(K_R)^t B$, where $\widehat{V} \in \mathbb{R}^{n\times r}$, $\widehat{\Sigma} \in \mathbb{R}^{r\times r}$, and $\widehat{U} \in \mathbb{R}^{r\times r}$ contain the first r left singular vectors, the nonzero singular values, and the right singular vectors, respectively. Then, since $\widehat{U}^t\widehat{U} = I$, the modified kernel matrix can be expressed as

$$\widehat{K} = \widehat{V}\widehat{\Lambda}\widehat{V}^t,$$

where $\widehat{\Lambda} = \widehat{\Sigma}\widehat{\Sigma}^t$ contains the eigenvalues. Note that we can obtain the above decomposition in $\mathcal{O}(nr^2)$ time, as performing the Cholesky decomposition and the reduced singular value decomposition takes $\mathcal{O}(r^3)$ and $\mathcal{O}(nr^2)$ time, respectively (Golub & Van Loan, 1989). Using the decomposition of $\widehat{K}$ and matrix calculations as in the non-approximation case, one can derive a decomposition of the fitness $\hat{F}(\bar{\mathbf{y}})$ having the form

$$\hat{F}(\bar{\mathbf{y}}) = 1 + \bar{\mathbf{y}}^t \widehat{V}\widehat{D}\widehat{V}^t\bar{\mathbf{y}}, \quad (15)$$

where $\widehat{D} \in \mathbb{R}^{r\times r}$ is a diagonal matrix. Given the auxiliary information $\mathbf{y}^t\widehat{V} \in \mathbb{R}^{1\times r}$ for the parental individual $\mathbf{y}$, the according information for the mutated individual can be obtained in $\mathcal{O}(sr)$ time via

$$\bar{\mathbf{y}}^t\widehat{V} = \mathbf{y}^t\widehat{V} + (\bar{\mathbf{y}}_S - \mathbf{y}_S)^t\widehat{V}_S. \quad (16)$$

Hence, given the diagonal matrix $\widehat{D}$ and the auxiliary information for the parental individual $\mathbf{y}$, one can obtain the fitness $\hat{F}(\bar{\mathbf{y}})$ for $\bar{\mathbf{y}}$ in $\mathcal{O}(sr)$ time.

Theorem 2 *Assuming that $\mathbf{y}^t\widehat{V} \in \mathbb{R}^{1\times r}$ and $\widehat{D} \in \mathbb{R}^{r\times r}$ are precomputed and stored in memory, one can compute the fitness $\hat{F}(\bar{\mathbf{y}})$ for a mutated individual with $\frac{1}{2}\sum_{j=1}^n |y_j - \bar{y}_j| = s$ in $\mathcal{O}(sr)$ time.*

[4]The details are depicted on page 111. We also assume that R is selected such that K_{RR} is strictly positive definite.

[5]Only such singular vectors of $(K_R)^t B$ are calculated that correspond to nonzero singular values.

The evolutionary algorithm induced by Theorem 2 spends $\mathcal{O}(nr^2 + \mu nr + \tau\nu sn)$ time and uses $\mathcal{O}(nr + \mu n)$ space.[6] Thus, both drawbacks, the cubic preprocessing time as well as the quadratic space consumption, are reduced at the cost of approximate fitness values.

4. Experiments

We denote the evolutionary algorithms induced by the computational shortcuts depicted in Theorem 1 and Theorem 2 with EvoMMC and FastEvoMMC, respectively. Both algorithms are implemented with *Python 2.5.2* including the *numpy* package. The runtime analyses are performed on a 3 GHZ Intel Core™ Duo PC running Ubuntu 8.10.

4.1. Data Sets

Various data sets are used to test the clustering accuracy as well as the runtime performance. Following previous works (Zhang et al., 2007; Zhao et al., 2008a), we apply our approach to data sets from the UCI repository[7] (`digits`, `ionosphere`, `letter`, `satellite`) and from the MNIST database[8]. More specifically, for the `letter` and `satellite` data set each containing several classes, we use the first two classes, i.e. A vs. B and 1 vs. 2, respectively.

4.2. Parameters

For both algorithms, a couple of parameters need to be tuned, where we use a different number of initial solutions as well as a different number of mutations for EvoMMC ($\mu = 10$, $\nu = 10$) and FastEvoMMC ($\mu = 1$, $\nu = 1$), respectively. Further, we stop a run when no more fitness changes occur for a longer period of generations. In addition to these parameters, the number r determining the size of the index set R has to be selected for FastEvoMMC, which we fix to $r = 0.1n$ for all data sets except for the MNIST digits where we use $r = 0.01n$. The corresponding set R of indices is selected randomly. While there are more sophisticated methods, this choice is based upon the observation of Rifkin *et al.* (2003) who reported that random selection yields very good results in practice.

We use the RBF kernel $\exp(-||\mathbf{x} - \mathbf{x}'||^2/\sigma^2)$ with ker-

[6]The term $\tau\nu sn$ stems from copy operations which are necessary for generating mutated individuals if $\mu > 1$ and $\nu > 1$. For the remaining cases, a single individual can be updated efficiently resulting in a $\tau\nu sr$ term. In these cases, one can afford $\tau \approx nr/s$ generations until this term dominates the runtime.

[7]`http://archive.ics.uci.edu/ml/`

[8]`http://yann.lecun.com/exdb/mnist/`

nel width σ for all experiments. As reported in related works, the parameters determining the optimization problems have a significant influence on the results. To select them, we set l to $0.03n$, $0.1n$, $0.3n$, or $0.4n$ depending on the balance ratio of the specific data set. The parameters λ and σ are tuned via grid search.[9]

4.3. Clustering Accuracy

Following Xu *et al.* (2005), we evaluate the clustering accuracy for our algorithms as follows: First, all class labels available in the data sets are removed. Afterwards, the algorithms are applied resulting in a prediction vector $\mathbf{y}$ for each data set. Finally, the accuracy is measured in terms of the *clustering error* which is obtained by counting the number of discrepancies between the real labels and those predicted by $\mathbf{y}$.

Evolutionary algorithms are susceptible to the problem of local optima. Typically, this problem is faced by running the algorithms multiple times and by taking the best result out of these runs afterwards. In our experiments, we apply each algorithm 10 times to each data set and take the partition vector with the smallest clustering error out of the reported vectors. To test the significance of such a result, we repeat this procedure 10 times and average the resulting errors.

Except for the MNIST data set, all experiments are performed on the complete data sets in the way described above. As a pairwise comparison of MNIST digits involves roughly $14,000$ samples, we apply EvoMMC only to subsets containing $1,000$ digits. As mentioned in Section 3, applying EvoMMC to (much) larger data sets becomes infeasible due to the cubic preprocessing time and the quadratic storage requirement. The latter two drawbacks are shortened by FastEvoMMC and we apply this algorithm to the complete sets of digits. Further, for the UCI as well as for the MNIST data set, the averaged clustering error on all 45 pairs of digits is reported in addition to four particular comparisons of digits.

Table 1 shows the resulting clustering accuracies. Previous results, including those of the k-means (KM) algorithm, are simply transferred from the according sources. It can clearly be seen that both algorithms consistently produce good clustering results and that, except for subsets of the UCI digits, either EvoMMC or FastEvoMMC yields the most accurate ones.

[9]Similarly to Zhang *et al.* (2007), we process $(\lambda, \sigma) \in \{\frac{1}{2n}, \frac{1}{200n}, \frac{1}{1000n}\} \times \{1s, 3s, 5s\}$, where the value $s = \left(\sum_1^d \left(\max\{X^k\} - \min\{X^k\}\right)^2\right)^{1/2}$ is a rough estimate of the maximum distance between any pair of samples; here, $\{X^k\}$ denotes the set of all k-th attribute values.

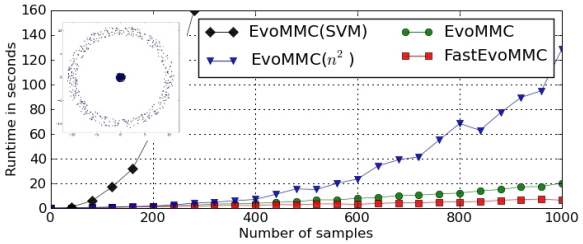


Figure 1. Runtime comparison between the different fitness computation methods on artificial data sets whose distribution is shown in the upper left corner.

Table 2. CPU time in seconds for single runs along with the average number of performed generations (in brackets).

Data	EvoMMC	FastEvoMMC
UCI Digits 3–8	20.80 (4923)	3.41 (8254)
UCI Digits 1–7	16.61 (3600)	2.92 (6978)
UCI Digits 2–7	18.40 (4374)	3.93 (9636)
UCI Digits 8–9	14.46 (3320)	3.09 (7383)
Ionosphere	14.18 (3288)	4.12 (10248)
Letter	172.09 (26867)	44.16 (66297)
Satellite	394.95 (58107)	70.44 (157971)
MNIST Digits 1–2	-	274.60 (565929)
MNIST Digits 1–7	-	304.00 (605932)

4.4. Runtime Performance

We start by comparing the efficiency of our computational shortcuts with two standard ways for obtaining the fitness values, i.e. we use Algorithm 1 and also resort to these standard techniques. The first alternative (EvoMMC(n^2)) is the quadratic time approach described at the beginning of Section 3.2. The second way (EvoMMC(SVM)) consists in applying a support vector machine to obtain fitness values, where we use the LIBSVM package (Chang & Lin, 2001) as implementation. To compare the efficiency, we apply all algorithms to a sequence of two-dimensional artificial data sets of varying sizes, see Figure 1.

For all induced evolutionary algorithms, various parameters need to be tuned. In order to evaluate the performance of the computational shortcuts, we set $\mu = 1$ and $\nu = 1$ for all four algorithms. Moreover, we set $l = 0.03n$, $\sigma = 1s$, $\lambda = 1/(1000n)$, and, for FastEvoMMC, $r = 0.1n$. The average runtime of a *single run* (measured over 10 runs) in dependence on the size of the data sets is shown in Figure 1 for each algorithm. Clearly, both EvoMMC as well as FastEvoMMC offer a significant runtime improvement over the two standard alternatives.

In addition to these experiments, we evaluate the performance of our approach on the real-world data sets. As assignments for the parameters, we use the best performing values leading to the clustering errors

Table 1. Clustering errors in % on the various data sets of KM (Hartigan & Wong, 1979), MMC (Xu et al., 2005), GMMC (Valizadegan & Jin, 2007), IterSVR (Zhang et al., 2007), CPMMC (Zhao et al., 2008a), EvoMMC, and FastEvoMMC. Bold numbers indicate the method with the lowest clustering error on the particular data set.

Data	Size	KM	MMC	GMMC	IterSVR	CPMMC	EvoMMC	FastEvoMMC
UCI Digits 3–8	357	5.32 ± 0	10	5.6	3.36 ± 0	3.08	$\mathbf{2.52 \pm 0}$	2.58 ± 0.5
UCI Digits 1–7	361	0.55 ± 0	31.25	2.2	0.55 ± 0	**0**	$\mathbf{0.0 \pm 0}$	$\mathbf{0.0 \pm 0}$
UCI Digits 2–7	356	3.09 ± 0	1.25	0.5	$\mathbf{0.0 \pm 0}$	**0.0**	$\mathbf{0.0 \pm 0}$	$\mathbf{0.0 \pm 0}$
UCI Digits 8–9	354	9.32 ± 0	3.75	16.0	3.67 ± 0	**2.26**	3.22 ± 0.40	3.78 ± 4.13
Ionosphere	351	32 ± 17.9	21.25	23.5	32.3 ± 16.6	27.64	$\mathbf{17.94 \pm 6.84}$	18.75 ± 3.19
Letter	1,555	17.94 ± 0	-	-	7.2 ± 0	5.53	3.67 ± 0.33	$\mathbf{3.27 \pm 1.08}$
Satellite	2,236	4.07 ± 0	-	-	3.18 ± 0	1.52	1.16 ± 0.45	$\mathbf{1.14 \pm 0.65}$
UCI Digits	1,797	2.62	-	-	1.82	**0.62**	0.72	0.81
MNIST Digits	70,000	10.79	-	-	7.59	4.29	(3.23)	**3.45**

shown in Table 1. The average runtimes as well as the average number of performed generations for *single runs* (measured over 10 runs) are depicted in Table 2. They demonstrate that especially FastEvoMMC can handle large data sets within a reasonable amount of time. The latter results also show that both methods can efficiently perform a huge number of generations in order to converge to (local) optima.[10]

Both IterSVR and CPMMC have a superior runtime performance compared to previous approaches (Zhao et al., 2008a). Hence, we will focus on indicating the (theoretical) runtime of our approach with respect to these techniques. The runtime of IterSVR depends on the specific implementation for solving the intermediate optimizations problems. For instance, Zhang *et al.* (2007) resort to LIBSVM. The CPMMC approach is guaranteed to converge after $\frac{CR}{\varepsilon^2}$ iterations, where ε is a user-supplied constant and where R is a constant being independent of n. Further, the runtime per iteration is bounded by $\mathcal{O}(sn)$, where s is the number of nonzero features. The results for CPMMC reported by Zhao *et al.* are based on linear kernels. Nonlinear kernels, however, can only be applied by decomposing the kernel matrix to form an empirical feature map beforehand spending $\mathcal{O}(n^3)$ time in the worst case. This can be accelerated, for example, by using a similar approximation as in Section 3.3 spending $O(nr^2)$ time, where r determines the number of features in the map.

In our experiments we observed that even a slight variation of the parameters of IterSVR and CPMMC may lead to significant differences in the actual runtimes. The dependence on both different kernel functions and parameter selections render a detailled runtime comparison beyond the scope of this paper. A systematic investigation of the runtimes in relation to the clustering errors will be subject of future work.

[10]Each generation is performed efficiently, but a huge number of generations is "wasted" for fine tuning (i.e. for the correct assignment of the last few elements). The probability for switching wrong cluster assignments becomes extremely small at the end of the search. Heuristic extensions like intermediate classification steps or a k-means preprocessing phase could be used to accelerate the search.

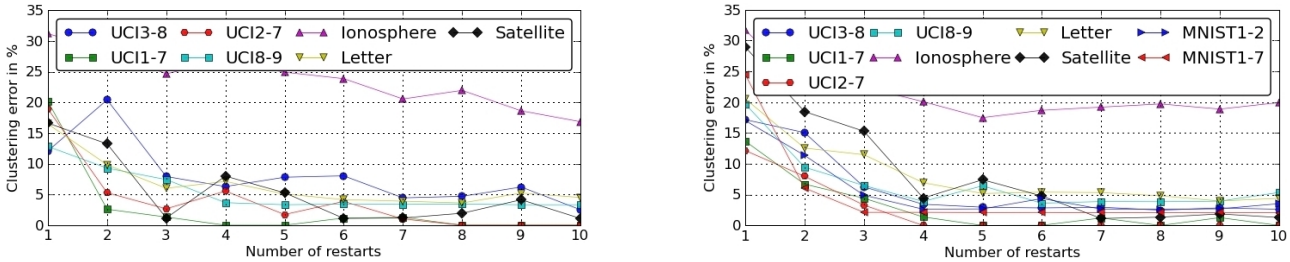
4.5. Number of Restarts

To assess how many restarts are needed for satisfying results, we investigate the number of restarts against the clustering error, i.e. m runs are performed and the lowest obtained clustering error is taken as result. For each m, this procedure is repeated 10 times and the averaged results are reported. Again, we select the values leading to the results given in Table 1 as values for the parameters. The plots given in Figure 2 show that a small number of restarts is sufficient to yield good clustering errors. Further, they demonstrate that both methods perform similarly on the data sets, i.e. the different assignments for the parameters μ and ν seem to have little influence on the clustering error.

5. Conclusions and Discussion

Maximum margin clustering methods turn out to be an effective approach to the clustering problem. Aiming at practical methods, we proposed a stochastic optimization approach along with an efficient computational shortcut making the overall algorithm capable of dealing with large data sets. Our experimental analyses reveal that our approach yields better clustering accuracies than competitive methods in most cases.

We expect our approach to be extendible in various situations. For instance, our stochastic search along with the computational shortcuts could also be applied as a fine-tuning step after a k-means preprocessing phase. As the partitions obtained by such a preprocessing would already constitute reasonable solutions, we expect a considerable runtime improvement by the preprocessing. We plan to investigate such extensions.



(a) EvoMMC (b) FastEvoMMC

Figure 2. Restarts vs. clustering error.

Acknowledgments

We thank the anonymous reviewers for their careful reading and detailed comments. This work was supported in part by the Academy of Finland.

References

Adankon, M. M., & Cheriet, M. (2007). Learning semi-supervised svm with genetic algorithm. *Proc. International Joint Conference on Neural Networks* (pp. 1825–1830).

Bach, F., & Harchaoui, Z. (2008). Diffrac: a discriminative and flexible framework for clustering. *Adv. in Neural Information Proc. Systems 20* (pp. 49–56).

Beyer, H.-G., & Schwefel, H.-P. (2002). Evolution strategies - A comprehensive introduction. *Natural Computing*, *1*, 3–52.

Chang, C.-C., & Lin, C.-J. (2001). *LIBSVM: a library for support vector machines*. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

Golub, G. H., & Van Loan, C. (1989). *Matrix computations*. Baltimore and London: Johns Hopkins University Press. Second edition.

Hartigan, J. A., & Wong, M. A. (1979). A K-means clustering algorithm. *Applied Statistics*, *28*, 100–108.

Jain, A. K., & Dubes, R. C. (1988). *Algorithms for clustering data*. Prentice Hall.

Rifkin, R., Yeo, G., & Poggio, T. (2003). Regularized least-squares classification. In *Adv. in learning theory: Methods, models and applications*. IOS Press.

Rifkin, R. M. (2002). *Everything old is new again: A fresh look at historical approaches in machine learning*. Doctoral dissertation, MIT.

Schölkopf, B., Herbrich, R., & Smola, A. J. (2001). A generalized representer theorem. *Proc. 14th Annual Conference on Computational Learning Theory* (pp. 416–426).

Silva, M. M., Maia, T. T., & Braga, A. P. (2005). An evolutionary approach to transduction in support vector machines. *Proc. 5th International Conference on Hybrid Intelligent Systems* (pp. 329–334).

Steinwart, I., & Christmann, A. (2008). *Support vector machines*. New York, NY, USA: Springer-Verlag.

Suykens, J. A. K., & Vandewalle, J. (1999). Least squares support vector machine classifiers. *Neural Processing Letters*, *9*, 293–300.

Valizadegan, H., & Jin, R. (2007). Generalized maximum margin clustering and unsupervised kernel learning. *Adv. in Neural Information Proc. Systems 19* (pp. 1417–1424).

Vapnik, V. (1998). *Statistical learning theory*. New York: Wiley.

Xu, L., Neufeld, J., Larson, B., & Schuurmans, D. (2005). Maximum margin clustering. *Adv. in Neural Information Proc. Systems 17* (pp. 1537–1544).

Xu, L., & Schuurmans, D. (2005). Unsupervised and semi-supervised multi-class support vector machines. *Proc. National Conference on Artificial Intelligence* (pp. 904–910).

Zhang, K., Tsang, I. W., & Kwok, J. T. (2007). Maximum margin clustering made practical. *Proc. International Conference on Machine Learning* (pp. 1119–1126).

Zhao, B., Wang, F., & Zhang, C. (2008a). Efficient maximum margin clustering via cutting plane algorithm. *Proc. SIAM International Conference on Data Mining* (pp. 751–762).

Zhao, B., Wang, F., & Zhang, C. (2008b). Efficient multiclass maximum margin clustering. *Proc. International Conference on Machine Learning* (pp. 1248–1255).