Sequential Learning with LS-SVM for Large-Scale Data Sets

Tobias Jung[1] and Daniel Polani[2]

[1] Department of Computer Science, University of Mainz, Germany
[2] School of Computer Science, University of Hertfordshire, UK

Abstract. We present a subspace-based variant of LS-SVMs (i.e. regularization networks) that sequentially processes the data and is hence especially suited for online learning tasks. The algorithm works by selecting from the data set a small subset of basis functions that is subsequently used to approximate the full kernel on arbitrary points. This subset is identified online from the data stream. We improve upon existing approaches (esp. the kernel recursive least squares algorithm) by proposing a new, supervised criterion for the selection of the relevant basis functions that takes into account the approximation error incurred from approximating the kernel as well as the reduction of the cost in the original learning task. We use the large-scale data set 'forest' to compare performance and efficiency of our algorithm with greedy batch selection of the basis functions via orthogonal least squares. Using the same number of basis functions we achieve comparable error rates at much lower costs (CPU-time and memory wise).

1 Introduction and Related Work

Introduction. In this paper we adress the problem of sequential learning when the predictor has the form of least squares SVM (LS-SVM). Since there is no way we can achieve this using in our model one independent parameter for each training example (i.e. basis function), we use a projection-based technique that only considers a small subset of all possible basis functions. This subset is selected online from the training data by just inspecting the most recent example. Our resulting algorithm is conceptually similar to the kernel recursive least squares (KRLS) algorithm proposed in [4], yet improves it in two important ways: one is that we consider a *supervised* criterion for the selection of the relevant basis functions that takes into account the reduction of the cost in the original learning task in addition to the error incurred from approximating the kernel. Since the per-step complexity only depends on the size of the subset, making sure that no unnecessary basis functions are selected ensures more efficient usage of otherwise scarce resources. And second, by considering a *pruning* operation we can also delete basis functions from the subset to have an even tighter control over its size. Overall the algorithm is very resource efficient, and only depends on the number of examples stored in the subset.

Related work. The unfavorable $\mathcal{O}(n^3)$ scaling of kernel-based learning has spawned a number of approaches where the exact solution is approximated by a solution with lower complexity. Well known examples are the Nyström method [13] or the subset of regressors method (SR), mentioned e.g. in [7, 12, 10]. Both methods work by projecting the kernel onto a much smaller subset of kernels chosen from the full data, say of size $m \ll n$, and reduce computational complexity to $\mathcal{O}(nm^2)$. To select the subset we can categorize the various approaches as being unsupervised and supervised. Unsupervised approaches like random selection [13] or the incomplete Cholesky decomposition (IC) [5] do not use information about the task we want to solve, i.e. the response variable we wish to regress upon. Random selection does not use any information at all whereas IC aims at reducing the error from approximating the kernel matrix. Supervised choice of the subset does take into account the response variable and usually proceeds by greedy forward selection, using e.g. matching pursuit techniques [11] or the recent Cholesky decomposition with side information [1]. However, none of these approaches are directly applicable for sequential learning, since they all use information from the complete data set. Working in the context of Gaussian process regression (GPR), [2] and also [4] have proposed an online variant, which adds examples directly from the data stream and is the basis of our work presented here.

2 Background

Traditional setup. Given t examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^t$ with $\mathbf{x}_i \in \mathcal{X} \subset \mathbb{R}^d$ being the inputs and $y_i \in \mathcal{Y} \subset \mathbb{R}$ being the outputs, the goal is to reconstruct (learn) the underlying function. Consider as the space of candidate functions the reproducing kernel Hilbert space (RKHS) $\mathcal{H}_k$ of functions $f : \mathcal{X} \to \mathcal{Y}$ endowed with reproducing kernel k, where $k : \mathcal{X} \times \mathcal{X} \to \mathcal{Y}$ is a symmetric, positive definite function (e.g. think of Gaussian RBF). The underlying function can be reconstructed solving the Tikhonov functional: $\min_{f \in \mathcal{H}_k} J[f] = \sum_{i=1}^t \big(y_i - f(\mathbf{x}_i)\big)^2 + \gamma \left\|f\right\|_{\mathcal{H}_k}^2$ with $\gamma > 0$ being the regularization parameter. The Representer theorem tells us that any solution to this variational problem has a representation in the form $f(\cdot) = \sum_{i=1}^t \beta_i k(\mathbf{x}_i, \cdot)$ i.e. as a sum of kernels centered on the data. Plugging this back into the original variational problem leads to the optimization problem

$$\min_{\beta \in \mathbb{R}^t} \left\|\mathbf{y} - \mathbf{K}\boldsymbol{\beta}\right\|^2 + \gamma \boldsymbol{\beta}^T \mathbf{K} \boldsymbol{\beta} \tag{1}$$

with $\mathbf{y}$ being the $t \times 1$ vector of observations, $\mathbf{K}$ being the dense $t \times t$ kernel matrix $[\mathbf{K}]_{ij} = k(\mathbf{x}_i, \mathbf{x}_j)$ of pairwise similarities and $\boldsymbol{\beta}$ being a $t \times 1$ vector. From (1) the coefficients $\boldsymbol{\beta}$ can be obtained by solving

$$\big(\mathbf{K}^T\mathbf{K} + \gamma\mathbf{K}\big)\boldsymbol{\beta} = \mathbf{K}^T\mathbf{y} \tag{2}$$

which gives the solution as $\boldsymbol{\beta} = (\mathbf{K} + \gamma\mathbf{I})^{-1}\mathbf{y}$ due to $\mathbf{K}$ being symmetric and positive definite. Solving (2) is generally a matter of $\mathcal{O}(t^3)$ operations. The overbearing computational burden stems from the fact that every training example will contribute one parameter to the resulting model.

The subset of regressors method (SR). Consider a subset $\{\tilde{\mathbf{x}}_i\}_{i=1}^m$, $m \ll t$, of data points selected from the full set $\{\mathbf{x}_i\}_{i=1}^t$, without loss of generality assume that these are the first m examples. We approximate the kernel on arbitrary points through linear combination of kernels from the subset (termed the *dictionary* or set of *basis vectors* $\mathcal{BV}$ in [2] which we adopt for the remainder of this paper) in the following way: $k(\mathbf{x}, \cdot) \approx \sum_{i=1}^m a_i k(\tilde{\mathbf{x}}_i, \cdot)$. The $m \times 1$ vector $\mathbf{a} = (a_1, \ldots, a_m)^T$ is determined such that the distance in $\mathcal{H}_k$ for a given $\mathbf{x}$

$$\delta = \min_{\mathbf{a}} \left\| k(\mathbf{x}, \cdot) - \sum_{i=1}^m a_i k(\tilde{\mathbf{x}}_i, \cdot) \right\|_{\mathcal{H}_k}^2 \tag{3}$$

is minimized. The solution to this problem follows as

$$\mathbf{a} = \mathbf{K}_{mm}^{-1} \mathbf{k}_m(\mathbf{x}) \tag{4}$$

where the $m \times m$ matrix $\mathbf{K}_{mm}$ is the kernel matrix corresponding to the dictionary (i.e. the upper left $m \times m$ submatrix of $\mathbf{K}$) and $m \times 1$ vector $\mathbf{k}_m(\mathbf{x})$ is shorthand for vector $\mathbf{k}_m(\mathbf{x}) = (k(\tilde{\mathbf{x}}_1, \mathbf{x}), \ldots, k(\tilde{\mathbf{x}}_m, \mathbf{x}))^T$. For arbitrary $\mathbf{x}, \mathbf{x}'$ we thus have the approximation

$$k(\mathbf{x}, \mathbf{x}') \approx [\mathbf{k}_m(\mathbf{x})]^T \mathbf{K}_{mm}^{-1} \mathbf{k}_m(\mathbf{x}'). \tag{5}$$

If either $\mathbf{x}$ or $\mathbf{x}'$ are in $\mathcal{BV}$ then (5) is exact. Replacing the true kernel by (5) gives $\mathbf{K}_{tm} \mathbf{K}_{mm}^{-1} \mathbf{K}_{tm}^T \approx \mathbf{K}$ as an approximation to the true kernel matrix $\mathbf{K}$, where $\mathbf{K}_{tm}$ is the $t \times m$ submatrix of the first m columns of $\mathbf{K}$ (again, corresponding to the $\mathcal{BV}$). Defining the $t \times m$ matrix $\mathbf{A}$ with rows $\mathbf{a}_i^T = \mathbf{K}_{mm}^{-1} \mathbf{k}_m(\mathbf{x}_i)$, $i = 1, \ldots, t$ from (4) we can write

$$\mathbf{K}_{tm} = \mathbf{A} \mathbf{K}_{mm}. \tag{6}$$

In the SR-method [7, 11, 10] instead of using the full representation one only uses the kernels in $\mathcal{BV}$, i.e. $f(\cdot) = \sum_{i=1}^m \beta_i k(\tilde{\mathbf{x}}_i, \cdot)$, and obtain in place of (1) the penalized least squares problem

$$\min_{\boldsymbol{\beta} \in \mathbb{R}^m} \|\mathbf{y} - \mathbf{K}_{tm} \boldsymbol{\beta}\|^2 + \gamma \boldsymbol{\beta}^T \mathbf{K}_{mm} \boldsymbol{\beta} \tag{7}$$

which has the solution $\boldsymbol{\beta} = \left(\mathbf{K}_{tm}^T \mathbf{K}_{tm} + \gamma \mathbf{K}_{mm}\right)^{-1} \mathbf{K}_{tm}^T \mathbf{y}$. Despite its cursory similarity with (2) we have gained much since now we are only dealing with m parameters and computational complexity is down to $\mathcal{O}(tm^2)$.

Csató and Opper's sparse greedy online approximation. Still, the SR-method is not directly applicable for online learning. Assume that the data arrives sequentially at $t = 1, 2, \ldots$ and that only one pass over the data set is possible, so that we cannot select the subset $\mathcal{BV}$ in advance. Working in the context of GPR, [2] and later [4] have proposed sparse greedy online approximation: start from an empty set $\mathcal{BV}$ and examine at every time step t, if the current example needs to be included in $\mathcal{BV}$ or if it can be processed without

augmenting $\mathcal{BV}$. The approximation in (5) is modified such that it uses the most recent version of $\mathcal{BV}$ and sets to zero those entries from $\mathbf{a}$ that correspond to basis vectors added in future time steps (denoted by $\tilde{\mathbf{A}}$). Thus the matrix used in (7) no longer equals the submatrix $\mathbf{K}_{tm}$ from (6), since now $\tilde{\mathbf{K}}_{tm} =_{\text{def}} \tilde{\mathbf{A}}\mathbf{K}_{mm}$ is only an approximation.

The one crucial advantage of this approach is that now we can use (penalized) least squares methods as in (7) together with online growing and pruning operations for sequential learning by using only the examples memorized in the set $\mathcal{BV}$. (Otherwise, to augment or prune an existing model we would need to work with all previously seen data or resort to a window of a fixed given size.)

3 Time-recursive LS-SVM

In this section we present the main contribution of our work: online LS-SVM using sparse online approximation and a novel criterion for the selection of relevant basis functions to include in the subset. The algorithm works along the lines of recursive least squares, i.e. propagates forward the inverse of the cross product matrix.

Let t be the current time step, $(\mathbf{x}_{t+1}, y^*)$ the currently observed input-output pair and assume that from the past t examples $\{(\mathbf{x}_i, y_i)\}_{i=1}^{t}$ the m examples $\{\tilde{\mathbf{x}}_i\}_{i=1}^{m}$ were selected into the dictionary $\mathcal{BV}$. Consider the penalized least squares problem that is LS-SVM (restated here from (7) for clarity)

$$\min_{\boldsymbol{\beta}\in\mathbb{R}^m} J_{tm}(\boldsymbol{\beta}) = \left\|\mathbf{y}_t - \tilde{\mathbf{K}}_{tm}\boldsymbol{\beta}\right\|^2 + \gamma\boldsymbol{\beta}^T\mathbf{K}_{mm}\boldsymbol{\beta} \tag{8}$$

with $\tilde{\mathbf{K}}_{tm} = \tilde{\mathbf{A}}\mathbf{K}_{mm}$ being the (approximated) $t \times m$ design matrix from (6) and $\mathbf{y}_t$ being the $t \times 1$ vector of the observed output values. Note that we are using a double index to indicate the dependence on the number of examples t and the number of basis functions m. If we define the $m \times m$ cross product matrix $\mathbf{P}_{tm} = (\tilde{\mathbf{K}}_{tm}^T\tilde{\mathbf{K}}_{tm} + \gamma\mathbf{K}_{mm})$ then the solution to (8) is given by $\boldsymbol{\beta}_{tm} = \mathbf{P}_{tm}^{-1}\tilde{\mathbf{K}}_{tm}^T\mathbf{y}_t$. Finally we introduce the costs $\xi_{tm} = J_{tm}(\boldsymbol{\beta}_{tm})$. Assuming that $\{\mathbf{P}_{tm}^{-1},\ \boldsymbol{\beta}_{tm},\ \xi_{tm}\}$ are known from previous computations, every time a new example $(\mathbf{x}_{t+1}, y^*)$ is presented we will perform one or more of the following update operations:

1. *Normal step*: Process $(\mathbf{x}_{t+1}, y^*)$ in the usual way using the fixed set of basis functions $\mathcal{BV}$.
2. *Growing step*: If the new example is sufficiently different from the previous examples in $\mathcal{BV}$ (i.e. the reconstruction error in (3) exceeds a given threshold) and strongly contributes to the solution of the problem (i.e. the decrease of the loss when adding the new basis function is greater than a given threshold) then the current example is added to $\mathcal{BV}$ and the number of basis functions in the model is increased by one.
3. *Pruning step*: If the current size of the $\mathcal{BV}$ set exceeds the allowed maximum number of $\mathcal{BV}$s specified prior to starting the algorithm, remove from $\mathcal{BV}$ the basis function that contributes the least to the reduction of the cost function.

Integral to these updates are two well-known matrix identities for recursively computing the inverse of a matrix: (for suitable matrices)

$$\text{if } \mathbf{B}_{t+1} = \mathbf{B}_t + \mathbf{b}\mathbf{b}^T \text{ then } \mathbf{B}_{t+1}^{-1} = \mathbf{B}_t^{-1} - \frac{\mathbf{B}_t^{-1}\mathbf{b}\mathbf{b}^T\mathbf{B}_t^{-1}}{1 + \mathbf{b}^T\mathbf{B}_t^{-1}\mathbf{b}} \tag{9}$$

which is used when adding a row to the design matrix. Likewise,

$$\text{if } \mathbf{B}_{t+1} = \begin{bmatrix} \mathbf{B}_t & \mathbf{b} \\ \mathbf{b}^T & b^* \end{bmatrix} \text{ then } \mathbf{B}_{t+1}^{-1} = \begin{bmatrix} \mathbf{B}_t^{-1} & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} + \frac{1}{\Delta_b} \begin{bmatrix} -\mathbf{B}_t^{-1}\mathbf{b} \\ 1 \end{bmatrix} \begin{bmatrix} -\mathbf{B}_t^{-1}\mathbf{b} \\ 1 \end{bmatrix}^T \tag{10}$$

with $\Delta_b = b^* - \mathbf{b}^T\mathbf{B}_t^{-1}\mathbf{b}$. This second update is used when adding a column to the design matrix.

3.1 Normal step: from $\{\mathbf{P}_{tm}^{-1}, \boldsymbol{\beta}_{tm}, \boldsymbol{\xi}_{tm}\}$ to $\{\mathbf{P}_{t+1,m}^{-1}, \boldsymbol{\beta}_{t+1,m}, \boldsymbol{\xi}_{t+1,m}\}$

Let $\mathbf{k}_{t+1}$ be $\mathbf{k}_{t+1} = (k(\tilde{\mathbf{x}}_1, \mathbf{x}_{t+1},), \ldots, k(\tilde{\mathbf{x}}_m, \mathbf{x}_{t+1}))^T$, then

$$\tilde{\mathbf{K}}_{t+1,m} = \begin{bmatrix} \tilde{\mathbf{K}}_{tm} \\ \mathbf{k}_{t+1}^T \end{bmatrix} \quad \text{and} \quad \mathbf{y}_{t+1} = \begin{bmatrix} \mathbf{y}_t \\ y^* \end{bmatrix}.$$

Thus $\mathbf{P}_{t+1,m} = \mathbf{P}_{tm} + \mathbf{k}_{t+1}\mathbf{k}_{t+1}^T$ and we obtain from (9) the well-known RLS updates

$$\begin{aligned} \mathbf{P}_{t+1,m}^{-1} &= \mathbf{P}_{tm}^{-1} - \frac{\mathbf{P}_{tm}^{-1}\mathbf{k}_{t+1}\mathbf{k}_{t+1}^T\mathbf{P}_{tm}^{-1}}{\Delta}, \qquad \boldsymbol{\beta}_{t+1,m} = \boldsymbol{\beta}_{tm} + \frac{\varrho}{\Delta}\mathbf{P}_{tm}^{-1}\mathbf{k}_{t+1} \\ \xi_{t+1,m} &= \xi_{tm} + \frac{\varrho^2}{\Delta} \end{aligned} \tag{11}$$

with scalars $\Delta = 1 + \mathbf{k}_{t+1}^T\mathbf{P}_{tm}^{-1}\mathbf{k}_{t+1}$ and $\varrho = y^* - \mathbf{k}_{t+1}^T\boldsymbol{\beta}_{tm}$. The set $\mathcal{BV}$ is not altered during this step. Operation count is $\mathcal{O}(m^2)$.

3.2 Growing step: from $\{\mathbf{P}_{tm}^{-1}, \boldsymbol{\beta}_{tm}, \boldsymbol{\xi}_{tm}\}$ to $\{\mathbf{P}_{t,m+1}^{-1}, \boldsymbol{\beta}_{t,m+1}, \boldsymbol{\xi}_{t,m+1}\}$

How to add a $\mathcal{BV}$. When adding an additional basis function (centered on $\mathbf{x}_{t+1}$) to the model we augment the set $\mathcal{BV}$ with $\tilde{\mathbf{x}}_{m+1}$ (note that $\tilde{\mathbf{x}}_{m+1}$ is the same as $\mathbf{x}_{t+1}$ from above). Again, define $\mathbf{k}_{t+1} = (k(\tilde{\mathbf{x}}_1, \tilde{\mathbf{x}}_{m+1}), \ldots, k(\tilde{\mathbf{x}}_m, \tilde{\mathbf{x}}_{m+1}))^T$ and $k^* = k(\tilde{\mathbf{x}}_{m+1}, \tilde{\mathbf{x}}_{m+1})$. Adding a basis function means appending a new $t \times 1$ vector $\mathbf{q}$ to the design matrix and appending $\mathbf{k}_{t+1}$ as row/column to the penalty matrix $\mathbf{K}_{mm}$, thus

$$\mathbf{P}_{t,m+1} = \begin{bmatrix} \tilde{\mathbf{K}}_{tm} & \mathbf{q} \end{bmatrix}^T \begin{bmatrix} \tilde{\mathbf{K}}_{tm} & \mathbf{q} \end{bmatrix} + \gamma \begin{bmatrix} \mathbf{K}_{mm} & \mathbf{k}_{t+1} \\ \mathbf{k}_{t+1}^T & k^* \end{bmatrix}.$$

Invoking (10) we obtain the updated inverse $\mathbf{P}_{t,m+1}^{-1}$ via

$$\mathbf{P}_{t,m+1}^{-1} = \begin{bmatrix} \mathbf{P}_{tm}^{-1} & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} + \frac{1}{\Delta_b} \begin{bmatrix} -\mathbf{w}_b \\ 1 \end{bmatrix} \begin{bmatrix} -\mathbf{w}_b \\ 1 \end{bmatrix}^T \tag{12}$$

where simple but tedious vector algebra reveals that

$$\begin{aligned}\mathbf{w}_b &= \mathbf{P}_{tm}^{-1}(\tilde{\mathbf{K}}_{tm}^T\mathbf{q} + \gamma\mathbf{k}_{t+1}) \\ \Delta_b &= \mathbf{q}^T\mathbf{q} + \gamma k^* - (\tilde{\mathbf{K}}_{tm}^T\mathbf{q} + \gamma\mathbf{k}_{t+1})^T\mathbf{w}_b. \end{aligned} \tag{13}$$

Without sparse online approximation this step requires us to recall all past examples $\{\mathbf{x}_i\}_{i=1}^t$ since $\mathbf{q}$ is given by $\mathbf{q}^T = (k(\tilde{\mathbf{x}}_{m+1}, \mathbf{x}_1), \ldots, k(\tilde{\mathbf{x}}_{m+1}, \mathbf{x}_t))^T$ and just obtaining (13) would come at the undesirable price of $\mathcal{O}(tm)$. However, we are going to get away with merely $\mathcal{O}(m)$ operations and only need to memorize examples in $\mathcal{BV}$. Due to the sparse approximation $\mathbf{q}$ is actually of the form $\mathbf{q}^T = \begin{bmatrix}\tilde{\mathbf{K}}_{t-1,m}\mathbf{a}_{t+1} & k^*\end{bmatrix}^T$ with $\mathbf{a}_{t+1} = \mathbf{K}_{mm}^{-1}\mathbf{k}_{t+1}$ from (4). Hence new information is injected only through the last component. Exploiting this special structure of $\mathbf{q}$ equation (13) becomes

$$\begin{aligned}\mathbf{w}_b &= \mathbf{a}_{t+1} + \frac{\delta}{\Delta}\mathbf{P}_{t-1,m}^{-1}\mathbf{k}_{t+1} \\ \Delta_b &= \frac{\delta^2}{\Delta} + \gamma\delta \end{aligned} \tag{14}$$

where $\delta = k^* - \mathbf{k}_{t+1}^T\mathbf{a}_{t+1}$ from (3). If we cache and reuse those terms already computed in the preceding step (see Sect. 3.1) then we can obtain $\mathbf{w}_b, \Delta_b$ in $\mathcal{O}(m)$ operations.

To obtain the updated coefficients $\boldsymbol{\beta}_{t,m+1}$ we first multiply (12) from the right side by $\tilde{\mathbf{K}}_{t,m+1}^T\mathbf{y}_t = \begin{bmatrix}\tilde{\mathbf{K}}_{tm}^T\mathbf{y}_t & \mathbf{q}^T\mathbf{y}_t\end{bmatrix}^T$ and get

$$\boldsymbol{\beta}_{t,m+1} = \begin{bmatrix}\boldsymbol{\beta}_{tm} \\ 0\end{bmatrix} + \kappa\begin{bmatrix}-\mathbf{w}_b \\ 1\end{bmatrix} \tag{15}$$

where scalar κ is defined by $\kappa = \mathbf{y}_t^T(\mathbf{q} - \tilde{\mathbf{K}}_{tm}\mathbf{w}_b)/\Delta_b$. Again we can now exploit the special structure of $\mathbf{q}$ to show that κ is equal to

$$\kappa = -\frac{\delta\varrho}{\Delta_b\Delta}$$

And again we can reuse terms computed in the previous step (see Sect. 3.1).

Skipping the necessary computations, we can show that the reduced (regularized) cost $\xi_{t,m+1}$ is recursively obtained from ξ_{tm} via the expression:

$$\xi_{t,m+1} = \xi_{tm} - \kappa^2\Delta_b. \tag{16}$$

Finally, every time we add an example to the $\mathcal{BV}$ set we must also update the inverse kernel matrix $\mathbf{K}_{mm}^{-1}$ needed during the computation of $\mathbf{a}_{t+1}$ and δ. This can be done using the formula for partitioned matrix inverses (10).

When to add a $\mathcal{BV}$. To decide whether or not the current example $\mathbf{x}_{t+1}$ should be added to the $\mathcal{BV}$ set, we employ a two-part criterion, similar to the one used in resource-allocating networks [8]. The first part measures the 'novelty' of the

current example: only examples that are 'far' from those already stored in the $\mathcal{BV}$ set are considered for inclusion. To this end we compute as in [2, 4] the squared norm of the residual from projecting (in RKHS) the example onto the span of the current $\mathcal{BV}$ set, i.e. we compute (restated from (3)) $\delta = k^* - \mathbf{k}_{t+1}^T \mathbf{a}_{t+1}$. If $\delta <$ `TOL1` for a given threshold `TOL1`, then $\mathbf{x}_{t+1}$ is well represented by the given $\mathcal{BV}$ set and its inclusion would not contribute much to reduce the error from approximating the kernel by the reduced set. On the other hand, if $\delta >$ `TOL1` then $\mathbf{x}_{t+1}$ is not well represented by the current $\mathcal{BV}$ set and leaving it behind could incur a large error in the approximation of the kernel.

However, using as sole criterion the reduction of the error incurred from approximating the kernel is probably too wasteful of resources, since examples could get selected into the subset that are unrelated to the original task [1]. We want to be more restrictive, particularly because the computational complexity per step scales with the square of basis functions in $\mathcal{BV}$ (so that the size of $\mathcal{BV}$ will soon become the limiting factor). Aside from novelty, here we thus consider as second part of the selection criterion the 'usefulness' of a basis function candidate. Usefulness is taken to be its contribution to the reduction of the regularized costs, i.e. the term $\kappa^2 \Delta_b$ from (16). Both parts together are combined into one rule: only if $\delta \cdot \kappa^2 \cdot \Delta_b >$ `TOL2` then the current example will become a new basis function and will be added to $\mathcal{BV}$.

3.3 Pruning step: from $\{\mathbf{P}_{tm}^{-1}, \boldsymbol{\beta}_{tm}, \boldsymbol{\xi}_{tm}\}$ to $\{\mathbf{P}_{t,m\setminus i}^{-1}, \boldsymbol{\beta}_{t,m\setminus i}, \boldsymbol{\xi}_{t,m\setminus i}\}$

How to delete a $\mathcal{BV}$. First consider the case when we are trying to delete the last one. Take as starting point eqs. (12),(15),(16) and switch the role of old and new: eq. (12) becomes

$$\begin{bmatrix} \mathbf{P}_{t,m-1}^{-1} & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} = \mathbf{P}_{tm}^{-1} - \frac{1}{\Delta_b} \begin{bmatrix} \mathbf{w}_b \\ -1 \end{bmatrix} \begin{bmatrix} \mathbf{w}_b \\ -1 \end{bmatrix}^T .$$

Both Δ_b and $\mathbf{w}_b$ can be obtained directly from $\mathbf{P}_{tm}^{-1}$: defining the $(m-1) \times 1$ vector $\mathbf{u}$ by $\mathbf{P}_{tm}^{-1}(1 : m-1, m)$ (i.e. the first $m-1$ rows of the m-th column) and scalar u^* by $\mathbf{P}_{tm}^{-1}(m, m)$ (i.e. the m-th diagonal element) we find $\Delta_b = 1/u^*$ and $\mathbf{w}_b = \mathbf{u}/u^*$. Hence

$$\begin{bmatrix} \mathbf{P}_{t,m-1}^{-1} & \mathbf{0} \\ \mathbf{0} & 0 \end{bmatrix} = \mathbf{P}_{tm}^{-1} - \frac{1}{u^*} \begin{bmatrix} \mathbf{u} \\ -1 \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ -1 \end{bmatrix}^T \tag{17}$$

where the left side is truncated to yield the $(m-1) \times (m-1)$ matrix $\mathbf{P}_{t,m-1}^{-1}$.

Likewise, to obtain $\boldsymbol{\beta}_{t,m-1}$ from $\boldsymbol{\beta}_{tm}$ we turn around update (15)

$$\begin{bmatrix} \boldsymbol{\beta}_{t,m-1} \\ 0 \end{bmatrix} = \boldsymbol{\beta}_{tm} - \kappa \begin{bmatrix} -\mathbf{w}_b \\ 1 \end{bmatrix} .$$

Again, we can see from update (15) that κ actually is the last component of $\boldsymbol{\beta}_{tm}$. So, defining $b^* = \boldsymbol{\beta}_{tm}(m)$ we get

$$\begin{bmatrix} \boldsymbol{\beta}_{t,m-1} \\ 0 \end{bmatrix} = \boldsymbol{\beta}_{tm} + \frac{b^*}{u^*} \begin{bmatrix} \mathbf{u} \\ -1 \end{bmatrix} \tag{18}$$

where the left side is truncated to yield the $(m-1) \times 1$ vector $\boldsymbol{\beta}_{t,m-1}$.

Finally, to obtain $\xi_{t,m-1}$ from ξ_{tm} we turn around update (16) to yield

$$\xi_{t,m-1} = \xi_{tm} + (b^*)^2/u^*. \tag{19}$$

If we need to delete an arbitrary basis function $i \in \{1, \ldots, m\}$ instead of just the m-th one, we exploit the fact that reordering the indices of the basis function within the set $\mathcal{BV}$ is equivalent to reordering the columns/rows of $\mathbf{P}_{tm}^{-1}$. So, to delete basis function i we just swap column/row i and m in all necessary places (i.e. in $\mathbf{P}_{tm}^{-1}, \boldsymbol{\beta}_{tm}, \mathbf{K}_{mm}$ and $\mathcal{BV}$). Afterwards we apply (17),(18),(19) as described above. Overall, deleting a basis function requires $\mathcal{O}(m^2)$ operations.

When to delete a $\mathcal{BV}$. To identify from the $\mathcal{BV}$ set the basis function best suited for removal we consider their contribution to the cost function. Compute as in (19) the score

$$\varepsilon_i = \frac{\boldsymbol{\beta}_{tm}(i)^2}{\mathbf{P}_{tm}^{-1}(i,i)} \qquad i = 1, \ldots, m$$

for every basis function in $\mathcal{BV}$ and delete the one with the lowest score. The computation of this criterion is very cheap and requires only $\mathcal{O}(m)$ operations.

4 Experiments

Comparing subset selection criteria. First, we compare our supervised approach with the unsupervised method used in the related KRLS algorithm [4]. As third competitor we consider greedy forward selection via orthogonal least squares (OLS). All three methods use the same dictionary of basis function candidates (built from RBF-kernels centered on the training data) to choose the subset from; note though that OLS is a batch method, whereas our method and KRLS process the data sequentially. We chose three well-known problems: the artifical *sinc* data set (noise $\sigma = 0.2$), and the small scale benchmarks *boston* (train 400, test 106) and *abalone* (train 3000, test 1177) from the UCI repository [3]. The data was scaled to have zero mean and unit variance. Parameters governing subset selection were `TOL1`=10^{-2} and `TOL2`=10^{-4}; for OLS we used the GCV as stopping criterion. The remaining parameters were set as in [6]. Since our method and KRLS depend on the ordering of the data we averaged over 100 different permutations of every training set. Table 1 shows the resulting prediction error (MSE) along with the size of the subset. Our method shows a similar performance as KRLS but uses fewer (sometimes far fewer) basis functions.

Large-scale real-world benchmark. Though our method is particularly tailored to online learning we show that it is also useful when dealing with large-scale data sets. To this end we chose the biggest data set available from UCI,

the data set *forest*. [1] Before we started training we set aside 81,012 randomly chosen examples to serve as independent test set. All of the remaining 500,000 examples were used to train. Since this is a rather large number (for OLS), we also considered smaller training sets of size 10,000, 50,000 and 200,000. In case of OLS we used the 'rule of the 59' [11] heuristic to restrict the search among all remaining candidates to a subset of 59 randomly drawn ones (termed OLS-59). For our approach we set RBF width $\sigma = 1/d$ (with $d = 54$ the input dimensionality), $\gamma = t \cdot 10^{-5}$ (with t being the number of training examples), $\texttt{TOL1} = 10^{-2}$ and $\texttt{TOL2} = 10^{-4}$. The generalization performance and also the CPU time will of course largely depend on the number of basis functions in the model. Hence we examine different models using an increasing number of maximum basis functions. To rule out the influence of randomness each single run was repeated 10 times. Table 2 and Fig. 1 show the achieved classification error (given as percentage of misclassified examples) on the independent test set along with the amount of variation over the different trials (given in parentheses as one standard deviation). Using the same number of basis functions m, we could achieve a classification performance that is comparable with OLS (only slightly worse). However, our approach being an online method needs far less resources (both CPU-time and memory) to achieve this result (see Fig. 1): the time needed for training is faster at nearly an order of magnitude, while the memory consumption is only $\mathcal{O}(m^2)$ as opposed to $\mathcal{O}(tm)$ when using OLS. In both cases our results are in line with the error rates achieved in the comparable experiments from [9].

Table 1. Prediction error (MSE) and number of selected basis functions (given in parentheses) for different subset selection variants.

Data set	OLS+GCV(subset)	KRLS (subset)	Our (subset)
Sinc	5.6e-4(10)	9.1e-4±1.5e-4 (14.36)	7.5e-4±3.1e-4 (11.06)
Boston	0.88 (44)	0.65±0.039 (220.65)	0.63±0.2 (59.24)
Abalone	0.35 (62)	0.35±0.014 (124.3)	0.37±0.05 (31.54)

5 Conclusion

We presented a subspace based variant of least squares SVM especially geared to online learning. It uses a novel criterion to select a subset of relevant basis functions from the full data set. Experiments indicate that our method improves upon the related KRLS algorithm by choosing a smaller subset and that it can even compete with powerful greedy forward selection; an alternative only amenable to offline learning and at considerably higher computational costs.

[1] *Forest* is a multi-class classification problem with 581,012 examples and 7 classes. As in [9] we transformed the problem into a two-class classification task: classify class 2 against the rest, which makes the resulting partitions of roughly the same size. *Forest* contains continuous as well as categorical attributes; the latter were transformed via a binary encoding so that the input dimensionality of the problem became $d = 54$. The inputs of the data were scaled to have zero mean and unit variance.

Table 2. Classification error for different sizes of the subset (given in parentheses).

Data set	OLS-59(100)	Our(100)	OLS-59(300)	Our(300)	OLS-59(500)	Our(500)
Forest-10k	22.80±0.18	24.61±2.31	21.26±0.23	22.33±0.51	20.35±0.11	21.77±0.34
Forest-50k	22.63±0.11	23.99±1.96	20.58±0.15	21.93±0.36	19.63±0.12	21.24±0.42
Forest-200k	—	24.49±1.64	—	21.83±0.36	—	21.19±0.30
Forest-500k	—	23.80±0.64	—	21.77±0.39	—	21.17±0.32

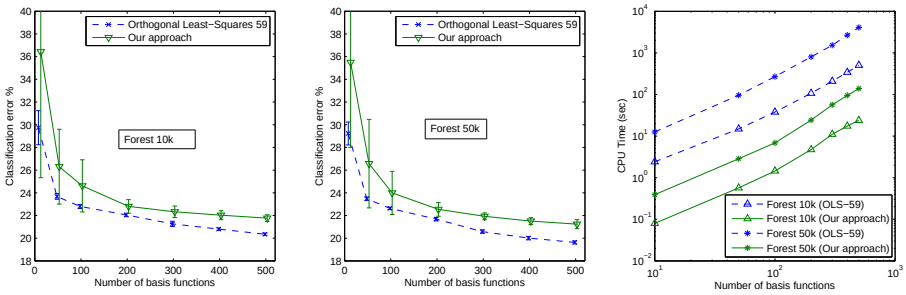

Fig. 1. Comparing our method with OLS

References

1. F. R. Bach and M. I. Jordan. Predictive low-rank decomposition for kernel methods. In *Proc. of ICML 22*, 2005.
2. L. Csató and M. Opper. Sparse representation for Gaussian process models. In *NIPS 13*, 2001.
3. C.L. Blake D.J. Newman, S. Hettich and C.J. Merz. UCI repository of machine learning databases, 1998.
4. Y. Engel, S. Mannor, and R. Meir. The kernel recursive least squares algorithm. *IEEE Trans. on Signal Processing*, 52(8):2275–2285, 2004.
5. S. Fine and K. Scheinberg. Efficient SVM training using low-rank kernel representation. *JMLR*, 2:243–264, 2001.
6. Hoegaerts L., Suykens J.A.K., Vandewalle J., and De Moor B. Subset based least squares subspace regression in RKHS. *Neurocomputing*, 63:293–323, 2005.
7. Z. Luo and G. Wahba. Hybrid adaptive splines. *J. Amer. Statist. Assoc.*, 92:107–114, 1997.
8. J. Platt. A resource-allocating network for function interpolation. *Neural Computation*, 3:213–225, 1991.
9. V. Popovici, S. Bengio, and J.-P. Thiran. Kernel matching pursuit for large datasets. *Pattern Recognition*, 38(12):2385–2390, 2005.
10. J. Quiñonero Candela and C. E. Rasmussen. A unifying view of sparse approximate Gaussian process regression. *JMLR*, 6:1935–1959, 2005.
11. A. J. Smola and P. L. Bartlett. Sparse greedy Gaussian process regression. In *NIPS 13*, 2001.
12. A. J. Smola and B. Schölkopf. Sparse greedy matrix approximation for machine learning. In *Proc. of ICML 17*, 2000.
13. C. Williams and M. Seeger. Using the nyström method to speed up kernel machines. In *NIPS 13*, 2001.