Cross-Validation Optimization for Large Scale Structured Classification Kernel Methods

Matthias W. Seeger SEEGER@TUEBINGEN.MPG.DE
Max Planck Institute for Biological Cybernetics
Spemannstr. 38, Tübingen, Germany

Editor: Zoubin Ghahramani

Abstract

We propose a highly efficient framework for penalized likelihood kernel methods applied to multi-class models with a large, structured set of classes. As opposed to many previous approaches which try to decompose the fitting problem into many smaller ones, we focus on a Newton optimization of the complete model, making use of model structure and linear conjugate gradients in order to approximate Newton search directions. Crucially, our learning method is based entirely on matrix-vector multiplication primitives with the kernel matrices and their derivatives, allowing straightforward specialization to new kernels, and focusing code optimization efforts to these primitives only.

Kernel parameters are learned automatically, by maximizing the cross-validation log likelihood in a gradient-based way, and predictive probabilities are estimated. We demonstrate our approach on large scale text classification tasks with hierarchical structure on thousands of classes, achieving state-of-the-art results in an order of magnitude less time than previous work.

Parts of this work appeared in the conference paper (Seeger, 2007).

Keywords: multi-way classification, kernel logistic regression, hierarchical classification, cross validation optimization, newton-raphson optimization

1. Introduction

In recent years, Machine Learning researchers started to address problems with kernel machines which require models with a large number of dependent variables, and whose fitting demand training samples with very many cases. For example, for multi-way classification models with a hierarchically structured label space (Cai and Hofmann, 2004), modern applications call for predictions on thousands of classes, and very large datasets become available. However, if n and C denote dataset size and number of classes respectively, nonparametric kernel methods like *support vector machines* (SVMs) or *Gaussian processes* (GPs) typically scale superlinearly in $n\,C$, if dependencies between the latent class functions are represented properly.

Furthermore, most large scale kernel methods proposed so far refrain from solving the problem of learning hyperparameters (kernel or loss function parameters), also known as "learning the kernels". The user has to run cross-validation schemes essentially "by hand", which is not suitable for learning more than a few hyperparameters. However, many models for modern applications come with a large number of hyperparameters (for example to represent dependencies through "mixing" as in independent components analysis), and adjusting them through optimization requires gradients.

We propose a general framework for learning in probabilistic kernel classification models. While the models treated here are not novel, a major feature of our approach is the high computational efficiency with which the primary fitting (for fixed hyperparameters) is done. For example, our framework applied to hierarchical classification with hundreds of classes and thousands of datapoints requires a few minutes for fitting. The central idea is to step back from what seems to be the dominating approach in Machine Learning at the moment, namely to solve a large convex optimization problem by iteratively solving very many small ones. A popular approach for these small steps is to minimize the criterion w.r.t. a few variables only, keeping the other ones fixed, and many variations of this theme have been proposed. In this paper, we focus on the opposite approach of trying to find directions which lead to fast descent, no matter how many of the variables are involved. This is essentially Newton's method, and one aspect of our work is to find approximate Newton directions very efficiently, making use of model structure and linear conjugate gradients in order to reduce the computation to standard linear algebra primitives on large contiguous chunks of memory. Interestingly, such global approaches are generally favoured in the Optimization community for problems (such as kernel methods fitting) which cannot be decomposed naturally into parts. While other gradient-based optimizers such as scaled conjugate gradients could be used as well, they require more fine-tuning (for example, preconditioning) to the specific problem they are applied to, while Newton's method is closer to a "black box" technique and can be transferred to novel situations without many changes.

For multi-way classification, our primary fitting method scales linearly in C, and depends on n mainly via a fixed number of *matrix-vector multiplications* (MVM) with $n \times n$ kernel matrices. In many situations, these MVM primitives can be computed very efficiently, often without having to store the kernel matrices themselves.

We also show how to choose hyperparameters *automatically* by maximizing the cross-validation log likelihood, making use of our primary fitting technology as inner loop in order to compute the CV criterion and its gradient. It is important to note that our hyperparameter learning method works by gradient-based optimization, where the dominating part of the gradient computation does not scale with the number of hyperparameters at all[1]. The gradient computation also requires a number of MVMs with derivatives of kernel matrices, which can be reduced to kernel MVMs for many frequently used kernels (see Section 7.3). Therefore, our approach can be used to learn a large number of hyperparameters without user interaction.

We apply our framework to hierarchical classification with many classes. The hierarchy is represented through an ANOVA setup. While the C latent class functions are fully dependent *a priori*, the scaling of our method stays close to what unstructured (flat) classification with C classes would require. We test our framework on the same tasks treated by Cai and Hofmann (2004), achieving comparable results in at least an order of magnitude less time.

Our proposal to use approximate Newton methods is not novel as such. The Newton method, or a variant of it called *Fisher scoring*, is the standard approach for fitting general-

1. Such scaling behaviour is fairly standard in Gaussian process marginal likelihood maximization techniques (Williams and Barber, 1998), but has only recently been brought to attention in the SVM community (Keerthi et al., 2007).

ized linear models in Statistics (Green and Silverman, 1994, McCullach and Nelder, 1983), at least if parametric models are fitted to moderately sized samples. Our primary fitting method for flat multi-way classification (see Section 2) appeared in (Williams and Barber, 1998). However, we demonstrate the usefulness of this principle on a much larger scale, showing how model structure can (and has to) be exploited in this context. Furthermore, we demonstrate how the secondary task of hyperparameter learning can be reduced to the same underlying primitives.

The structure of the paper is as follows. Our model and method of parameter fitting is given in Section 2. An extension to hierarchical classification is provided in Section 3, and in Section 4 we give our automatic hyperparameter learning procedure. Essential computational details are discussed in Section 5. Experimental results on a very large hierarchical text classification and several standard machine learning problems are given in Section 6. We close with a discussion in Section 7, relating our global direction approach to popular block coordinate descent techniques in Section 7.2, and pointing out future work in Section 7.4.

Optimized C++ software for our framework is available as part of the *LHOTSE* toolbox for adaptive statistical models, which is freely available for non-commercial purposes[2]. The implementation contains the linear kernel case used in Section 6.1 (see Appendix D.3), as well as a generic representation described in Appendix D.1, with which the experiments in Section 6.2, Section 6.3 have been done. It is fairly simple to include new kernels or (approximate) kernel MVM implementations.

2. Penalized Multiple Logistic Regression

In this section, we introduce our framework on a multi-way classification model with C classes, where structure between classes is not modelled. We refer to this setup as *flat classification*, in that the label set is flat (unstructured).

In general, our framework is applicable to models of the form depicted in Figure 1. A set of latent (unobserved) functions $u_c(\cdot)$ is fitted to observed data by penalized likelihood maximization. For many models, the *penalisation term* (also called *regulariser*) corresponds to the logarithm of a prior density over the $u_c(\cdot)$. This *primary fitting* step corresponds to a convex optimization problem over finitely many variables. Structure in such models is represented either as couplings in the log likelihood function, or in the penalisation (or log prior) term. The latter can be realized through the linear mixing of *a priori* independent functions $\breve{u}_p(\cdot)$, in other words the penaliser over the latter decouples w.r.t. p (our main example of such mixing is hierarchical classification, developed in Section 3).

We now apply this general framework to flat classification, where $y \in \{1, \dots, C\}$ is to be predicted from $\boldsymbol{x} \in \mathcal{X}$, given some i.i.d. data $D = \{(\boldsymbol{x}_i, y_i) \,|\, i = 1, \dots, n\}$. Our notation convention for vectors and matrices is detailed in Appendix A, where we also collect all major notational definitions in a table. We code y_i as $\boldsymbol{y}_i \in \{0,1\}^C$, $\mathbf{1}^T \boldsymbol{y}_i = 1$ (zero-one coding)[3]. We employ the *multiple logistic regression model*, consisting of C latent class functions $u_c(\cdot)$ feeding into the multiple logistic (or softmax) likelihood $P(y_{ic} = 1|\boldsymbol{x}_i, \boldsymbol{u}_i(\cdot)) = e^{u_c(\boldsymbol{x}_i)}/(\sum_{c'} e^{u_{c'}(\boldsymbol{x}_i)})$.

2. Available at `www.kyb.tuebingen.mpg.de/bs/people/seeger/lhotse/`.
3. We switch between the formats y_i, $\boldsymbol{y}_i$. Note that y_{ic} denotes a component in $\boldsymbol{y}_i = (y_{ic})_c$.

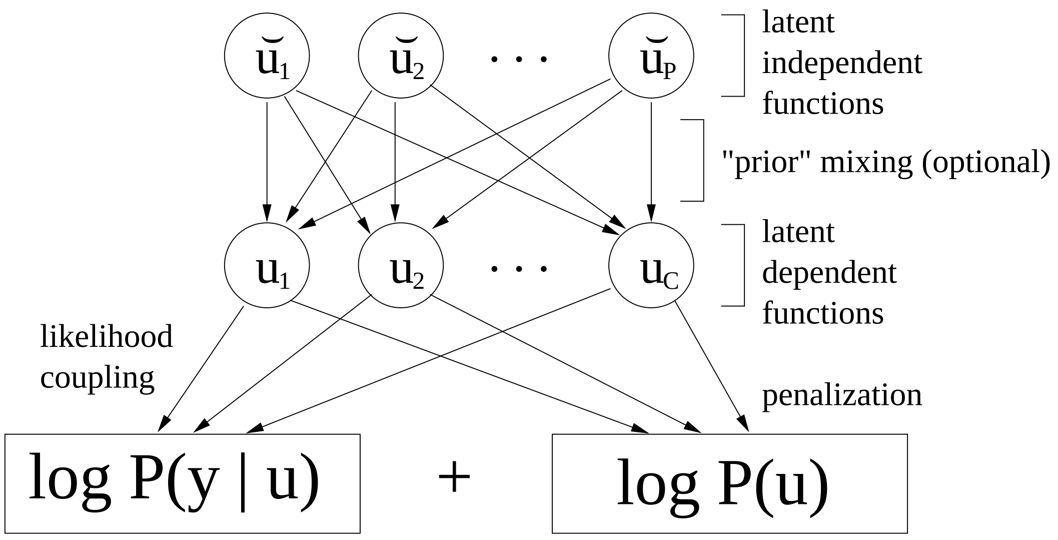

Figure 1: Structure of penalized likelihood optimization.

We write $u_c(\cdot) = f_c(\cdot) + b_c$ for intercept (or bias) parameters $b_c \in \mathbb{R}$ and functions $f_c(\cdot)$ living in a reproducing kernel Hilbert space (RKHS) with kernel $K^{(c)} = K^{(c)}(\cdot,\cdot)$ (Schölkopf and Smola, 2002), and consider the *penalized negative log likelihood*

$$\Phi = -\sum_{i=1}^{n} \log P(\boldsymbol{y}_i|\boldsymbol{u}_i) + (1/2)\sum_{c=1}^{C} \|f_c(\cdot)\|_c^2 + (1/2)\sigma^{-2}\|\boldsymbol{b}\|^2, \quad \boldsymbol{u}_i = (u_c(\boldsymbol{x}_i))_c \in \mathbb{R}^C,$$

which we minimize for primary fitting. Here, $\|\cdot\|_c$ is the RKHS norm for kernel $K^{(c)}$. The idea is that deviations in f_c from desired functional properties encoded in $K^{(c)}$ are penalized by a large $\|f_c(\cdot)\|_c^2$. For example, for the Gaussian kernel (8), non-smooth f_c are penalized, and for the linear kernel (Appendix D.3), $\|f_c(\cdot)\|_c^2$ is the squared norm of the weight vector. Details on penalized likelihood kernel methods and RKHS penalisation can be found in (Green and Silverman, 1994, Schölkopf and Smola, 2002).

The model can also be understood in a Bayesian context, where the penalisation terms come from zero mean Gaussian process priors on the functions $f_c(\cdot)$, and $\boldsymbol{b}$ has a zero mean Gaussian prior with variance σ^2. From this viewpoint, we do a *maximum a-posteriori* (MAP) approximation here, without however taking covariances into account properly (which would be much more expensive to do). Details on Gaussian processes for Machine Learning can be found in (Seeger, 2004, Rasmussen and Williams, 2006).

Since the likelihood depends on the $f_c(\cdot)$ only through the values $f_c(\boldsymbol{x}_i)$ at the datapoints, every minimizer of Φ must be a kernel expansion: $f_c(\cdot) = \sum_i \alpha_{ic} K^{(c)}(\cdot, \boldsymbol{x}_i)$. This fact is known as *representer theorem* (Green and Silverman, 1994, Wahba, 1990). Plugging this in, the regulariser becomes $(1/2)\boldsymbol{\alpha}^T\boldsymbol{K}\boldsymbol{\alpha} + (1/2)\sigma^{-2}\|\boldsymbol{b}\|^2$, where $\boldsymbol{K}^{(c)} = (K^{(c)}(\boldsymbol{x}_i, \boldsymbol{x}_j))_{i,j} \in \mathbb{R}^{n,n}$, and $\boldsymbol{K} = \mathrm{diag}(\boldsymbol{K}^{(c)})_c$ is block-diagonal. The kernels $K^{(c)}$ can in general be different, although sharing kernels among classes can lead to computational savings, in that some of the blocks $\boldsymbol{K}^{(c)}$ are identical. Our implementation of block sharing is described in Appendix D.1.

We show in Section 5.1.1 that the b_c may be eliminated as $\boldsymbol{b} = \sigma^2(\boldsymbol{I} \otimes \boldsymbol{1}^T)\boldsymbol{\alpha}$. Thus, if $\tilde{\boldsymbol{K}} = \boldsymbol{K} + \sigma^2(\boldsymbol{I} \otimes \boldsymbol{1})(\boldsymbol{I} \otimes \boldsymbol{1}^T)$, then our criterion Φ becomes

$$\Phi = \Phi_{lh} + \frac{1}{2}\boldsymbol{\alpha}^T\tilde{\boldsymbol{K}}\boldsymbol{\alpha}, \quad \Phi_{lh} = -\boldsymbol{y}^T\boldsymbol{u} + \boldsymbol{1}^T\boldsymbol{l}, \quad l_i = \log \boldsymbol{1}^T \exp(\boldsymbol{u}_i), \quad \boldsymbol{u} = \tilde{\boldsymbol{K}}\boldsymbol{\alpha}. \tag{1}$$

Φ is strictly convex in $\boldsymbol{\alpha}$, being a sum of linear, quadratic, and *logsumexp* terms of the form $\log \boldsymbol{1}^T \exp(\boldsymbol{u}_i)$ (Boyd and Vandenberghe, 2002), so it has a unique minimum point $\hat{\boldsymbol{\alpha}}$. The corresponding kernel expansions are

$$\hat{u}_c(\cdot) = \sum_i \hat{\alpha}_{ic}(K^{(c)}(\cdot, \boldsymbol{x}_i) + \sigma^2).$$

Estimates of the conditional probability on test points $\boldsymbol{x}_*$ are obtained by plugging $\hat{u}_c(\boldsymbol{x}_*)$ into the likelihood. These estimates are asymptotically consistent, although better finite sample estimates could probably be obtained by a more Bayesian treatment.

We note that this setup is related to the multi-class SVM (Crammer and Singer, 2001), where $-\log P(y_i|\boldsymbol{u}_i)$ is replaced by the margin loss $-u_{y_i}(\boldsymbol{x}_i) + \max_c\{u_c(\boldsymbol{x}_i) + 1 - \delta_{c,y_i}\}$. Here, $\delta_{a,b} = \mathrm{I}_{\{a=b\}}$. The negative log multiple logistic likelihood has similar properties, but is smooth as a function of $\boldsymbol{u}$, and the primary fitting of $\boldsymbol{\alpha}$ does not require constrained convex optimization. Furthermore, universal consistency for estimates of $P(y_*|\boldsymbol{x}_*)$ can be established for the multiple logistic loss, but fails to hold for the SVM variant (Bartlett and Tewari, 2004).

We will minimize Φ using the *Newton-Raphson* (NR) algorithm. The computation of Newton search directions requires solving a system with the Hessian and the gradient of Φ, which we will do approximately using the *linear conjugate gradients* (LCG) algorithm. This can be done without fully computing, storing, or inverting the Hessian, all of which would not be possible for large nC. In fact, the task is reduced to computing $k_1(k_2+2)$ MVMs with $\boldsymbol{K}$, where k_1 is the number of NR iterations, k_2 the number of LCG steps for computing each Newton direction. Since NR is a second-order convergent method, k_1 is generally small. k_2 determines the quality of each Newton direction, and again, fairly small values seem sufficient (see Section 6.1). Details are provided in Section 5.1.

Finally, some readers may wonder why we favour the NR algorithm here, which in practice can be fairly complicated to implement, while we could do a simpler gradient-based optimization of Φ w.r.t. $\boldsymbol{\alpha}$, for example by scaled (non-linear) conjugate gradients (SCG). The problem is that on tasks of the size we want to address, non-invariant methods such as SCG tend to fail completely if not properly preconditioned, and we experienced exactly that in preliminary experiments. In contrast to that, NR is invariant to the choice of optimization variables, so does not have to be preconditioned. It is by far the preferred method in the Optimization literature (Bertsekas, 1999, Boyd and Vandenberghe, 2002), and many ideas for preconditioning or Quasi-Newton try to approximate the NR directions. We think that a proper SCG implementation can be at least as efficient as NR, but needs fine-tuning to the specific problem, which in the case of hierarchical classification (discussed next) is already quite difficult. More details on this point are given in Section 5.4 and also Section 7.2.

3. Hierarchical Classification

So far we dealt with flat classification, the classes being independent *a priori*, with block-diagonal kernel matrix $\boldsymbol{K}$. However, if the label set has a known structure[4], we can benefit from representing it in the model. Here we focus on *hierarchical classification*, the label set $\{1, \dots, C\}$ being the leaf nodes of a tree. Classes with lower common ancestor should be more closely related. In this section, we propose a model for this setup and show how it can be dealt with in our framework with minor modifications and reasonable extra cost.

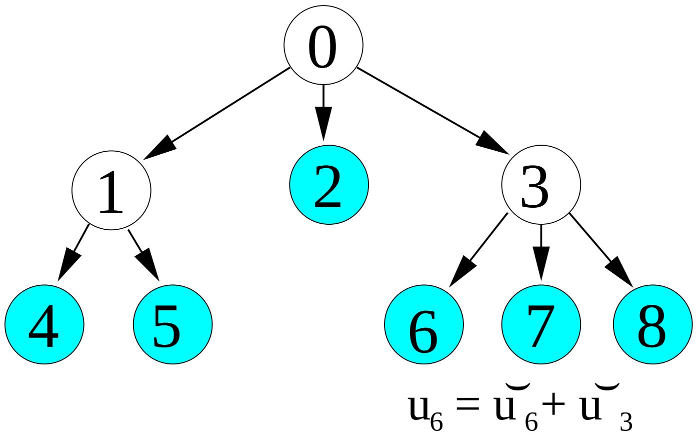

Figure 2: Example of a tree-structured target space, where labels correspond to leaf nodes (shaded).

In flat classification, the latent class functions $u_c(\cdot)$ are modelled as *a priori* independent, in that the penaliser (or the log prior in the GP view) is a sum of individual terms for each c, without interaction terms. *Analysis of variance* (ANOVA) models go beyond such independent designs, they have previously been applied to text classification by Cai and Hofmann (2004), see also Shahbaba and Neal (2007). Let $\{0, \dots, P\}$ be the nodes of the tree, 0 being the root, and the numbers are assigned breadth first ($1, 2, \dots$ are the root's children). The tree is determined by P and n_p, $p = 0, \dots, P$, the number of children of node p. Let L be the set of leaf nodes, $|L| = C$. Assign a *pair* of latent functions u_p, $\breve{u}_p$ to each node, except the root. The $\breve{u}_p$ are assumed *a priori* independent, as in flat classification. u_p is the sum of $\breve{u}_{p'}$, where p' is running over the nodes (including p) on the path from the root to p. An example is given in Figure 2. The class functions to be fed into the classification likelihood are the $u_{L(c)}$ of the leafs. This setup represents similarities according to the hierarchy. For example, if leafs $L(c)$, $L(c')$ have the common parent p, then $u_{L(c)} = u_p + \breve{u}_{L(c)}$, $u_{L(c')} = u_p + \breve{u}_{L(c')}$, so the class functions *share* the effect u_p. Since regularisation forces all independent effects $\breve{u}_{p'}$ to be smooth, the classes c, c' are urged to behave similarly *a priori*.

Let $\boldsymbol{u} = (u_p(\boldsymbol{x}_i))_{i,p}$, $\breve{\boldsymbol{u}} = (\breve{u}_p(\boldsymbol{x}_i))_{i,p} \in \mathbb{R}^{nP}$. The vectors are linearly related as $\boldsymbol{u} = (\boldsymbol{\Phi} \otimes \boldsymbol{I})\breve{\boldsymbol{u}}$, $\boldsymbol{\Phi} \in \{0,1\}^{P,P}$, a special case of the mixing of Figure 1. Importantly, $\boldsymbol{\Phi}$ has a simple

4. Learning an unknown label set structure may be achieved by expectation maximization techniques, but this is subject to future work.

structure which allows MVM with $\boldsymbol{\Phi}$ or $\boldsymbol{\Phi}^T$ to be computed easily in $O(P)$, without having to compute or store $\boldsymbol{\Phi}$ explicitly. Let $cs_p = \sum_{p'<p} n_{p'}$, and define $\boldsymbol{\Phi}_p \in \mathbb{R}^{d,d}$, $d = cs_p + n_p$, to be the upper left block of $\boldsymbol{\Phi}$, so that $\boldsymbol{\Phi} = \boldsymbol{\Phi}_P$. If p is a leaf node, then $\boldsymbol{\Phi}_p = \boldsymbol{\Phi}_{p-1}$. Otherwise, $\boldsymbol{\Phi}_p$ is obtained from $\boldsymbol{\Phi}_{p-1}$ by attaching rows $(\boldsymbol{\delta}_p^T \boldsymbol{\Phi}_{p-1}, \boldsymbol{\delta}_j^T)$, $j = 1, \dots, n_p$, where $\boldsymbol{\delta}_p^T \boldsymbol{\Phi}_{p-1}$ is the p-th row of $\boldsymbol{\Phi}_{p-1}$. This is because $u_{cs_p+j} = u_p + \breve{u}_{cs_p+j}$ for the functions of the children of p. Formally,

$$\boldsymbol{\Phi}_p = \begin{pmatrix} \boldsymbol{\Phi}_{p-1} & \boldsymbol{0} \\ \boldsymbol{1}\boldsymbol{\delta}_p^T \boldsymbol{\Phi}_{p-1} & \boldsymbol{I} \end{pmatrix},$$

where the lower right $\boldsymbol{I} \in \mathbb{R}^{n_p,n_p}$. Note that $\boldsymbol{\Phi}$ is lower triangular with $\operatorname{diag} \boldsymbol{\Phi} = \boldsymbol{I}$. This recursive definition directly implies simple methods for computing $\boldsymbol{v} \mapsto \boldsymbol{\Phi}\boldsymbol{v}$ and $\boldsymbol{v} \mapsto \boldsymbol{\Phi}^T \boldsymbol{v}$.

Under the hierarchical model, the class functions $u_{L(c)}$ are strongly dependent *a priori*. Representing this prior coupling in our framework amounts to simply plugging in the implied kernel matrix

$$\boldsymbol{K} = (\boldsymbol{\Phi}_{L,\cdot} \otimes \boldsymbol{I}) \breve{\boldsymbol{K}} (\boldsymbol{\Phi}_{L,\cdot}^T \otimes \boldsymbol{I}), \tag{2}$$

into the flat classification model of Section 2. Here, the inner $\breve{\boldsymbol{K}}$ is block-diagonal, while in the flat model, $\boldsymbol{K}$ itself had this property. In the hierarchical case, $\boldsymbol{K}$ is not sparse and certainly not block-diagonal, but we are still able to compute kernel MVMs efficiently: pre- and postmultiplying by $\boldsymbol{\Phi}$ is very cheap, and $\breve{\boldsymbol{K}}$ is block-diagonal just as in the flat case.

In fact, the step from flat to hierarchical classification requires minor modifications of existing code only. If code for representing a block-diagonal $\boldsymbol{K}$ is available, we can use it to represent the inner $\breve{\boldsymbol{K}}$, just replacing C by P. This simplicity carries through to the hyperparameter learning method (see Section 4). The cost of a kernel MVM is increased[5] by a factor $P/C < 2$, which in most hierarchies in practice is close to 1.

However, it would be wrong to claim that hierarchical classification in general comes as cheap as flat classification. In fact, primary fitting becomes more costly, precisely because there is more coupling between the variables. In the flat case, the Hessian of Φ (1) is close to block-diagonal. The LCG algorithm to compute Newton directions converges quickly, because it nearly decomposes into C independent ones, and fewer NR steps are required. In the hierarchical case, this "near-decomposition" does not hold, and both LCG and NR need more iterations to attain the same accuracy, although each LCG step comes at about the same cost as in the flat case.

In numerical mathematics, much work has been done to approximately decouple linear systems by *preconditioning*. In some of these strategies, knowledge about the structure of the system matrix (in our case: the hierarchy) can be used to drive preconditioning. An important point for future research is to find a good preconditioning strategy for the system (6). However, in all our experiments so far the fitting of the hierarchical model took less than twice the time required for the flat model on the same task.

4. Hyperparameter Learning

Our framework comes with an automatic method for setting free hyperparameters $\boldsymbol{h}$, by gradient-based maximization of the *cross-validation* (CV) log likelihood. Our primary fitting

5. Nodes with a single child only can be pruned from the hierarchy. Note that our formalism does not require all leaf nodes to have the same depth.

method of Section 2 is used here as principal subroutine. Such a setup is commonplace in Bayesian Statistics, where (marginal) inference is typically employed as subroutine in parameter learning.

Recall that primary fitting works by minimizing Φ (1) w.r.t. $\boldsymbol{\alpha}$. Let $\{I_k\}$ be a partition of the dataset range $\{1,\dots,n\}$, with $J_k = \{1,\dots,n\} \setminus I_k$, and let

$$\Phi_{J_k} = \boldsymbol{u}_{[J_k]}^T((1/2)\boldsymbol{\alpha}_{[J_k]} - \boldsymbol{y}_{J_k}) + \boldsymbol{1}^T\boldsymbol{l}_{[J_k]}$$

be the negative log likelihood of the subset J_k of the data. Here, $\boldsymbol{u}_{[J_k]} = \tilde{\boldsymbol{K}}_{J_k}\boldsymbol{\alpha}_{[J_k]}$. The $\boldsymbol{\alpha}_{[J_k]}$ are independent variables, *not* part of a common[6] $\boldsymbol{\alpha}$. The cross-validation criterion is

$$\Psi = \sum_k \Psi_{I_k}, \quad \Psi_{I_k} = -\boldsymbol{y}_{I_k}^T\boldsymbol{u}_{[I_k]} + \boldsymbol{1}^T\boldsymbol{l}_{[I_k]}, \quad \boldsymbol{u}_{[I_k]} = \tilde{\boldsymbol{K}}_{I_k,J_k}\boldsymbol{\alpha}_{[J_k]}, \tag{3}$$

where $\boldsymbol{\alpha}_{[J_k]}$ is the minimizer of Φ_{J_k}. Since for each k, we fit and evaluate the likelihood on disjoint parts of $\boldsymbol{y}$, Ψ is an unbiased estimator of the true negative expected log likelihood.

In order to adjust $\boldsymbol{h}$, we pick a fixed partition at random, then do gradient-based minimization of Ψ w.r.t. $\boldsymbol{h}$. To this end, we maintain the set $\{\boldsymbol{\alpha}_{[J_k]}\}$ of primary variables, and iterate between re-fitting those for each fold k, and computing Ψ and $\nabla_{\boldsymbol{h}}\Psi$. The gradient can be determined analytically, using a computation which is equivalent to the Newton direction computations for $\boldsymbol{\alpha}_{[J_k]}$, meaning that the same code can be used. Details are given in Section 5.2. Note that Ψ is not a convex objective.

As for computational complexity, suppose there are q folds. The update of the $\boldsymbol{\alpha}_{[J_k]}$ requires q primary fitting applications, but since they are initialized with the previous values $\boldsymbol{\alpha}_{[J_k]}$, they do converge very rapidly, especially during later iterations. Computing Ψ based on the $\boldsymbol{\alpha}_{[J_k]}$ comes basically for free. The gradient computation decomposes into two parts: accumulation, and kernel derivative MVMs. The accumulation part requires solving q systems of size $((q-1)/q)n\,C$, thus $q\,k_3$ kernel MVMs on the $\tilde{\boldsymbol{K}}_{J_k}$ if linear conjugate gradients (LCG) is used, k_3 being the number of LCG steps. We also need two buffer matrices $\boldsymbol{E}$, $\boldsymbol{F}$ of $q\,n\,C$ elements each. Note that the accumulation step is *independent* of the number of hyperparameters. The second part consists of q kernel derivative MVMs for each independent component of $\boldsymbol{h}$. This second part is much simpler than the accumulation one, consisting entirely of large matrix operations, which can be run very efficiently using specialized numerical linear algebra code. The method for computing Ψ and $\nabla_{\boldsymbol{h}}\Psi$ can be plugged into a custom gradient-based optimizer, such as Quasi-Newton or conjugate gradients, in order to learn $\boldsymbol{h}$.

As shown in Section 5.3, the extension of hyperparameter learning to the hierarchical case of Section 3 is done by wrapping the accumulation part with $\boldsymbol{\Phi}$ MVMs, the coding and additional memory effort being minimal.

We finally note from our findings in practice (see Section 6.3) that on large tasks, our automatic method can require some fine-tuning. This is due to the delicate dependencies between the different approximations used. The accuracy of Ψ and $\nabla_{\boldsymbol{h}}\Psi$ depends on how accurate the inner NR optimizations for $\boldsymbol{\alpha}_{[J_k]}$ turn out, and the latter depend on how many iterations of LCG are done in order to compute search directions. Fortunately, Φ_{J_k} and its gradient w.r.t. $\boldsymbol{u}_{[J_k]}$ can be computed exactly in order to assess inner optimization

6. Which is why they are not referred to as $\boldsymbol{\alpha}_{J_k}$.

convergence, so we do at least know when things go wrong. In our implementation, we deem an evaluation of Ψ and $\nabla_{\boldsymbol{h}}\Psi$ usable if the average of $\|\nabla_{\boldsymbol{u}_{[J_k]}}\Phi_{J_k}\|$ over folds is below a threshold, which depends on the problem and on time constraints. A failed evaluation leads to a right bracket there for the outer optimization line search, in that step sizes beyond the failed one are not accessed. We can now tune the basic running time parameters k_1, k_2 so that Ψ evaluations do not fail too often. In this context, it is important to regard the $\{\boldsymbol{\alpha}_{[J_k]}\}$ as an *inner state* alongside the hyperparameter vector $\boldsymbol{h}$. Although inner optimizations are convex, for large problems and reasonable k_1, k_2, successive minima are attained only when we start from the previous best inner state. This is true especially during later stages, where for certain problems (see Section 6.3) $\boldsymbol{h}$ attends "extreme" values and the inner optimizations become quite hard[7]. Therefore, the inner state used to initialize a given Ψ evaluation is the final one for the last recent *successful* evaluation[8]. Inner states attained during failed evaluations are discarded.

5. Computational Details

In this section, we provide details for the material above. The techniques given here do characterize our framework, they are novel in this combination, and some of them may be useful in other contexts as well. More specific details of our implementation can be found in Appendix D.

5.1 Details for Flat Classification

In this section, we provide details for the primary fitting optimization in the case of flat multi-way classification, introduced in Section 2. Note that this fitting method appeared in (Williams and Barber, 1998) in the context of approximate Gaussian process inference, although some fairly essential ideas here are novel to our knowledge (symmetrisation of Newton system, pair optimization line search, numerical stability considerations).

Recall that we want to minimize the strictly convex criterion Φ (1) w.r.t. $\boldsymbol{\alpha}$, using the Newton-Raphson (NR) method. Modern variants of this algorithm iterate line searches along the *Newton directions* $-\boldsymbol{H}^{-1}\boldsymbol{g}$, where $\boldsymbol{g}$, $\boldsymbol{H}$ are gradient and Hessian of Φ at the current $\boldsymbol{\alpha}$. We will start with the Newton direction computation in Section 5.1.1, commenting on the line searches afterwards in Section 5.1.2 (it turns out that it basically comes for free). An overview of the fitting algorithm is given in Section 5.1.3.

5.1.1 Computing the Newton Direction

Recall Φ and related variables from (1). Let $\pi_{ic} = P(y_{ic} = 1|\boldsymbol{u}_i)$, *i.e.* $\boldsymbol{\pi} = \exp(\boldsymbol{u} - \boldsymbol{1} \otimes \boldsymbol{l})$, and recall that Φ_{lh} is the likelihood part in Φ. Now,

$$\boldsymbol{g} := \nabla\Phi_{lh} = \boldsymbol{\pi} - \boldsymbol{y},\ \boldsymbol{W} := \nabla\nabla\Phi_{lh} = \boldsymbol{D} - \boldsymbol{D}\boldsymbol{P}_{cls}\boldsymbol{D},\ \boldsymbol{P}_{cls} = (\boldsymbol{1} \otimes \boldsymbol{I})(\boldsymbol{1}^T \otimes \boldsymbol{I}).$$

7. Although inner optimizations are convex, speed of convergence of NR depends strongly on the value of $\boldsymbol{h}$. For "extreme" values, the Newton direction computation by LCG is harder, and search directions can become large in early NR iterations. The latter may be because we work in $\boldsymbol{u}$ rather than $\boldsymbol{\alpha}$ space, but only the former is really feasible.
8. Within outer line searches, we use $\{\boldsymbol{\alpha}_{[J_k]}\}$ from the last recent successful evaluation *to the left* (along the search direction).

Here, $\boldsymbol{D} = \operatorname{diag}\boldsymbol{\pi}$, and gradient and Hessian are taken w.r.t. $\boldsymbol{u}$ (*not* w.r.t. $\boldsymbol{\alpha}$). Our convention for nC vectors and matrices and the use of $\otimes$ is explained in Appendix A. The form of $\boldsymbol{W}$ can be understood by noting that $\boldsymbol{W}$ is block-diagonal in a *different* ordering, which uses c (classes) as inner and i (datapoints) as outer index, then switching to our standard ordering.

It is easy to compute gradient and Hessian of Φ w.r.t. $\boldsymbol{\alpha}$, $\boldsymbol{b}$. A full (classical) Newton step is given by the system

$$\begin{aligned}&(\boldsymbol{I} + \boldsymbol{W}\boldsymbol{K})\boldsymbol{\alpha}' + \boldsymbol{W}(\boldsymbol{I}\otimes\boldsymbol{1})\boldsymbol{b}' = \boldsymbol{W}\boldsymbol{u} - \boldsymbol{g},\\ &(\boldsymbol{I}\otimes\boldsymbol{1}^T)\boldsymbol{W}\boldsymbol{K}\boldsymbol{\alpha}' + (\boldsymbol{I}\otimes\boldsymbol{1}^T)\boldsymbol{W}(\boldsymbol{I}\otimes\boldsymbol{1})\boldsymbol{b}' + \sigma^{-2}\boldsymbol{b}' = (\boldsymbol{I}\otimes\boldsymbol{1}^T)(\boldsymbol{W}\boldsymbol{u} - \boldsymbol{g}),\end{aligned}$$

and the Newton search direction is obtained as the difference $\boldsymbol{\alpha}' - \boldsymbol{\alpha}$, $\boldsymbol{b}' - \boldsymbol{b}$. Subtracting $(\boldsymbol{I}\otimes\boldsymbol{1}^T)$ times the first from the second, we obtain $\boldsymbol{b}' = \sigma^2(\boldsymbol{I}\otimes\boldsymbol{1}^T)\boldsymbol{\alpha}'$, and plugging this into the first equation, we have

$$\left(\boldsymbol{I} + \boldsymbol{W}\left(\boldsymbol{K} + \sigma^2\boldsymbol{P}_{data}\right)\right)\boldsymbol{\alpha}' = \boldsymbol{W}\boldsymbol{u} - \boldsymbol{g}, \quad \boldsymbol{P}_{data} = (\boldsymbol{I}\otimes\boldsymbol{1})(\boldsymbol{I}\otimes\boldsymbol{1}^T). \tag{4}$$

Note that $\boldsymbol{P}_{data}\boldsymbol{a} = (\sum_{i'}\boldsymbol{a}_{i'})_i$, which does the same as $\boldsymbol{P}_{cls}$, but on index i rather than c. We denote

$$\tilde{\boldsymbol{K}} = \boldsymbol{K} + \sigma^2\boldsymbol{P}_{data},$$

noting that this corresponds to $\tilde{\boldsymbol{K}}^{(c)} = \boldsymbol{K}^{(c)} + \sigma^2\boldsymbol{1}\boldsymbol{1}^T$. The correct way of incorporating intercept parameters is to add the constant σ^2 to the kernels, then to obtain $b_c = \sigma^2\sum_i \alpha_{ic}$. This is the meaning of "eliminating $\boldsymbol{b}$" in Section 2. While we could optimize σ^2 as a hyperparameter, we consider it fixed and given for simplicity[9]. In the sequel, we consider $\boldsymbol{b}$ being eliminated from the model by replacing $\boldsymbol{K}\to\tilde{\boldsymbol{K}}$ everywhere. We have $\boldsymbol{u} = \tilde{\boldsymbol{K}}\boldsymbol{\alpha}$.

We can solve the system (4) exactly if we can tolerate a scaling of $O(n^3\,C)$ and $O(n^2\,C)$ memory. Note that this scaling is linear rather than cubic in C. The exact solution is derived in Appendix C. It is efficient for moderate n, and generally useful for code debugging, and is supported by our implementation. In the remainder of this section, we focus on approximate computations.

Although we could solve the system using a biconjugate gradients solver, we can do much better by transforming it into symmetric positive definite form. First, note that $\boldsymbol{W}$ is positive semidefinite, but singular. This can be seen by noting that the parameterization of our likelihood in terms of $\boldsymbol{u}_i$ is overcomplete, *i.e.* $\boldsymbol{u}_i + \kappa\boldsymbol{1}$ gives the same likelihood values for all κ. We could fix one of the $\boldsymbol{u}_i$ components, which would however lead to subtle dependencies between the remaining $C-1$ functions $u_c(\cdot)$. In order to justify our *a priori* independent treatment of these functions, we have to retain the overcomplete likelihood. The nullspace $\ker\boldsymbol{W}$ is given by $\{(\boldsymbol{d})_c \,|\, \boldsymbol{d}\in\mathbb{R}^n\}$ and has dimension n. This can be seen by noting that $\boldsymbol{W}\boldsymbol{a} = \boldsymbol{0}$ iff $\boldsymbol{a} = (\bar{\boldsymbol{a}})_c$, $\bar{\boldsymbol{a}} = \sum_{c'}\boldsymbol{a}^{(c')}$. $\boldsymbol{W}$ has rank $n(C-1)$. We have $\boldsymbol{a}\in\operatorname{ran}\boldsymbol{W}$ iff $\sum_c \boldsymbol{a}^{(c)} = (\boldsymbol{1}^T\otimes\boldsymbol{I})\boldsymbol{a} = \boldsymbol{0}$ (recall that $\ker\boldsymbol{W}$ and $\operatorname{ran}\boldsymbol{W}$ are orthogonal, and their direct sum is $\mathbb{R}^{nC}$). From (4) we see that $\boldsymbol{\alpha}' + \boldsymbol{g}$ lies in $\operatorname{ran}\boldsymbol{W}$. Note that $\sum_c\boldsymbol{g}^{(c)} = \sum_c(\boldsymbol{\pi}^{(c)} - \boldsymbol{y}^{(c)}) = 1 - 1 = 0$, therefore $\boldsymbol{g}\in\operatorname{ran}\boldsymbol{W}$, thus $\boldsymbol{\alpha}'\in\operatorname{ran}\boldsymbol{W}$. We see that the dual coefficients must fulfil the constraint $\boldsymbol{\alpha}\in\operatorname{ran}\boldsymbol{W}$. Note that $\operatorname{ran}\boldsymbol{W}$ is in

9. In our experience so far, a good value of σ^2 is fairly robust across different tasks for the same problem, but may differ strongly between different problems. It can be chosen based on some initial experiments.

fact independent of $\boldsymbol{D}$. Whatever starting value is used for $\boldsymbol{\alpha}$, it should be projected onto $\operatorname{ran}\boldsymbol{W}$, which is done by subtracting $C^{-1}\boldsymbol{P}_{cls}\boldsymbol{\alpha}$. The NR updates then make sure that the constraint remains fulfilled.

Next, we need a decomposition $\boldsymbol{W} = \boldsymbol{V}\boldsymbol{V}^T$ of $\boldsymbol{W}$. Such a $\boldsymbol{V}$ exists (because $\boldsymbol{W}$ is positive semidefinite). In fact,

$$\boldsymbol{W} = \boldsymbol{A}\boldsymbol{D}\boldsymbol{A}^T, \quad \boldsymbol{A} = \boldsymbol{I} - \boldsymbol{D}\boldsymbol{P}_{cls}. \tag{5}$$

This follows easily from $(\boldsymbol{1}^T \otimes \boldsymbol{I})\boldsymbol{D}(\boldsymbol{1} \otimes \boldsymbol{I}) = \sum_{c'} \boldsymbol{D}^{(c')} = \boldsymbol{I}$. Thus, $\boldsymbol{W} = \boldsymbol{V}\boldsymbol{V}^T$ with $\boldsymbol{V} = \boldsymbol{A}\boldsymbol{D}^{1/2}$. The matrix $\boldsymbol{A}$ has fixed points $\operatorname{ran}\boldsymbol{W}$, namely if $\boldsymbol{a} \in \operatorname{ran}\boldsymbol{W}$, then $(\boldsymbol{1}^T \otimes \boldsymbol{I})\boldsymbol{a} = \boldsymbol{0}$, i.e. $\boldsymbol{A}\boldsymbol{a} = \boldsymbol{a}$.

Since there exists some $\tilde{\boldsymbol{v}}$ (not unique) s.t. $\boldsymbol{\alpha}' = \boldsymbol{W}\tilde{\boldsymbol{v}}$, we can rewrite the system (4) as

$$\boldsymbol{V}\left(\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}\right)\boldsymbol{V}^T\tilde{\boldsymbol{v}} = \boldsymbol{V}\left(\boldsymbol{V}^T\boldsymbol{u} - \tilde{\boldsymbol{g}}\right),$$

where $\tilde{\boldsymbol{g}}$ is s.t. $\boldsymbol{g} = \boldsymbol{V}\tilde{\boldsymbol{g}}$ (such a vector exists because $\boldsymbol{g} \in \operatorname{ran}\boldsymbol{W}$). This suggests the following procedure for finding $\boldsymbol{\alpha}'$:

$$\left(\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}\right)\boldsymbol{\beta} = \boldsymbol{V}^T\boldsymbol{u} - \tilde{\boldsymbol{g}}, \quad \boldsymbol{\alpha}' = \boldsymbol{V}\boldsymbol{\beta}. \tag{6}$$

To see the validity of this approach, simply multiply both sides of (6) by $\boldsymbol{V}$ from the left, which shows that $\boldsymbol{V}\boldsymbol{\beta}$ solves the original system. Since the latter has a unique solution (strict convexity!), we must have $\boldsymbol{V}\boldsymbol{\beta} = \boldsymbol{\alpha}'$. Finally, we note that $\tilde{\boldsymbol{g}} = \boldsymbol{D}^{-1/2}\boldsymbol{g}$ does the job, because $\boldsymbol{V}\boldsymbol{D}^{-1/2}\boldsymbol{g} = \boldsymbol{A}\boldsymbol{g} = \boldsymbol{g}$. The latter follows because $\boldsymbol{g} \in \operatorname{ran}\boldsymbol{W}$.

Thus, in exact arithmetic, the Newton direction computation is implemented in a three-stage procedure. First, compute $\tilde{\boldsymbol{g}} = \boldsymbol{D}^{-1/2}\boldsymbol{g}$. Second, solve the system (6) for $\boldsymbol{\beta}$. This is a symmetric positive definite system with the typically well-conditioned matrix $\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}$, and can be solved efficiently using the *linear conjugate gradients* (LCG) algorithm (Saad, 1996). The cost of each step is dominated by the MVM $\boldsymbol{v} \mapsto \boldsymbol{K}\boldsymbol{v}$, which scales linearly in C, due to the block-diagonal structure of $\boldsymbol{K}$. Third, set $\boldsymbol{\alpha}' = \boldsymbol{V}\boldsymbol{\beta}$. The Newton direction is obtained as $\boldsymbol{\alpha}' - \boldsymbol{\alpha}$.

We can start the LCG run from a good guess as follows. Let $\boldsymbol{\alpha}$ be the current dual vector which solved the last recent system. We would like to initialize $\boldsymbol{\beta}$ s.t. $\boldsymbol{\alpha} = \boldsymbol{V}\boldsymbol{\beta} = \boldsymbol{A}\boldsymbol{D}^{1/2}\boldsymbol{\beta}$. If we assume that $\boldsymbol{D}^{1/2}\boldsymbol{\beta} \in \operatorname{ran}\boldsymbol{W}$, then $\boldsymbol{\alpha} = \boldsymbol{D}^{1/2}\boldsymbol{\beta}$. Therefore, a good initialization is $\boldsymbol{\beta} = \boldsymbol{D}^{-1/2}\boldsymbol{\alpha}$. Alternatively, we may also retain $\boldsymbol{\beta}$ from the last recent system.

Issues of numerical stability are addressed in Appendix B. Furthermore, the LCG algorithm is hardly ever run without some sort of preconditioning. Our present implementation uses diagonal preconditioning, as described in Appendix B. We have already noted in Section 3 that a non-diagonal preconditioning strategy could be valuable, but this is subject to future work.

5.1.2 The Line Search

The classical NR algorithm proceeds doing full steps $\boldsymbol{\alpha} \to \boldsymbol{\alpha}'$, but modern variants typically employ a line search along the Newton direction $\boldsymbol{\alpha}' - \boldsymbol{\alpha}$. In the non-convex case, this ensures global convergence, and even for our convex objective Φ, a line search saves time and leads

Algorithm 1 Newton-Raphson optimization to find posterior mode $\hat{\boldsymbol{\alpha}}$.

```
Require: Starting values for $\boldsymbol{\alpha}, \boldsymbol{b}$. Targets $\boldsymbol{y}$.
  $\boldsymbol{\alpha} = \boldsymbol{\alpha} - C^{-1}(\sum_{c'} \boldsymbol{\alpha}^{(c')})_c$, so that $\boldsymbol{\alpha} \in \operatorname{ran} \boldsymbol{W}$. $\boldsymbol{u} = \tilde{\boldsymbol{K}}\boldsymbol{\alpha}$.
  repeat
    Compute $\boldsymbol{l}$, $\log(\boldsymbol{\pi})$ from $\boldsymbol{u}$. Compute $\Phi$.
    if relative improvement in $\Phi$ small enough then
      Terminate outer loop.
    else if maximum number of iterations done then
      Terminate outer loop.
    end if
    Initialize $\boldsymbol{\beta} = \boldsymbol{D}^{-1/2}\boldsymbol{\alpha}$. Compute r.h.s. $\boldsymbol{r} = \boldsymbol{V}^T\boldsymbol{u} - \tilde{\boldsymbol{g}}$, $\tilde{\boldsymbol{g}} = \boldsymbol{D}^{-1/2}\boldsymbol{g}$.
    Compute preconditioner $\operatorname{diag}(\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V})$.
    Run preconditioned CG algorithm in order to solve the system (6) approximately. The
    CG code is configured by a primitive to compute $\boldsymbol{v} \mapsto (\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V})\boldsymbol{v}$, which in turn
    calls the primitive for $\boldsymbol{v} \mapsto \boldsymbol{K}\boldsymbol{v}$.
    Compute $\boldsymbol{\alpha}' = \boldsymbol{A}\boldsymbol{D}^{1/2}\boldsymbol{\beta}'$.
    Do line search along $\boldsymbol{s} = \boldsymbol{\alpha}' - \boldsymbol{\alpha}$. This is done in $\boldsymbol{u}$, along $\tilde{\boldsymbol{s}} = \tilde{\boldsymbol{K}}\boldsymbol{s}$.
    Assign line minimizer to $\boldsymbol{\alpha}$, $\boldsymbol{u}$.
  until forever
```

to numerically more stable behaviour. Interestingly, the special structure of our problem leads to the fact that line searches essentially come for free, certainly compared with the effort of obtaining Newton directions. We refer to this simple idea as *pair optimization*, the reader may be reminded of similar tricks in primal-dual schemes for SVM.

Let $\boldsymbol{s} = \boldsymbol{\alpha}' - \boldsymbol{\alpha}$ be the NR direction, computed as shown above, and set $\boldsymbol{\alpha}_0$ to $\boldsymbol{\alpha}$. The line search minimizes (or sufficiently decreases) Φ on the line segment $\boldsymbol{\alpha}_0 + \lambda\boldsymbol{s}$, $\lambda \in (0, 1]$, starting with $\lambda = 1$ (which is the classical Newton step). The idea is to treat Φ as a function of the pair $(\boldsymbol{u}, \boldsymbol{\alpha})$, where $\boldsymbol{u} = \tilde{\boldsymbol{K}}\boldsymbol{\alpha}$. The corresponding line segment is $\boldsymbol{u} = \boldsymbol{u}_0 + \lambda\tilde{\boldsymbol{s}}$, $\tilde{\boldsymbol{s}} = \tilde{\boldsymbol{K}}\boldsymbol{s}$, requiring a single kernel MVM for computing $\tilde{\boldsymbol{s}}$. Let $j = \operatorname{argmax} |\tilde{s}_j|$. For an evaluation of Φ at $\boldsymbol{u}$, we reconstruct $\lambda = (u_j - u_{0,j})/\tilde{s}_j$ and $\boldsymbol{\alpha} = \boldsymbol{\alpha}_0 + \lambda\boldsymbol{s}$, then

$$\Phi = \boldsymbol{u}^T\left((1/2)\boldsymbol{\alpha} - \boldsymbol{y}\right) + \boldsymbol{1}^T\boldsymbol{l}, \quad \nabla\Phi = \boldsymbol{\pi} - \boldsymbol{y} + \boldsymbol{\alpha},$$

so that an evaluation comes at the cost $O(n\,C)$ and does not require additional kernel MVM applications. We now do the line minimization of Φ in the variable $\boldsymbol{u}$. The driving feature of pair optimization is that we can go back and forth between $\boldsymbol{\alpha}$ and $\boldsymbol{u}$ without significant cost, once the search direction is known w.r.t. both variables.

5.1.3 Overview of the Optimization Algorithm

In Algorithm 1, we give a schematic overview of the primary fitting algorithm, written in terms of a MVM primitive $\boldsymbol{v} \mapsto \boldsymbol{K}\boldsymbol{v}$. For simplicity, we do not include the measures discussed in Appendix B to increase numerical stability.

5.2 Details for Hyperparameter Learning

In this section, we provide details for the CV hyperparameter learning scheme, introduced in Section 4. The gradient of the CV criterion Ψ (3) is computed as follows. Ψ is a sum of terms Ψ_{I_k}, one for each fold. We focus on a single term and write $I = I_k$, $J = J_k$. $\boldsymbol{\alpha}_{[J]}$ is determined by the stationary equation $\boldsymbol{\alpha}_{[J]} + \boldsymbol{g}_{[J]} = \boldsymbol{0}$ (all terms of subscript $[J]$ are as in Section 5.1.1, but for the subset J of the data, and w.r.t. $\boldsymbol{\alpha}_{[J]}$). Taking derivatives gives

$$d\boldsymbol{\alpha}_{[J]} = -\boldsymbol{W}_{[J]}\left((d\boldsymbol{K}_J)\boldsymbol{\alpha}_{[J]} + \tilde{\boldsymbol{K}}_J(d\boldsymbol{\alpha}_{[J]})\right),$$

since $d\boldsymbol{g}_{[J]} = \boldsymbol{W}_{[J]}d\boldsymbol{u}_{[J]}$. We obtain a system for $d\boldsymbol{\alpha}_{[J]}$ which is symmetrised as in Section 5.1.1:

$$\left(\boldsymbol{I} + \boldsymbol{V}_{[J]}^T\tilde{\boldsymbol{K}}_J\boldsymbol{V}_{[J]}\right)\boldsymbol{\beta} = -\boldsymbol{V}_{[J]}^T(d\boldsymbol{K}_J)\boldsymbol{\alpha}_{[J]}, \quad d\boldsymbol{\alpha}_{[J]} = \boldsymbol{V}_{[J]}\boldsymbol{\beta}.$$

Also,

$$d\Psi_I = \left(\boldsymbol{\pi}_{[I]} - \boldsymbol{y}_I\right)^T((d\boldsymbol{K}_{I,J})\boldsymbol{\alpha}_{[J]} + \tilde{\boldsymbol{K}}_{I,J}(d\boldsymbol{\alpha}_{[J]})).$$

With

$$\boldsymbol{f} = \boldsymbol{I}_{\cdot,I}(\boldsymbol{\pi}_{[I]} - \boldsymbol{y}_I) - \boldsymbol{I}_{\cdot,J}\boldsymbol{V}_{[J]}\left(\boldsymbol{I} + \boldsymbol{V}_{[J]}^T\tilde{\boldsymbol{K}}_J\boldsymbol{V}_{[J]}\right)^{-1}\boldsymbol{V}_{[J]}^T\tilde{\boldsymbol{K}}_{J,I}(\boldsymbol{\pi}_{[I]} - \boldsymbol{y}_I),$$

we have that $d\Psi_I = (\boldsymbol{I}_{\cdot,J}\boldsymbol{\alpha}_{[J]})^T(d\boldsymbol{K})\boldsymbol{f}$.

If we collect these vectors as columns of $\boldsymbol{E}$, $\boldsymbol{F} \in \mathbb{R}^{nC,q}$, q the number of folds, we have that

$$d\Psi = \operatorname{tr}\boldsymbol{E}^T(d\boldsymbol{K})\boldsymbol{F} \tag{7}$$

for the complete criterion. The computation of $\boldsymbol{E}$, $\boldsymbol{F}$ was called "accumulation" in Section 4. It involves a loop over folds, in which $\boldsymbol{\alpha}_{[J_k]}$ is determined by NR optimization, starting from its previous value, then $\boldsymbol{f}$ (column of $\boldsymbol{F}$) is computed by solving one more system of the same form as is required to compute Newton directions. Importantly, this accumulation phase is independent of the number of hyperparameters. The gradient computation then requires to compute (7) for each component, using kernel derivative MVMs. First of all, $\partial\boldsymbol{K}/\partial h_p$ is block-diagonal just as $\boldsymbol{K}$, and for many standard kernels, it is a simple expression, involving $\boldsymbol{K}$ itself (see Section 7.3), so one may be able to share computations between the different gradient components. Importantly, the computation of (7) is easily broken down into large numerical linear algebra primitives, for which very efficient code may be used (see Section 7.2). This is a significant advantage in the presence of many hyperparameters. For moderately many hyperparameters, the accumulation clearly dominates the CV criterion and gradient computation.

The dominating part of the accumulation is the reoptimisation of the $\boldsymbol{\alpha}_{[J]}$, which are done by calling the optimized code for primary fitting (Section 5.1) as subroutine. Here, a feature of our implementation becomes important. Instead of representing each $\boldsymbol{K}_{J_k}$ separately, we represent the full $\boldsymbol{K}$ only for all subset kernel MVMs. The representation depends on the covariance function, and in general on how kernel MVMs are actually done. A generic representation is described in Appendix D.1. In order to work on the data subset J_k, we *shuffle* the representation such that in the permuted kernel matrix, $\boldsymbol{K}_{J_k}$ forms the upper left corner. This means that linear algebra primitives with $\boldsymbol{K}_{J_k}$ can be run without mapping matrix coordinates through an index, which would be many times slower. Details on "covariance shuffling" are given in Appendix D.2.

As mentioned in Section 5.1.1 and detailed in Appendix C, we can also compute Newton directions exactly in $O(C\,n^3)$ in the flat classification case. This exact treatment can be extended to the computation of Ψ and its gradient, as is shown in Appendix C. Exact computations lead to more robust behaviour, and may actually run faster for small to moderate n. Exact computations are also useful for debugging purposes.

5.3 Details for Hierarchical Classification

In this section, we provide details for hierarchical classification method, introduced in Section 3. Recall that $\boldsymbol{u} = (\boldsymbol{\Phi} \otimes \boldsymbol{I})\breve{\boldsymbol{u}}$ for an indicator matrix $\boldsymbol{\Phi}$ of simple structure, and that MVM with $\boldsymbol{\Phi}$ or $\boldsymbol{\Phi}^T$ can be computed easily in $O(P)$, without having to store $\boldsymbol{\Phi}$. Since the $\breve{u}_p(\cdot)$ are given independent priors (or regularisers), the corresponding kernel matrix $\breve{\boldsymbol{K}}$ is block-diagonal. The induced covariance matrix $\boldsymbol{K}$ over $\boldsymbol{u}_L$ is given by (2), and hierarchical classification differs from the flat variant only in that this non-block-diagonal matrix is used.

The MVM primitive $\boldsymbol{v} \mapsto \boldsymbol{K}\boldsymbol{v}$ is computed in three steps. MVM with $(\boldsymbol{\Phi}_{L,\cdot} \otimes \boldsymbol{I})$ and $(\boldsymbol{\Phi}_{L,\cdot}^T \otimes \boldsymbol{I})$ works by computing $\boldsymbol{S} \mapsto \boldsymbol{S}\boldsymbol{\Phi}$, $\boldsymbol{S} \mapsto \boldsymbol{S}\boldsymbol{\Phi}^T$ for $\boldsymbol{S} \in \mathbb{R}^{n,P}$. In between, MVM with $\breve{\boldsymbol{K}}$ has to be done in the same way as for flat classification, only that $\breve{\boldsymbol{K}}$ has P rather than C diagonal blocks.

The diagonal preconditioning of LCG (see Appendix B) requires the computation of $\operatorname{diag}\boldsymbol{K} \in \mathbb{R}^{nC}$. We have

$$\boldsymbol{K}_{ic,ic} = (\boldsymbol{\delta}_p^T\boldsymbol{\Phi} \otimes \boldsymbol{\delta}_i^T)\breve{\boldsymbol{K}}(\boldsymbol{\Phi}^T\boldsymbol{\delta}_p \otimes \boldsymbol{\delta}_i) = \boldsymbol{\delta}_p^T\boldsymbol{\Phi}(\operatorname{diag}(\breve{\boldsymbol{K}}_i^{(p')})_{p'})\boldsymbol{\Phi}^T\boldsymbol{\delta}_p, \quad p = L(c),$$

where $\boldsymbol{\delta}_p^T\boldsymbol{\Phi}$ is the p-th row of $\boldsymbol{\Phi}$. From the recursive structure of $\boldsymbol{\Phi}$ we know that if $n_p > 0$, then $\boldsymbol{\delta}_{cs_p+j}^T\boldsymbol{\Phi} = \boldsymbol{\delta}_p^T\boldsymbol{\Phi} + \boldsymbol{\delta}_{cs_p+j}^T$, $j = 1, \dots, n_p$, so if

$$d_{i(cs_p+j)} = d_{ip} + \breve{\boldsymbol{K}}_i^{(cs_p+j)}, \; j = 1, \dots, n_p,$$

then $\operatorname{diag}\boldsymbol{K} = \boldsymbol{d}_L$.

Hyperparameter learning (see Section 5.2) is easily extended to the hierarchical case, recalling (2) and the fact that $\boldsymbol{\Phi}$ does not depend on hyperparameters. Define $\tilde{\boldsymbol{E}} = (\boldsymbol{\Phi}_{L,\cdot}^T \otimes \boldsymbol{I})\boldsymbol{E} \in \mathbb{R}^{nP,q}$, $\tilde{\boldsymbol{F}}$ accordingly, with $\boldsymbol{E}$, $\boldsymbol{F}$ given in Section 5.2. The gradient components (7) translate to $\operatorname{tr}\tilde{\boldsymbol{E}}^T(d\breve{\boldsymbol{K}})\tilde{\boldsymbol{F}}$, where $\breve{\boldsymbol{K}}$ is block-diagonal as before. In our implementation, we reserve buffer space for $\tilde{\boldsymbol{E}}$, $\tilde{\boldsymbol{F}}$, yet build $\boldsymbol{E}$, $\boldsymbol{F}$ there during accumulation. We then transform them to $\tilde{\boldsymbol{E}}$, $\tilde{\boldsymbol{F}}$ using in-place computations.

The step from flat to hierarchical classification requires only minor modifications of existing code. Wrappers for MVM and the other primitives essentially pre- and postmultiply their input with $\boldsymbol{\Phi}$ and $\boldsymbol{\Phi}^T$ respectively, calling the existing "flat" primitives for $\breve{\boldsymbol{K}}$ in between (block-diagonal with P rather than C blocks).

5.4 Why Newton Raphson?

Why do we propose to use the second-order NR method for minimizing Φ, instead of using a simpler gradient-based technique such as scaled conjugate gradients (SCG)? We already motivated our choice at the end of Section 2, but give more details concerning this important point here.

The convex problems we are interested in here live in very high-dimensional spaces and come with complicated couplings between the components which cannot be characterized simply. Certainly, there is no simple decomposition into parts. It is well known in the Optimization literature (Bertsekas, 1999) that simple gradient-based techniques such as SCG require well-chosen preconditioning in order to work effectively in such cases.

For example, we could optimize Φ (1) w.r.t. $\boldsymbol{\alpha}$ directly, the gradient requires a single MVM with $\boldsymbol{K}$ rather than solving a system. However, this problem is very ill-conditioned, the Hessian being $\tilde{\boldsymbol{K}}\boldsymbol{W}\tilde{\boldsymbol{K}} + \tilde{\boldsymbol{K}}$ (large kernel matrices are typically very ill-conditioned, and here we deal with $\boldsymbol{K}^2$), and SCG runs exceedingly slowly to the point of being essentially useless (as we determined in experiments). It can be saved (to our knowledge) only by preconditioning, which in our case requires to solve a system again. Another idea is to optimize Φ w.r.t. $\boldsymbol{u}$ by SCG, which works better. The Hessian is $\boldsymbol{W} + \tilde{\boldsymbol{K}}^{-1}$, whose condition number is similar to $\boldsymbol{K}$. In preliminary direct comparisons, the NR method still works more efficiently, meaning that SCG would require additional preconditioning specific to the problem at hand, which would likely be different for flat and hierarchical classification. From our experience, and also from the predominance of NR in the Optimization literature, we opted for this method which seems to come with self-tuning capabilities, making it easier to transfer the framework to novel problems.

6. Experiments

In this section, we provide experimental results for our method on a range of flat and hierarchical classification tasks.

6.1 Hierarchical Classification: Patent Texts

We use the WIPO-alpha collection[10], many thanks to L. Cai, T. Hofmann for providing us with the count data and dictionary. We did Porter stemming, stop word removal, and removal of empty categories. The attributes are bag-of-words over the dictionary. All input vectors $\boldsymbol{x}_i$ were scaled to unit norm. Many thanks to Peter Gehler for helping us with the preprocessing.

These tasks have previously been studied by Cai and Hofmann (2004), where patents (title and claim text) are to be classified w.r.t. the standard taxonomy *IPC*, a tree with 4 levels and 5229 nodes. Sections A, B,..., H form the first level. As in (Cai and Hofmann, 2004), we concentrate on the 8 subtasks rooted at the sections, ranging in size from D ($n = 1140$, $C = 160$, $P = 187$) to B ($n = 9794$, $C = 1172$, $P = 1319$). We use linear kernels (see Appendix D.3) with variance parameters v_c.

All experiments are averaged over three training/test splits, different methods using the same ones. The CV criterion Ψ is used with a different (randomly drawn) 5-partition per section and split, the same across all methods. Our method outputs a predictive distribution $\boldsymbol{p}_j \in \mathbb{R}^C$ for each test case $\boldsymbol{x}_j$. The standard prediction $y(\boldsymbol{x}_j) = \operatorname{argmax}_c p_{jc}$ maximizes expected accuracy, classes are ranked as $r_j(c) \leq r_j(c')$ iff $p_{jc} \geq p_{jc'}$, where $r_j(c) \in \{1, \dots, C\}$

10. Available at `www.wipo.int/tools/en/dbindex.html`, or google for "Data Collections hosted by WIPO".

is the rank of class c for case $\boldsymbol{x}_j$. Let y_j denote the true label for $\boldsymbol{x}_j$. The test scores we use here are the same as in (Cai and Hofmann, 2004): *accuracy* (acc) $m^{-1}\sum_j \mathrm{I}_{\{y(\boldsymbol{x}_j)=y_j\}}$, *precision* (prec) $m^{-1}\sum_j r_j(y_j)^{-1}$, *parent accuracy* (pacc) $m^{-1}\sum_j \mathrm{I}_{\{\mathrm{par}(y(\boldsymbol{x}_j))=\mathrm{par}(y_j)\}}$, $\mathrm{par}(c)$ being the parent of leaf node $L(c)$ (recall that $L(c)$ corresponds to class c). Here, m is the test set size. Let $\Delta(c,c')$ be half the length of the shortest path between leafs $L(c)$, $L(c')$. The *taxo-loss* (taxo) is $m^{-1}\sum_j \Delta(y(\boldsymbol{x}_j),y_j)$. These scores are motivated by Cai and Hofmann (2004). For taxo-loss and parent accuracy, we better choose $y(\boldsymbol{x}_j)$ to minimize expected loss[11], which is different in general than the standard prediction (the latter maximizes expected accuracy and precision).

We compare methods F1, F2, H1, H2 (F: flat, not using IPC; H: hierarchical). F1: all v_c shared (1); H1: v_c shared across each level of the tree (3). F2, H2: v_c shared across each subtree rooted at root's children (A: 15, B: 34, C: 17, D: 7, E: 7, F: 17, G: 12, H: 5). The numbers in parentheses are the total number of hyperparameters. Recall that there are three parameters determining the running time (see Section 2, Section 4). For hyperparameter learning: $k_1 = 8, k_2 = 4, k_3 = 15$ (F1, F2); $k_1 = 10, k_2 = 4, k_3 = 25$ (H1, H2)[12]. For the final fitting (after hyperpars have been learned): $k_1 = 25, k_2 = 12$ (F1, F2); $k_1 = 30, k_2 = 17$ (H1, H2). The optimization is started from $v_c = 5$ for all methods. We set $\sigma^2 = 0.01$ throughout. Results are given in Table 1.

The hierarchical model outperforms the flat one consistently, especially w.r.t. taxo-loss and parent accuracy. Also, minimizing expected loss is consistently better than using the standard rule for the latter, although the differences are not significant. H1 and H2 do not perform differently: choosing many different v_c in the linear kernel seems no advantage here (but see Section 6.2). The results are quite similar to the ones of Cai and Hofmann (2004), obtained with a support vector machine variant. However, for our method, the recommendation in (Cai and Hofmann, 2004) to use $v_c = 1$ (not further motivated there) leads to significantly worse results in all scores. The v_c chosen by our method are generally larger. Note that their code has not been made publicly available, so a direct comparison with "all other things equal" could not be done.

In Table 2, we present running times[13] for the final fitting and for a single fold during hyperparameter optimization (5 of these are required for Ψ, $\nabla_{\boldsymbol{h}}\Psi$). In comparison, Cai and Hofmann (2004) quote a final fitting time of $2200s$ on the D section, using a SVM variant, while we require $119s$ (more than six times faster)[14]. It is precisely this high efficiency of primary fitting, which allows us to use it as inner loop for automatic hyperparameter learning (Cai and Hofmann (2004) do not adjust hyperparameters to the data). Possible reasons for the performance difference are given in Section 7.2.

11. For parent accuracy, let $p(j)$ be the node with maximal mass (under $\boldsymbol{p}_j$) of its children which are leafs, then $y(\boldsymbol{x}_j)$ must be a child of $p(j)$.
12. Except for section C, where $k_1 = 14, k_2 = 6, k_3 = 35$.
13. Processor time on 64bit 2.33GHz AMD machines.
14. Cai and Hofmann average over three training/test splits. The timing figure $2200s$ in their paper is for three splits (thanks to one of the reviewers to point this out).

	acc (%)				prec (%)				taxo			
	F1	H1	F2	H2	F1	H1	F2	H2	F1	H1	F2	H2
A	40.6	**41.9**	40.5	**41.9**	51.6	**53.4**	51.4	**53.4**	1.27	**1.19**	1.29	**1.19**
B	32.0	**32.9**	31.7	32.7	41.8	**43.8**	41.6	43.7	1.52	**1.44**	1.55	**1.44**
C	33.7	**34.7**	34.1	34.5	45.2	**46.6**	45.4	46.4	1.34	**1.26**	1.35	1.27
D	40.0	40.6	39.7	**40.8**	52.4	54.1	52.2	**54.3**	1.19	**1.11**	1.18	**1.11**
E	33.0	**34.2**	32.8	34.1	45.1	**47.1**	45.0	**47.1**	1.39	**1.31**	1.38	**1.31**
F	31.4	32.4	31.4	**32.5**	42.8	44.9	42.8	**45.0**	1.43	**1.34**	1.43	**1.34**
G	40.1	**40.7**	40.2	**40.7**	51.2	**52.5**	51.3	**52.5**	1.32	**1.26**	1.32	**1.26**
H	39.3	39.6	39.4	**39.7**	52.4	53.3	52.5	**53.4**	1.17	1.15	1.17	**1.14**

	taxo[0-1]				pacc (%)				pacc[0-1] (%)			
	F1	H1	F2	H2	F1	H1	F2	H2	F1	H1	F2	H2
A	1.28	1.19	1.29	**1.18**	58.9	**61.6**	58.2	61.5	57.2	61.3	56.9	**61.4**
B	1.54	**1.44**	1.56	**1.44**	53.6	56.4	52.7	**56.6**	51.9	**55.9**	51.4	**55.9**
C	1.33	**1.26**	1.32	**1.26**	58.9	**62.6**	58.5	62.0	58.6	**61.8**	58.9	61.6
D	1.20	**1.12**	1.22	**1.12**	64.6	67.0	64.4	**67.1**	63.5	**67.1**	62.6	67.0
E	1.43	**1.33**	1.44	1.34	56.0	59.1	56.2	**59.2**	54.0	**58.2**	53.5	57.9
F	1.43	**1.34**	1.44	**1.34**	56.8	59.7	56.8	**59.8**	54.9	58.7	54.6	**58.9**
G	1.32	**1.26**	1.32	**1.26**	58.0	**59.7**	57.6	59.6	56.8	**59.2**	56.6	58.9
H	1.19	1.16	1.19	**1.15**	61.6	**62.5**	61.8	**62.5**	59.9	61.6	60.0	**61.8**

Table 1: Results on patent text classification tasks A-H. Methods F1, F2 flat, H1, H2 hierarchical. taxo[0-1], pacc[0-1] for $\mathrm{argmax}_c\, p_{jc}$ standard prediction rule, rather than minimization of expected loss.

	Final NR (s)		CV Fold (s)			Final NR (s)		CV Fold (s)	
	F1	H1	F1	H1		F1	H1	F1	H1
A	2030	3873	573	598	E	131.5	203.4	32.2	49.6
B	3751	8657	873	1720	F	1202	2871	426	568
C	4237	7422	719	1326	G	1342	2947	232	579
D	56.3	118.5	9.32	20.2	H	971.7	1052	146	230

Table 2: Running times for tasks A-H. Method F1 flat, H1 hierarchical. Final NR: Final fitting with Newton-Raphson. CV Fold: Re-optimization of $\boldsymbol{\alpha}_{[J]}$ and gradient accumulation for single fold J.

6.2 Flat Classification: Remote Sensing

We use the *satimage* remote sensing task from the *statlog* repository.[15] This task has been used in the extensive SVM multi-class study of Hsu and Lin (2002), where it is among the datasets on which the different methods show the most variance. It has $n = 4435$ training, $m = 2000$ test cases, and $C = 6$ classes. Covariates $\boldsymbol{x}$ have 36 attributes with values in $\{0, \dots, 255\}$. No preprocessing was done.

We use the isotropic *Gaussian* (RBF) covariance function

$$K^{(c)}(\boldsymbol{x}, \boldsymbol{x}') = v_c \exp\left(-\frac{w_c}{2}\|\boldsymbol{x} - \boldsymbol{x}'\|^2\right), \quad v_c, w_c > 0. \tag{8}$$

We compare the methods *mc-sep* (ours with separate kernels for each class; 12 hyperparameters), *mc-tied* (ours with a single shared kernel; 2 hyperparameters), *mc-semi* (ours with single kernel $M^{(1)}$, but different v_c; 7 hyperparameters), *1rest* (one-against-rest; 12 hyperparameters). For *1rest*, C binary classifiers are fitted on the tasks of separating class c from all others. They are combined afterwards by the rule $\boldsymbol{x}_* \mapsto \operatorname{argmax}_c \hat{P}_c(+1|\boldsymbol{x}_*)$, where $\hat{P}_c(+1|\boldsymbol{x}_*)$ is the predictive probability estimate of the c-classifier[16]. Note that *1rest* is arguably the most efficient method, in that its binary classifiers can be fitted separately and in parallel. Even if run sequentially, *1rest* typically requires less memory by a factor of C than a joint multi-class method, although this is not true if the kernel matrices are dominating the memory requirements and they are shared between classes in a multi-class method (as in *mc-tied* and *mc-semi* here).

We use our 5-fold CV criterion Ψ for each method. Results here are averaged over ten randomly drawn 5-partitions of the training set (the same partitions are used for the different methods). All optimizations are started from $v_c = 10$, $w_c = (\sum_j \mathrm{Var}[x_j])^{-1} = 0.017$, $\mathrm{Var}[x_j]$ being the empirical variance of attribute j. We set $\sigma^2 = 16$ throughout. The parameters determining the running time (see Section 2, Section 4) are set to $k_1 = 13$, $k_2 = 25$, $k_3 = 40$ during hyperparameter learning, and $k_1 = 30$, $k_2 = 50$ for final fitting (these are very conservative settings). Error-reject curves are shown in Figure 3.

Test errors are 7.95%(±0.15%) for *mc-sep*, 8.00%(±0.10%) for *1rest*, 8.10%(±0.13%) for *mc-semi*, and 8.35%(±0.20%) for *mc-tied*. Therefore, using a single fixed kernel for all $K^{(c)}$ does significantly worse than allowing for an individual $K^{(c)}$ per class. The test error difference between *mc-sep* and *1rest* is not significant here, but the error-reject curve is significantly better for our method *mc-sep* than for one-against-rest, especially in the domain $\alpha \in [0.025, 0.25]$, arguably most important in practice (where the rejection of a small fraction of test cases may often be an option). This indicates that the predictive probability estimates from our method are better than from one-against-rest, at least w.r.t. their ranking property. The curves for *mc-semi*, *mc-tied* are closer to *1rest*, underlining that different kernels $K^{(c)}$ should be used for each class. The result for *mc-sep* is state-of-the-art. The best SVM technique tested in (Hsu and Lin, 2002) attained 7.65% (no error-reject curves were given there), and SVM one-against-rest attained 8.3% in this study. To put this into perspective, note that Hsu and Lin (2002) did extensive hyperparameter selection

15. Available at `http://www.niaad.liacc.up.pt/old/statlog/`.
16. Asymptotically, $\hat{P}_c(+1|\boldsymbol{x}_*)$ converges to the true $P(y_* = c|\boldsymbol{x}_*)$, and this combination rule is optimal. We use our method with $C = 2$ in order to implement the binary classifiers.

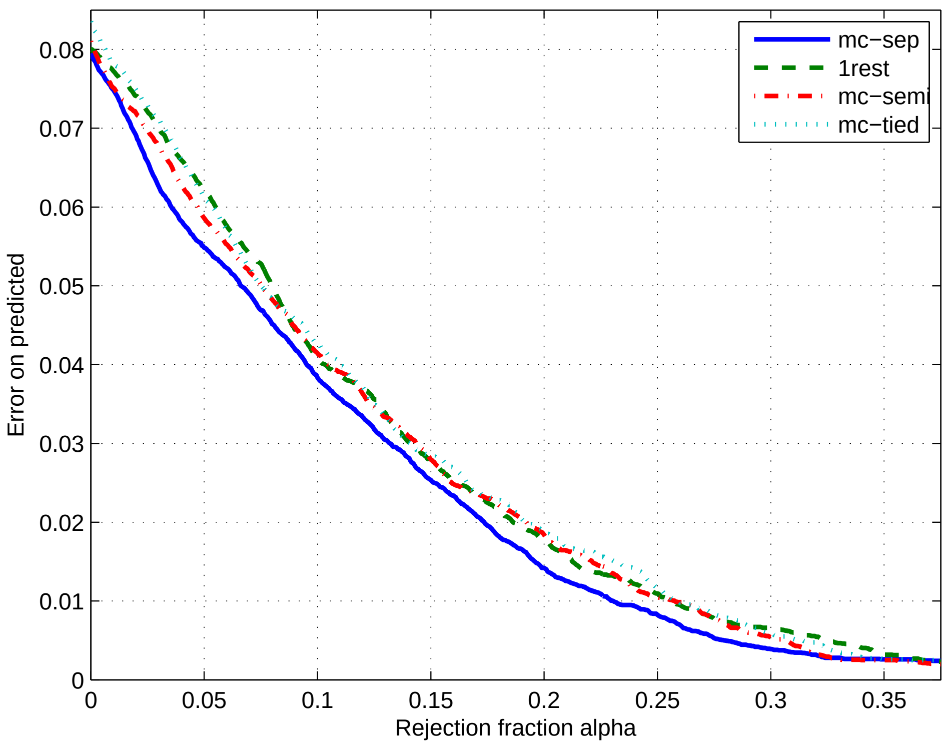

Figure 3: Error-reject curves (averaged over 10 runs) for different methods on the *satimage* task. Curve obtained by allowing the method to abstain from prediction on fraction α of test set, counting errors for predictions only. Depends on ranking of test points. Ranking score (over test points $\boldsymbol{x}_*$): $\max_c \hat{P}(y_* = c|\boldsymbol{x}_*)$ (*mc-sep*, *mc-semi*, *mc-tied*), $\max_c \hat{P}_c(+1|\boldsymbol{x}_*)$ (*1rest*).

by cross-validation, in what seems to be a quite user-intensive process, while our method is completely automatic.

6.3 Flat Classification: Handwritten Digits

We use the *USPS* handwritten digits recognition task (LeCun et al., 1989). The covariates $\boldsymbol{x}$ are 16×16 grayscale images with values in $\{16k + 15 \,|\, k = 0, \dots, 30\}$. The task has $n = 7291$ training, $m = 2007$ test cases, and $C = 10$ classes. No preprocessing was done.

We use Gaussian kernels (8) once more, different ones for each class. We do not optimize the 5-fold CV criterion Ψ using the full training set, but subsets of size $n' = 2000$. Our results are averaged over five runs with different randomly drawn training subsets for hyperparameter learning, while we use the full training set for final fitting. All optimizations are started from $v_c = 10$, $w_c = (\sum_j \mathrm{Var}[x_j])^{-1} = 0.0166$, and we set $\sigma^2 = 4$ throughout. The parameters determining the running time are $k_1 = 25$, $k_2 = 35$, $k_3 = 40$ during hyperparameter learning (on $n' = 2000$ points), and $k_1 = 45$, $k_2 = 80$ for final fitting (on $n = 7291$ points). The settings for hyperparameter learning are quite conservative, and the final fitting ones were sufficient for convergence on three of the five runs, whereas on two we had to add another $k_1 = 25$ iterations with $k_2 = 90$. An error-reject curve is shown in Figure 4.

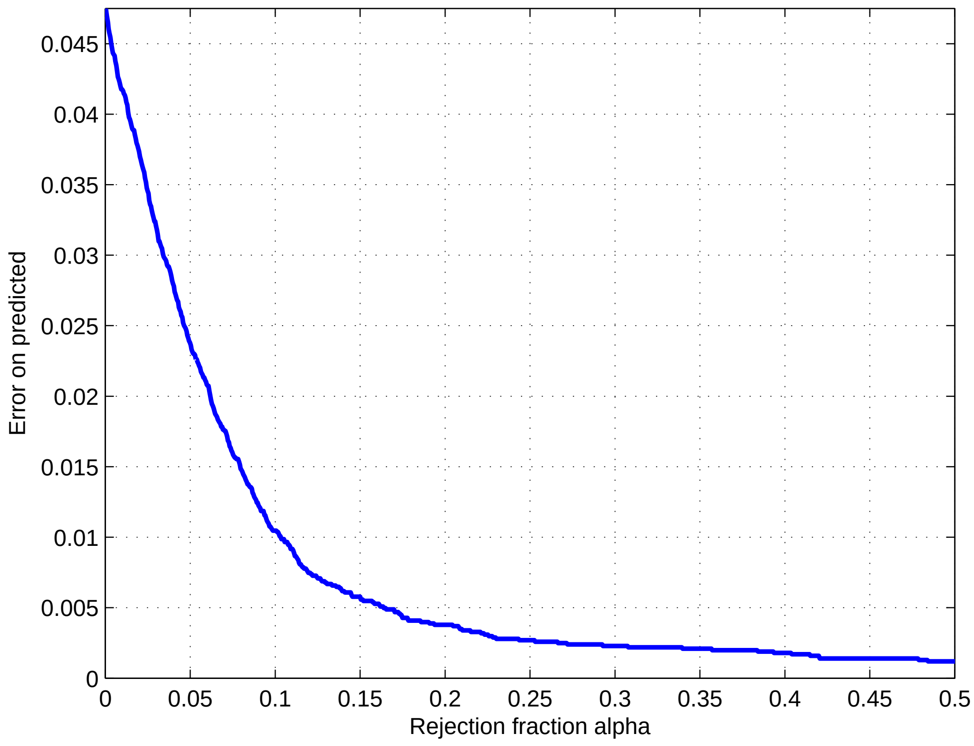

Figure 4: Error-reject curves (averaged over 5 runs) for different methods on the *USPS* task. Curve obtained by allowing the method to abstain from prediction on fraction α of test set, counting errors for predictions only. Depends on ranking of test points.

Test errors are 4.77%(±0.18%). These results are state-of-the-art for kernel classification. Seeger (2003) reports 4.98% for the IVM (Sect. 4.8.4), where hyperparameters are learned automatically. Csató (2002) states 5.15% for his sparse online method with multiple sweeps over the data (Sect. 5.2). Results for the support vector machine are given in Schölkopf and Smola (2002), Table 18.1, method SV-254, where a combination heuristic based on kernel PCA was used to attain a test error of 4.4%. Crammer and Singer (2001) quote a test error of 4.38%, kernel parameters having been selected by 5-fold cross-validation. All these used the Gaussian kernel as well. The latter studies do not quote fluctuations w.r.t. choices such as the fold partition in CV, which is not negligible in our case here. The SVM-based methods do not attempt test set rankings or predictive probability estimation, and the corresponding studies do not show error-reject (or ROC) curves. Seeger (2003) gives an error-reject curve, which is very similar to ours here.

Note that the harder settings of k_1, k_2 for the final fitting are necessary due to the problem size, and are motivated in Section 4. There are 72910 parameters, and the hyperparameters found through optimizing Ψ spread by 3 orders of magnitude, so that the corresponding final fitting problems are computationally hard to solve without a good initialization of $\boldsymbol{\alpha}$ (in the absence of such, we start with $\boldsymbol{\alpha} = \mathbf{0}$). If we solve for Newton directions using too few LCG steps, the approximations often do not lead to much (or any) descent. Such "stalling" of NR line searches does happen now and then even after $k_2 = 80$ LCG steps[17]. Lessons learned from these large scale experiments are commented on in Section 4. There are delicate dependencies between k_1, k_2 and the running time to convergence, which need to be explored in large scale settings, but this was not done thoroughly here.

7. Discussion

We have presented a general framework for learning kernel-based penalized likelihood classification methods from data. A central feature of the framework is its high computational effiency, even though all classes are treated jointly. This is achieved by employing approximate Newton-Raphson optimization for the parameter fitting, which requires few large steps only for convergence. These steps are reduced to matrix-vector multiplication (MVM) primitives with kernel matrices. For general kernels, these MVM primitives can be reduced to large numerical linear algebra primitives, which can be computed very efficiently on modern computer architectures. This is very much in contrast to many chunking algorithms for kernel method fitting, which have been proposed in Machine Learning, and the advantages of our approach are detailed in Section 7.2. Dependencies between classes can be encoded *a priori* with minor additional efforts, as has been demonstrated for the case of hierarchical classification. Our method provides estimates of predictive probabilities which are asymptotically correct. Hyperparameters can be adjusted automatically, by optimizing a cross-validation log likelihood score in a gradient-based manner, and these computations are once more reduced to the same MVM primitives. This means that within our framework, all code optimization efforts can be concentrated on these essential primitives (see also Section 7.3), rather than having to tune a set of further heuristics.

7.1 Related Work

Our primary fitting optimization for flat multi-way classification appeared in (Williams and Barber, 1998), although some fairly essential features are novel here. They also did not consider large scale problems or class structures. Empirical Bayesian criteria such as the marginal likelihood are routinely used for hyperparameter learning in Gaussian process models (Williams and Barber, 1998, Williams and Rasmussen, 1996). However, in cases other than regression estimation with Gaussian noise, the marginal likelihood for a GP

17. We cannot obtain a good initial $\boldsymbol{\alpha}$ value from the final $\boldsymbol{\alpha}_{[J_k]}$ of hyperparameter learning, because this is done on training subsets only. Moreover, in our implementation, the "stalling" (no improvement) of a NR step means that LCG is restarted from its last recent $\boldsymbol{\beta}$, so that eventually an improvement in Φ is still obtained. Of course, the stalled NR iterations counts as such, and we do k_1 iterations in total.

model cannot be computed analytically, and approximations differ strongly in terms of accuracy and computational complexity. For the multi-class model, Williams and Barber (1998) use an MAP approximation for fixed hyperparameters, just as we do, but their second-order approximation to the marginal likelihood is quite different from our criterion, conceptually as well as computationally (see below). Approximately solving large linear system by linear conjugate gradients (LCG) is standard in numerical mathematics, and has been used in Machine Learning as well (Gibbs, 1997, Williams and Barber, 1998, Keerthi and DeCoste, 2005).

The idea of optimizing approximations to a cross-validation score for hyperparameter learning is not novel (Craven and Wahba, 1979, Qi et al., 2004). Our approach is different to these, in that the CV score and gradient computations are reduced to elementary steps of the primary fitting method, so both can be done with the same code[18]. In constrast, scores like GCV (Craven and Wahba, 1979) or second order marginal likelihood (Williams and Barber, 1998) come in terms of the form $\operatorname{tr} \boldsymbol{H}^{-1}$ or $\log |\boldsymbol{H}|$ for the Hessian $\boldsymbol{H}$ of size nC. There are approximate reductions of computing these terms to solving linear systems (randomized trace estimator, Lanczos), but they rely on additional sampling of Gaussian noise, which introduces significant inaccuracies. In practice, optimizing such "noisy" criteria is quite difficult, whereas our criterion can be optimized using standard optimization code. Qi et al. (2004) propose an interesting approach of approximating *leave-one-out (LOO)* CV using *expectation propagation (EP)* (see also (Opper and Winther, 2000)) and use a sparse approximation for efficiency, but they deal with a single-process model only ($C = 1$), and it is not clear how to implement EP efficiently (*i.e.* scaling linearly in C) for the multi-class model. Interestingly, they observe that optimizing their approximate CV score is more robust to overfitting than the marginal likelihood. Finally, none of these papers propose (or achieve) a complete reduction to kernel MVM primitives only, nor do they deal with representing class structures or work on problems of the scale considered here.

Many different multi-class SVM techniques have been proposed, see (Crammer and Singer, 2001, Hsu and Lin, 2002) for references. These can be split into joint ("all-together") and decomposition methods. The latter reduce the multi-class problem to a set of binary ones ("one-against-rest" of Section 6.2 is a prominent example), with the advantage that good code is available for the binary case. The problem with these methods is that the binary discriminants are fitted separately without knowledge of each other, or of their role in the final multi-way classifier, so information from data is wasted. Also, their posthoc combination into a multi-way discriminant is heuristic. Joint methods are like ours here, in that all classes are jointly represented. Fitting is a constrained convex problem, and often fairly sparse solutions (many zeros in $\boldsymbol{\alpha}$) are found. However, in difficult structured label tasks, the degree of sparsity is usually not high, and in these cases, commonly used chunking algorithms for multi-class SVM can be very inefficient (see Section 7.2). We should note that our approach here cannot be applied directly to multi-class SVMs, since they require the solution of a constrained convex problem, but the principles used here should hold there as well (some novel suggestions here appear independently in (Keerthi et al., 2007)). SVM methods typically do not come with efficient automatic kernel parameter

18. A small drawback of our approach is that our CV score Ψ depends on a partitioning of the training set. In our experiments here, we chose this at random. Leave-one-out (LOO) CV does not depend on a partitioning, but it is not clear how to reduce LOO CV to solving a small number of linear systems.

learning schemes, and they do not provide estimates of predictive probabilities which are asymptotically correct.

On the other hand, in a direct comparison our implementation would still be slower than the highly optimized multi-class SVM code of Crammer and Singer (2001), at least on standard non-structured tasks such as USPS (Section 6.3) or MNIST. Especially on the latter, sparsity in $\boldsymbol{\alpha}$ is clearly present, and years of experience with the SVM problem led to very effective ways of exploiting it. In contrast, $\boldsymbol{\alpha}$ in our approach is not sparse, and it is not our goal here to find a sparse approximation. Hyperparameters are selected "by hand" in their method, not via gradient-based optimization. For a small number of hyperparameters, this traditional approach is often faster than our optimization-based one here, and importantly, it can be fully parallelized. However, our approach is still workable in situations with many dependent hyperparameters (for example, Section 7.4.1), where CV by hand simply cannot be done.

Our ANOVA setup for hierarchical classification is proposed by Cai and Hofmann (2004), whose use it within a SVM "all-together" method. We compare our method against theirs in Section 6.1, achieving quite similar results in an order of magnitude less time. They also do not address the problem of hyperparameter learning.

7.2 Global versus Decomposition Methods

In most kernel methods proposed so far in Machine Learning, the primary fitting to data (for fixed hyperparameters) translates to a convex minimization problem, where the penalization terms correspond to quadratic expressions with kernel matrices. While kernel matrices may show a fastly decaying eigenvalue spectrum, they certainly do couple the optimization variables strongly[19]. While a convex function can be optimized by any method which just guarantees descent in each step, there are huge differences in how fast the minimum is attained to a desired accuracy. In fact, in the absence of local minima, the speed of convergence becomes the most important characteristic of a method, besides robustness and ease of implementation.

In Machine Learning, the most dominant technique for large scale (structured label) kernel classification is what Optimization researchers call "block coordinate descent methods" (BCD; see (Bertsekas, 1999), Sect. 2.7). The idea is to minimize the objective w.r.t. a few variables at a time, keeping all others fixed, and to iterate this process using some scheduling over the variables. Each step is convex again[20], yet much smaller than the whole, and often the steps can be solved analytically. Ignoring the aspect of scheduling, such methods are simple to implement.

A complementary approach is to find search directions which lead to as fast a descent as possible, these directions typically involve all degrees of freedom of the optimization variables. If local first and second order information can be computed, the optimal search direction is Newton's, which has to be corrected if constraints are present (conditional gradient or gradient projection methods). If the Newton direction cannot be computed feasibly, approximations may be used. Such Newton-like methods are certainly vastly preferred in

19. An almost low-rank kernel matrix translates into a coupling of a simple structure, but the dominant couplings are typically strong and not sparse.
20. If $f(\boldsymbol{x})$ is convex, so is $f(\boldsymbol{A}\boldsymbol{x})$ for any matrix $\boldsymbol{A}$. The same is true for linear constraints.

the Optimization community, due to superior convergence rates, but also because features of modern computer architectures are used more efficiently, as is detailed below. In this paper, we advocate to follow this preference for kernel machine fitting in Machine Learning. We are encouraged not only by our own experiences, but can refer to the fact that (approximate) Newton methods are standard for fitting generalized linear models in Statistics, and that such methods are also routinely used for Gaussian process models (Williams and Barber, 1998, Rasmussen and Williams, 2006), albeit typically on problems of smaller scale than treated here.

The dominance of BCD methods for kernel machine fitting, while somewhat surprising, can be attributed to early success stories with SVM training, culminating in the SMO algorithm (Platt, 1998), where only two variables are changed at a time. If an SVM is fitted to a task with low noise, the solution can be highly sparse, and if the active set of "support vectors" is detected early in the optimization process, methods like SMO can be very efficient. Importantly, SMO or other BCD methods are easily implemented. On the other hand, as SVMs are increasingly applied to hard structured label problems which usually do not have very sparse solutions, or whose active sets are hard to find, weaknesses of BCD methods become apparent.

Block coordinate descent methods are often referred to as using the "divide-and-conquer" principle, but this is not the case for kernel method fitting. BCD methods "are often useful in contexts where the cost function and the constraints have a partially decomposable structure with respect to the problem's optimization variables." (Bertsekas (1999), p. 269) In kernel methods, such a decomposable structure is not present, because the penalization terms couple all variables strongly via the kernel matrices. In such cases, chunking techniques *divide without conquering*, often running for very many steps, because improvements w.r.t. some block of variables tend to erase earlier improvements. This central problem of block coordinate descent methods is well known as "zig-zagging" in Optimization. It occurs whenever variables not in the same block are significantly coupled, a situation which is to be expected for kernel machines in general. The situation is similar to a number of cases in Machine Learning and Statistics. Iterative proportional fitting (Della Pietra et al., 1997) is a BCD method for learning the potentials of an undirected graphical model (Markov random field), which is all but superseded now by modern global direction methods such as limited memory Quasi-Newton, running orders of magnitude faster. Gibbs sampling is a basic Markov chain Monte Carlo technique for approximate Bayesian inference, which typically is very simple to implement, but is exceedingly slow in the presence of many coupled variables. Modern techniques such as Hybrid Monte Carlo, or Swendsen-Wang can be seen as "global direction" variants of "block coordinate" Gibbs sampling, and while they are harder to implement, they typically run orders of magnitude faster.

Another significant problem with BCD methods may come more as a surprise, namely because it concerns a characteristic which is often sold as advantage of these methods: they make each step as small as possible. Such small steps can often be dealt with analytically, or by using simple methods. This characteristic certainly makes BCD methods easy to implement. However, in light of modern computer architectures, the advice must be to make each step as *large as possible*, with the aim of requiring fewer steps. Modern systems use many internal parallelisms and hierarchies of cacheing, with the aim of processing vector operations many times faster than an equivalent loop over scalar operations, and large global steps do make use of these features. In contrast, a method which calls very many small steps in a non-linear ordering, runs contrary to these mechanisms. For example, data transfer between cache levels is done in blocks of significant sizes, and a method which accesses memory elementwise from all over the place, leads to inefficiences up to cache thrashing, where the majority of cache accesses are misses (see Appendix D.3 for an example).

In well-designed global direction methods, the bottleneck operations (where almost all real-world running time is spent) are large vectorized mappings which access memory contiguously. Even better, these operations should lie in a standard class, for which highly optimized implementations are readily available. In our case here, the bottleneck operations are numerical linear algebra primitives from the *basic linear algebra subroutines* (BLAS), a standardized interface for high-performance dense linear algebra code. Very efficient implementations of the BLAS are available for all computer architectures[21].

In this paper, we advocate to take a step back and to use global direction methods as approximation to Newton's method for kernel machine fitting. The prospect seems daunting, since there are many thousands of variables with complicated couplings, and the reader may be reminded of early desastrous trials of applying off-the-shelf QP packages to SVM fitting, or of ongoing efforts to formulate otherwise tractable Machine Learning problems as semidefinite programs and "solving" them using $O(n^7)$ SDP packages. This association is wrong. Our advice is to *approximate* the global Newton direction, making use of all structure in the model in order to gain efficiency, which is exactly the opposite of running a black box solver or implementing Newton's method straight out of a textbook. In the context of kernel machines, where couplings through large unstructured matrices are present, the large steps of approximating the Newton direction should be reduced to standard linear algebra primitives on dense or sparse matrices, operating on contiguous chunks of memory *as large as possible*, since highly optimized code for such primitives is readily available.

7.3 Matrix-Vector Multiplication

The computational load in our framework is determined by applications of the MVM primitives $\boldsymbol{v} \mapsto \boldsymbol{K}^{(c)}\boldsymbol{v}$ and $\boldsymbol{v} \mapsto (\partial \boldsymbol{K}^{(c)}/\partial h_p)\boldsymbol{v}$. A user only needs to provide those for a kernel of choice. The generic representation of Appendix D.1 applies to general covariance functions, but much more efficient alternatives may be used in special cases (see Appendix D.3).

If the cost for a direct evaluation of these primitives is prohibitive, several well-known approximations may be applied. Its has been suggested to use specialized data structures to approximate MVM with matrices from isotropic kernels (Yang et al., 2005, Shen et al., 2006) such as the Gaussian one (8). For such kernels, the derivative MVM can often be addressed using the same techniques. For the Gaussian kernel, we have $\boldsymbol{K} = v \exp(w\boldsymbol{A})$, $\boldsymbol{A} = (-(1/2)\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2)_{i,j}$, in which case $(\partial \boldsymbol{K}/\partial \log v) = \boldsymbol{K}$ and $(\partial \boldsymbol{K}/\partial \log w) = w\boldsymbol{K} \circ \boldsymbol{A}$, where $\circ$ denotes componentwise product. Since the specialized data structures concentrate on approximating $\boldsymbol{A}$, they apply to all required MVM variants. Our public code could fairly easily be extended, given specialized approximate kernel MVM code is available.

21. ATLAS, a self-tuning BLAS implementation, is available as free software, see `http://math-atlas.sourceforge.net/`.

7.4 Extensions and Future Work

Some concrete extensions are mentioned just below. In general, we think that many structured label kernel methods proposed in the SVM context can be addressed in our framework as well. For example, the kernel conditional random field (CRF) (Lafferty et al., 2004) for label sequence learning can be treated by recognizing that MVM with the Hessian of the CRF log likelihood can be implemented efficiently using the method of Pearlmutter (1994). We also plan to address hierarchical multi-label problems, which differ from hierarchical multi-class in that each instance can have multiple associated labels.

7.4.1 Modelling Dependencies between Classes

In the flat classification application of Section 2, we do not explicitly represent dependencies between classes. This is done in the hierarchical classification application of Section 3, but the dependency structure is fixed *a priori*. In this section, we motivate how dependencies between classes can be learned from data.

Let $\boldsymbol{B} \in \mathbb{R}^{C,C}$ be a nonsingular coupling matrix, which will be a part of the model. In fact, $\boldsymbol{B}$ should be regarded as hyperparameters. In the flat model, we have $\boldsymbol{u}_i = \boldsymbol{f}_i + \boldsymbol{b}$, which is now replaced by $\boldsymbol{u}_i = \boldsymbol{B}\boldsymbol{f}_i + \boldsymbol{b}$, or

$$\boldsymbol{u} = (\boldsymbol{B} \otimes \boldsymbol{I})\boldsymbol{K}\boldsymbol{\alpha} + \boldsymbol{b} \otimes \boldsymbol{1}.$$

This is the same modification which led to the hierarchical extension, only that the fixed coupling matrix $\boldsymbol{\Phi}$ is replaced by the variable $\boldsymbol{B}$. Therefore, the same modification of our method can be done, *i.e.* replacing $\boldsymbol{K}$ by $\boldsymbol{K}^{(B)} = (\boldsymbol{B} \otimes \boldsymbol{I})\boldsymbol{K}(\boldsymbol{B}^T \otimes \boldsymbol{I})$. Again, MVM with $\boldsymbol{K}^{(B)}$ is of the same complexity as with $\boldsymbol{K}$, because MVM with $(\boldsymbol{B} \otimes \boldsymbol{I})$ can be done in $O(C^2 n)$. Note that $\boldsymbol{B}$ represents *conditional* dependencies between the $u_c(\cdot)$, its role is comparable to the "mixing matrix" in independent components analysis.

We are also interested in learning $\boldsymbol{B}$, whose elements are taken as hyperparameters. The corresponding gradient of Ψ is obtained in the same way as described in Section 5.2, only that $d\boldsymbol{K}_{[B]}$ now further decomposes into parts for $d\boldsymbol{B}$ and for $d\boldsymbol{K}$. Note that learning $\boldsymbol{B}$ by non-automatic cross-validation would not be possible, due to the large number of components.

Consider the case where we have many classes C, but not much data for most of them. We can postulate the assumptions that the behaviour of the C class functions $u_c(\cdot)$ is represented by $p \ll C$ underlying latent factors, which are then modelled as independent. This is achieved in our framework by having a non-square mixing matrix $\boldsymbol{B} \in \mathbb{R}^{C,p}$ (the "factor loadings"). It is not hard to adapt our framework to this case, in which it is obviously necessary to learn $\boldsymbol{B}$ as hyperparameters from data. A related model in a Bayesian context was considered by Teh et al. (2005).

7.4.2 Uncertain Targets in Hierarchical Classification

Recall the hierarchical classification setup of Section 3. Suppose that for some patterns $\boldsymbol{x}_i$, the target is unknown, but we know that the path from its class to the root goes through an inner node p. Denote by L_p the set of leaf nodes of the subtree rooted at p, *i.e.* $L_p = \{p\}$ for a leaf node $p \in L$, and $L_0 = L$.

We can allow for such uncertain target information by using pseudo-targets $\tilde{y}_i \in \{1, \dots, P\}$. If $\tilde{y}_i \notin L$, it is the lowest inner node we are certain about. The corresponding likelihood factor is

$$\sum_{c \in L_{\tilde{y}_i}} P(y_i = c | \boldsymbol{u}_i).$$

Note that the log likelihood is not a concave function anymore, whenever $|L_{\tilde{y}_i}| > 1$, and in the presence of such factors, primary fitting is not a convex problem. However, an *expectation-maximization* (EM) (Dempster et al., 1977) approach can be used to deal with uncertain targets. Namely, in "E steps" we compute

$$q_{ic} \propto \mathrm{I}_{\{c \in L_{\tilde{y}_i}\}} P(y_i = c | \boldsymbol{u}_i)$$

for the current $u_c(\cdot)$, where $\boldsymbol{q}_i = (q_{ic})_c$ are distributions. "M steps" consists of Newton-Raphson iterations as before, but using $\sum_c q_{ic} \log P(y_i = c|\boldsymbol{u}_i)$ as log likelihood factors. To this end, we just have to replace the vector $\boldsymbol{y} \in \mathbb{R}^{nC}$ by $\boldsymbol{q}$. Importantly, we only used the properties $\mathbf{1}^T \boldsymbol{y}_i = 1,\ \boldsymbol{y}_i \geq \mathbf{0}$ above (but not that $y_{ic} \in \{0, 1\}$), which are true for $\boldsymbol{q}$ just as well.

7.4.3 Low Rank Approximations

Our generic kernel matrix representation is described in Appendix D.1. If the dataset size n is large, we may not be able to keep the correlation matrices $\boldsymbol{M}^{(l)}$ in memory, and MVM with them becomes prohibitively expensive. We can use standard low rank matrix approximations to deal with this problem (see also Section 7.3). Namely, suppose that $\boldsymbol{M}^{(l)}$ is approximated by $\boldsymbol{P}^{(l)} \boldsymbol{L}^{(l)} \boldsymbol{L}^{(l)T} \boldsymbol{P}^{(l)T}$, where $\boldsymbol{P}^{(l)}$ is a permutation matrix, and $\boldsymbol{L}^{(l)} \in \mathbb{R}^{n, d_l}$ for $d_l \ll n$. Denote $I_l \subset \{1, \dots, n\}$ the active set of size d_l. The approximation may be obtained by an *incomplete Cholesky factorization*[22] (ICF), which has the special property that only a small set of d_l columns of $\boldsymbol{M}^{(l)}$ (along with its diagonal) ever have to be evaluated (Bach and Jordan, 2002). In this case, $\boldsymbol{L}^{(l)}_{1 \dots d_l, \cdot}$ is the lower triangular Cholesky factor of $\boldsymbol{M}^{(l)}_{I_l} \in \mathbb{R}^{d_l, d_l}$, so that $\boldsymbol{P}^{(l)T} \boldsymbol{M}^{(l)}_{\cdot, I_l} = \boldsymbol{L}^{(l)} \boldsymbol{L}^{(l)}_{1 \dots d_l, \cdot}{}^T$. Note that in the ICF case, we have that

$$\operatorname{diag}\left(\boldsymbol{P}^{(l)T} \boldsymbol{M}^{(l)} \boldsymbol{P}^{(l)} - \boldsymbol{L}^{(l)} \boldsymbol{L}^{(l)T}\right) \geq \mathbf{0}$$

pointwise, because the elements are simply the squared pivots for a potential continuation of the factorization (which has been stopped after d_l steps). Therefore, we can correct the approximation by replacing the diagonal of $\boldsymbol{L}^{(l)} \boldsymbol{L}^{(l)T}$ by the true one, ending up with the approximation

$$\boldsymbol{M}^{(l)} \approx \tilde{\boldsymbol{M}}^{(l)} := (\operatorname{diag}^2 \boldsymbol{M}^{(l)}) + \boldsymbol{P}^{(l)} \left(\boldsymbol{L}^{(l)} \boldsymbol{L}^{(l)T} - (\operatorname{diag}^2 \boldsymbol{L}^{(l)} \boldsymbol{L}^{(l)T})\right) \boldsymbol{P}^{(l)T}.$$

Snelson and Ghahramani (2006) motivate this diagonal correction in another context. It is clear that MVM with $\tilde{\boldsymbol{M}}^{(l)}$ can be done in $O(n\, d_l)$.

22. Matlab code for ICF (in the form required here) can be downloaded from `http://www.kyb.tuebingen.mpg.de/bs/people/seeger/software.html`.

If $*$ indexes test points different from the training points, then the test-training correlation matrix is

$$\boldsymbol{M}^{(l)}_{*,\cdot} = \boldsymbol{M}^{(l)}_{*,I}(\boldsymbol{L}^{(l)}_{1\ldots d_l,\cdot})^{-T}\boldsymbol{L}^{(l)T}\boldsymbol{P}^{(l)T}.$$

We can also learn parameters of the $M^{(l)}$ functions in this low rank setup by gradient-based optimization, assuming that the choice of I_l does not depend on these kernel parameters, but this is not discussed here.

Acknowledgments

We would like to thank Olivier Chapelle for many useful discussions on approximations of cross-validation optimization, and Peter Gehler for help with preprocessing the WIPO-alpha data, furthermore L. Cai and T. Hofmann for providing us with counts and dictionary data. Dilan Görur gave helpful comments on the NIPS paper related to this work. Supported in part by the IST Programme of the European Community, under the PASCAL Network of Excellence, IST-2002-506778.

Appendix A. Notation

In this section, we describe the notation used in this paper. We denote vectors and matrices by bold-face lower-case and upper-case letters, scalars and scalar functions are set normally. Subscripts select parts of objects, they can be single indexes or index sets. For example, $\boldsymbol{a} = (a_i)_i$ is a vector with components a_i, $\boldsymbol{A} = (a_{i,j})_{i,j}$ a matrix with entries $a_{i,j}$. We also write $\boldsymbol{a} = (a_i)$, $\boldsymbol{A} = (a_{i,j})$ if the indexes are clear from context. $\boldsymbol{A}_{\cdot,i}$ is the i-th column of $\boldsymbol{A}$ ("$\cdot$" is short for the full index set). $\otimes$ denotes the Kronecker product, $\boldsymbol{A} \otimes \boldsymbol{B} = (a_{i,j}\boldsymbol{B})_{i,j}$, $\mathbf{1}$ ($\mathbf{0}$) the vector of all ones (vector/matrix of all zeros), $\boldsymbol{I}$ the identity matrix, and $\boldsymbol{\delta}_j = (\mathrm{I}_{\{i=j\}})_i$ (columns of $\boldsymbol{I}$). For a matrix $\boldsymbol{A}$, $\operatorname{diag}\boldsymbol{A} = (a_{i,i})_i$ extracts the diagonal. For a vector $\boldsymbol{v}$, $\operatorname{diag}\boldsymbol{v}$ is the corresponding diagonal matrix. We also use this for matrix-valued vectors, an example is the diagonal kernel matrix $\boldsymbol{K} = \operatorname{diag}(\boldsymbol{K}^{(c)})_c$ in flat classification.

Many vectors and matrices are indexed by datapoints (i) and classes (c) at the same time, for example $\boldsymbol{u} = (u_{ic}) \in \mathbb{R}^{nC}$. We use double indexes ic for these, which are flattened as $i+(c-1)n$, so the component ordering[23] is $\boldsymbol{u} = (u_{11}, u_{21}, \ldots, u_{n1}, u_{12}, \ldots)$. In this context, selection index sets I are applied to the i (datapoint) index only: $\boldsymbol{u}_I = (u_{ic})_{i\in I,c} \in \mathbb{R}^{|I|C}$. Kronecker product notation works nicely with this double index convention. If $\boldsymbol{A} \otimes \boldsymbol{B}$ is applied to $\boldsymbol{u}$, $\boldsymbol{A}$ has C, $\boldsymbol{B}$ n columns. We frequently use $(\mathbf{1}^T \otimes \boldsymbol{I})\boldsymbol{u} = \sum_c \boldsymbol{u}^{(c)}$, where $\boldsymbol{u}^{(c)} = (u_{ic})_i \in \mathbb{R}^n$, or $(\mathbf{1} \otimes \boldsymbol{I})\boldsymbol{v}$ for $\boldsymbol{v} \in \mathbb{R}^n$, which stacks $\boldsymbol{v}$ on top of each other C times. The matrix $\boldsymbol{P}_{cls} = (\mathbf{1} \otimes \boldsymbol{I})(\mathbf{1}^T \otimes \boldsymbol{I})$ (introduced in Section 5.1) combines these operations:

$$\boldsymbol{P}_{cls}\boldsymbol{x} = \left.\begin{pmatrix} \bar{\boldsymbol{x}} \\ \bar{\boldsymbol{x}} \\ \vdots \end{pmatrix}\right\} C, \quad \bar{\boldsymbol{x}} = \sum_c \boldsymbol{x}^{(c)},$$

and $\boldsymbol{P}_{data}$ does the same, but operating on the i rather than the c index.

23. In `Matlab`, `reshape(u,n,C)` would give a matrix in $\mathbb{R}^{n,C}$.

All major notational definitions are listed in Table 3 for reference. For kernel matrices (for example, $\boldsymbol{K}^{(c)}$), we do not list the kernel functions (here: $K^{(c)}$), and for evaluation vectors (for example, $\boldsymbol{u}$), we do not list the underlying functions (here: $u^{(c)}(\cdot)$).

n	Number datapoints	2	LCG	Linear Conjugate Gradients	2
C	Number classes	2	P	Number nodes (hierarchy)	3
$\boldsymbol{y}$	Targets (zero-one)	2	L	Leaf nodes (hierarchy)	3
$\boldsymbol{x}_i$	Input points	2	$\breve{\boldsymbol{u}}$	Latent output (before mixing)	3
$\boldsymbol{u}$	Latent output (after mixing)	2	$\boldsymbol{\Phi}$	Hierarchy mixing matrix	3
$\boldsymbol{b}$	Intercepts	2	$\breve{\boldsymbol{K}}$	Kernel matrix (before mixing)	(2)
σ^2	Penalizing constant for $\boldsymbol{b}$	2	I_k, J_k	Partitions for CV criterion	4
Φ	Criterion for primary fitting	2	Ψ	CV criterion	4
$\boldsymbol{\alpha}$	Dual variables	2	q	Number of folds	4
$\boldsymbol{K}$	Kernel matrix (after mixing)	2	$\boldsymbol{h}$	Hyperparameters	4
$\boldsymbol{K}^{(c)}$	Kernel matrix block	2	k_3	Complexity parameter	4
$\tilde{\boldsymbol{K}}$	Kernel matrix ($\boldsymbol{b}$ eliminated)	2	$\boldsymbol{g}, \boldsymbol{W}$	Gradient, Hessian Φ_{lh}	5.1
$\boldsymbol{l}$	Logsumexp vector	(1)	$\boldsymbol{P}_{cls}$	Sum-distribute matrix	5.1
k_1, k_2	Complexity parameters	2	$\boldsymbol{P}_{data}$	Sum-distribute matrix	5.1
NR	Newton-Raphson	2	$\boldsymbol{E}, \boldsymbol{F}$	Accumulation matrices	5.2

Table 3: Reference for notational definitions. k: Section of definition; (k): Equation of definition.

Appendix B. Details for Primary Fitting Algorithm

In this section, we discuss further details of the primary fitting algorithm of Section 2, in addition to Section 5.1.

We need to counter the problem that roundoff errors may lead to numerical instabilities. The criterion we minimize is strictly convex, even if the kernel matrix $\boldsymbol{K}$ is singular (or nearly so). However, problems could arise from components in $\boldsymbol{\pi}$ becoming very small. Recall that $\log \pi_{ic} = u_{ic} - l_i$. We make use of a threshold $\kappa < 0$ and define

$$I = \{(i,c) \mid \log \pi_{ic} < \kappa,\ y_{ic} > 0\}, \quad I_0 = \{(i,c) \mid \log \pi_{ic} < \kappa,\ y_{ic} = 0\}.$$

The indices in I can be problematic due to the corresponding component $\tilde{g}_{ic} \approx y_{ic}/\pi_{ic}^{1/2}$ becoming large. Note that this happens only if $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ is a strong outlier w.r.t. the current predictor. Now, from the system (6) we see that $\boldsymbol{D}^{1/2}\boldsymbol{\beta} = \boldsymbol{D}\boldsymbol{A}^T(\boldsymbol{u} - \tilde{\boldsymbol{K}}\boldsymbol{\alpha}') - \boldsymbol{g}$. Therefore, if $(i,c) \in I$, then $(\boldsymbol{D}^{1/2}\boldsymbol{\beta})_{ic} \approx -g_{ic} \approx y_{ic}$. The idea is to solve the reduced system on the components in $\setminus I$ for $(\boldsymbol{D}^{1/2}\boldsymbol{\beta})_{\setminus I}$ and to plug in $(\boldsymbol{D}^{1/2}\boldsymbol{\beta})_I = \boldsymbol{y}_I$. Finally, within $\setminus I$, the components in I_0 may be problematic when computing the starting value $\boldsymbol{\beta} = \boldsymbol{D}^{-1/2}\boldsymbol{\alpha}$ for the CG run. However, in this case $\tilde{g}_{ic} \approx 0$, leading to $\beta_{ic} \approx 0$ from Eq. 6. The corresponding components in the starting value $\boldsymbol{\beta}$ can therefore be set to zero.

Next, the LCG algorithm for solving systems of the form (6) needs to be preconditioned. Suppose we want to solve $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$. If we have an approximation $\tilde{\boldsymbol{A}}$ to $\boldsymbol{A}$ so that $\boldsymbol{v} \mapsto \tilde{\boldsymbol{A}}^{-1}\boldsymbol{v}$

can be computed efficiently (essentially in linear time in the size of $\boldsymbol{v}$), the preconditioned CG algorithm solves the system $\tilde{\boldsymbol{A}}^{-1}\boldsymbol{A}\boldsymbol{x} = \tilde{\boldsymbol{A}}^{-1}\boldsymbol{b}$ instead. The idea is that $\tilde{\boldsymbol{A}}^{-1}\boldsymbol{A}$ typically has a lower condition number than $\boldsymbol{A}$, and LCG converges faster and less erratically. Our implementation does preconditioning with the diagonal of the system matrix $\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}$. Note that $\boldsymbol{V}\boldsymbol{\delta}_{ic} = \pi_{ic}^{1/2}(\boldsymbol{\delta}_c - \boldsymbol{\pi}_i) \otimes \boldsymbol{\delta}_i$, so that

$$\left(\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}\right)_{ic,ic} = 1 + \pi_{ic}\left((1 - 2\pi_{ic})(K_i^{(c)} + \sigma^2) + \sum_{c'} \pi_i^{(c')\,2}(K_i^{(c')} + \sigma^2)\right).$$

Therefore, the diagonal can be computed based on the diag $\boldsymbol{K}^{(c)}$ vectors. If the joint kernel matrix $\boldsymbol{K}$ is not block-diagonal (as in hierarchical classification, see Section 3), diag $\boldsymbol{K}$ is not sufficient for computing the system matrix diagonal. Let $\boldsymbol{v} \in \mathbb{R}^{nC}$ be defined via $\boldsymbol{v}_i = \boldsymbol{K}_i\boldsymbol{\pi}_i$, where $\boldsymbol{K}_i = (\boldsymbol{I} \otimes \boldsymbol{\delta}_i^T)\boldsymbol{K}(\boldsymbol{I} \otimes \boldsymbol{\delta}_i) \in \mathbb{R}^{C,C}$. Then, the system matrix diagonal has elements

$$1 + \pi_{ic}\left(\boldsymbol{K}_{ic} + \sigma^2 - 2w_{ic} + \boldsymbol{\pi}_i^T\boldsymbol{w}_i\right), \quad \boldsymbol{w} = \boldsymbol{v} + \sigma^2\boldsymbol{\pi}.$$

Appendix C. Solving Systems Exactly

In this section, we show how to implement our flat multi-class scheme using exact rather than approximate solutions of linear systems, yet still scaling linearly in C (at present, we do not know how to implement hierarchical classification exactly with such scaling).

For a Newton step, we need to solve $(\boldsymbol{I} + \boldsymbol{W}\tilde{\boldsymbol{K}})\boldsymbol{\alpha}' = \boldsymbol{r}$ with $\boldsymbol{W} = \boldsymbol{D} - \boldsymbol{D}\boldsymbol{P}_{cls}\boldsymbol{D}$. This can be written as

$$\left(\boldsymbol{A} - \boldsymbol{U}\boldsymbol{V}^T\right)\boldsymbol{D}^{-1/2}\boldsymbol{\alpha}' = \boldsymbol{D}^{-1/2}\boldsymbol{r}, \quad \boldsymbol{A} = \boldsymbol{I} + \boldsymbol{D}^{1/2}\tilde{\boldsymbol{K}}\boldsymbol{D}^{1/2},$$
$$\boldsymbol{U} = \boldsymbol{D}^{1/2}(\boldsymbol{1} \otimes \boldsymbol{I}), \quad \boldsymbol{V} = (\boldsymbol{A} - \boldsymbol{I})\boldsymbol{U}.$$

We now use the Sherman-Morrison-Woodbury formula together with the fact that $\boldsymbol{U}^T\boldsymbol{U} = \sum_c \boldsymbol{D}^{(c)} = \boldsymbol{I}$ to obtain

$$\boldsymbol{\alpha}' = \boldsymbol{D}^{1/2}\left(\boldsymbol{A}^{-1} + \boldsymbol{A}^{-1}\boldsymbol{U}(\boldsymbol{U}^T\boldsymbol{A}^{-1}\boldsymbol{U})^{-1}\boldsymbol{U}^T(\boldsymbol{I} - \boldsymbol{A}^{-1})\right)\boldsymbol{D}^{-1/2}\boldsymbol{r}. \tag{9}$$

We used that $\boldsymbol{V}^T\boldsymbol{A}^{-1} = \boldsymbol{U}^T(\boldsymbol{I} - \boldsymbol{A}^{-1})$. Note that $\boldsymbol{A}^{-1}$ is block-diagonal, and that

$$\boldsymbol{U}^T\boldsymbol{A}^{-1}\boldsymbol{U} = \sum_c \boldsymbol{D}^{(c)1/2}\boldsymbol{A}^{(c)-1}\boldsymbol{D}^{(c)1/2}.$$

We maintain Cholesky factors of all $\boldsymbol{A}^{(c)}$, as well as the Cholesky decomposition $\boldsymbol{U}^T\boldsymbol{A}^{-1}\boldsymbol{U} = \boldsymbol{R}\boldsymbol{R}^T$ (where $\boldsymbol{A}^{(c)-1}$ are obtained from the Cholesky factors).

For hyperparameter learning, we consider the partitions (I, J) sequentially. Since the $\boldsymbol{\alpha}_{[J]}$ are different across folds, we cannot obtain the $\boldsymbol{A}_{[J]}$, $\boldsymbol{H}_{[J]}$ as parts of underlying common matrices. Recall Section 5.2. $\boldsymbol{\alpha}_{[J]} + \boldsymbol{g}_{[J]} = \boldsymbol{0}$ gives $\boldsymbol{H}_{[J]}(d\boldsymbol{\alpha}_{[J]}) = -\boldsymbol{W}_{[J]}(d\boldsymbol{K}_J)\boldsymbol{\alpha}_{[J]}$. With

$$\boldsymbol{f} = \boldsymbol{I}_{\cdot,I}(\boldsymbol{\pi}_{[I]} - \boldsymbol{y}_I) - \boldsymbol{I}_{\cdot,J}\boldsymbol{W}_{[J]}\boldsymbol{H}_{[J]}^{-T}\tilde{\boldsymbol{K}}_{J,I}(\boldsymbol{\pi}_{[I]} - \boldsymbol{y}_I),$$

we have that $d\Psi_I = (\boldsymbol{I}_{\cdot,J}\boldsymbol{\alpha}_{[J]})^T(d\boldsymbol{K})\boldsymbol{f}$. Again, these vectors are accumulated in matrices $\boldsymbol{E}$, $\boldsymbol{F}$. Solving a system with $\boldsymbol{H}_{[J]}^T$ is an obvious variant of the procedure discussed above.

Appendix D. Further Details of the Implementation

Our implementation is designed to be as efficient as possible, while still being general and easy to extend to novel situations. This is achieved mainly by breaking down the problems to calling sequences of MVM primitives. These are then reduced to large numerical linear algebra primitives, where matrices are organized contiguously in memory, in order to exploit modern caching architectures (see Section 7.2).

D.1 A Generic Kernel Matrix Representation

A kernel matrix representation is some data structure which allows to compute kernel matrix MVMs $\boldsymbol{v} \mapsto \boldsymbol{K}^{(c)}\boldsymbol{v}$ efficiently, being the principal primitives of our primary fitting method. Further requirements arise if additional features of our framework are used. For example, if hyperparameters are to be learned as well, derivative MVMs $\boldsymbol{v} \mapsto (\partial \boldsymbol{K}^{(c)}/\partial h_p)\boldsymbol{v}$ are required as well, and "covariance shuffling" should be possible (see Section 5.2).

An efficient representation depends strongly on the covariance function used, and also on whether kernel matrix MVMs are approximated rather than computed exactly. For example, for linear kernels a special representation is used (see Appendix D.3). In this section, we describe a generic representation, which is part of our implementation.

The generic representation can be used with any covariance function, in that no special structure is assumed. It requires kernel matrices to be stored explicitly, which may not be possible for very large n. In general, we allow for different covariance functions $K^{(c)}$ for each class c, although sharing of kernels is supported, in that $K^{(c)}(\cdot,\cdot) = v_c M^{(l_c)}(\cdot,\cdot)$ and $v_c > 0$. Here, $l_c = l_{c'}$ is allowed for $c \neq c'$. The matrices $\boldsymbol{M}^{(l)}$ are stored explicitly. Note that the flexibility of using different variance parameters v_c with the same $\boldsymbol{M}^{(l)}$ does come at no extra cost, except for the fact that these have to be adjusted individually.

Since the $\boldsymbol{M}^{(l)}$ are symmetric, two can be stored each in a $n \times n$ block, say the odd-numbered ones in the lower triangles. Here, the diag $\boldsymbol{M}^{(l)}$ are stored separately, and whenever a specific $\boldsymbol{M}^{(l)}$ is required explicitly, the diagonal is copied into the block. It is important to note that the BLAS (see Section 7.2) directly supports symmetric matrices which are stored in the lower or upper triangle of a rectangular block.

The reader may wonder whether space could be saved by storing intermediates of the $\boldsymbol{M}^{(l)}$ instead. For example, if the $M^{(l)}$ are isotropic kernels of the form $f^{(l)}(\|\boldsymbol{x} - \boldsymbol{x}'\|)$, we could store the inner product matrix $(\boldsymbol{x}_i^T \boldsymbol{x}_j)_{i,j}$ only. In practice, this turns out to be significantly slower (by a factor), the reason being that the optimized BLAS primitives are many times more efficient than applying a non-linear function $f^{(l)}$ pointwise to a matrix, even if the matrix is stored contiguously. For the same reason, computing MVMs on the fly without storing matrices is even more costly.

D.2 Shuffling the Kernel Matrix Representation

Covariance matrix shuffling has been motivated in Section 5.2. It is required during hyperparameter optimization, because the MVM primitives for submatrices $\boldsymbol{K}_{J_k}$ have to be driven by a single representation of the complete $\boldsymbol{K}$ (note that each $\boldsymbol{K}_{J_k}$ is of size $n(q-1)/q$, thus almost as large as $\boldsymbol{K}$). A simple approach would be to use subindexed matrix-vector multiplication code, but this is very inefficient (usually more than one order of magnitude slower than the flat BLAS functions).

Instead, when dealing with fold k, we shuffle the representation so that $\boldsymbol{K}_{J_k}$ moves to the upper left corner of the matrix. How this is done, depends on the representation. In this context, it is important to note that the underlying BLAS explicitly allows working on submatrices within upper left corners of larger frames, with virtually no loss in efficiency[24]. In the generic representation of Appendix D.1, we simply permute the kernel matrices $\boldsymbol{K}^{(c)}$ using the index (J_k, I_k). A corresponding de-shuffling operation has to restore the old representation for $\boldsymbol{K}$.

D.3 The Linear Kernel

Our application described in Section 6.1 uses the linear kernel $K^{(c)}(\boldsymbol{x}, \boldsymbol{x}') = v_c \boldsymbol{x}^T \boldsymbol{x}'$, where $\boldsymbol{x}$ is very high-dimensional (*e.g.*, word counts over a dictionary), but also extremely sparse (by far the most entries are zero). The linear kernel fits the setup of Appendix D.1 with a single $\boldsymbol{M}^{(1)} = \boldsymbol{X}\boldsymbol{X}^T$, where $\boldsymbol{X} \in \mathbb{R}^{n,d}$ is the design matrix. $\boldsymbol{X}$ is very sparse, and in our implementation is represented using a standard sparse matrix format.

An MVM is done as $\boldsymbol{v} \mapsto v_c(\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{v})$, where $\boldsymbol{X}$ is sparse. More generally, we do $\boldsymbol{S} \mapsto \boldsymbol{X}\boldsymbol{X}^T\boldsymbol{S}$ with large matrices $\boldsymbol{S}$. Kernel matrix shuffling (Appendix D.2) is implemented by simply reordering the non-zero positions for $\boldsymbol{X}$. In this context, it is interesting to remark a finding which underlines the arguments in Section 7.2. The sparse matrix format is such that $\boldsymbol{X}\boldsymbol{X}^T\boldsymbol{S}$ is reduced to so-called *daxpy* operations ($\boldsymbol{a} = \boldsymbol{a} + \alpha\boldsymbol{b}$) on the *rows* of $\boldsymbol{S}$. By Fortran (and BLAS) convention, $\boldsymbol{S}$ is stored in column-order, so that rows can only be accessed directly by using a striding value > 1 (the distance between consecutive vector elements in memory). We added a simple trick (called *dimension flipping*) to the implementation, which in essence switches our default ordering of $C\,n$ vectors $\boldsymbol{v} = (v_{11}, v_{21}, v_{31}, \dots)^T$ to $(v_{11}, v_{12}, v_{13}, \dots)^T$ before major kernel MVM computations are done. This simple modification led to a direct five-times speedup, which underlines the importance of contiguous memory access in the bottleneck computations of a method (which allows optimal usage of cache hierarchies).

References

F. Bach and M. Jordan. Kernel independent component analysis. *Journal of Machine Learning Research*, 3:1–48, 2002.

P. Bartlett and A. Tewari. Sparseness vs estimating conditional probabilities: Some asymptotic results. In *Conference on Computational Learning Theory 17*, pages 564–578. Springer, 2004.

D. Bertsekas. *Nonlinear Programming.* Athena Scientific, 2nd edition, 1999.

S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2002.

24. Matrices in BLAS are stored column-wise, each column has to be contiguous in memory, but the *striding value*, *i.e.* the offset in memory required to jump to the next column, can be larger than the number of rows.

L. Cai and T. Hofmann. Hierarchical document categorization with support vector machines. In *CIKM 13*, pages 78–87, 2004.

K. Crammer and Y. Singer. On the algorithmic implementation of multiclass kernel-based vector machines. *Journal of Machine Learning Research*, 2:265–292, 2001.

P. Craven and G. Wahba. Smoothing noisy data with spline functions: Estimating the correct degree of smoothing by the method of generalized cross-validation. *Numerische Mathematik*, 31:377–403, 1979.

L. Csató. *Gaussian Processes — Iterative Sparse Approximations.* PhD thesis, Aston University, Birmingham, UK, March 2002.

S. Della Pietra, V. Della Pietra, and J. Lafferty. Inducing features of random fields. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 19(4):380–393, 1997.

A. Dempster, N. Laird, and D. Rubin. Maximum likelihood from incomplete data via the EM algorithm. *Journal of Roy. Stat. Soc. B*, 39:1–38, 1977.

M. Gibbs. *Bayesian Gaussian Processes for Regression and Classification.* PhD thesis, University of Cambridge, 1997.

P.J. Green and B. Silverman. *Nonparametric Regression and Generalized Linear Models.* Monographs on Statistics and Probability. Chapman & Hall, 1994.

C.-W. Hsu and C.-J. Lin. A comparison of methods for multi-class support vector machines. *IEEE Transactions on Neural Networks*, 13:415–425, 2002.

S. Keerthi and D. DeCoste. A modified finite Newton method for fast solution of large scale linear SVMs. *Journal of Machine Learning Research*, 6:341–361, 2005.

S. Keerthi, V. Sindhwani, and O. Chapelle. An efficient method for gradient-based adaptation of hyperparameters in SVM models. In Schölkopf et al. (2007).

J. Lafferty, Y. Liu, and X. Zhu. Kernel conditional random fields: Representation, clique selection, and semi-supervised learning. Technical Report CMU-CS-04-115, Carnegie Mellon University, 2004.

Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1: 541–551, 1989.

P. McCullach and J.A. Nelder. *Generalized Linear Models.* Number 37 in Monographs on Statistics and Applied Probability. Chapman & Hall, 1st edition, 1983.

M. Opper and O. Winther. Gaussian processes for classification: Mean field algorithms. *Neural Computation*, 12(11):2655–2684, 2000.

B. Pearlmutter. Fast exact multiplication by the Hessian. *Neural Computation*, 6(1):147–160, 1994.

J. Platt. Fast training of support vector machines using sequential minimal optimization. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods: Support Vector Learning*, pages 185–208. MIT Press, 1998.

Y. Qi, T. Minka, R. Picard, and Z. Ghahramani. Predictive automatic relevance determination by expectation propagation. In C. Brodley, editor, *International Conference on Machine Learning 21*. Morgan Kaufmann, 2004.

C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006.

Y. Saad. *Iterative Methods for Sparse Linear Systems*. International Thomson Publishing, 1st edition, 1996.

B. Schölkopf and A. Smola. *Learning with Kernels*. MIT Press, 1st edition, 2002.

B. Schölkopf, J. Platt, and T. Hofmann, editors. *Advances in Neural Information Processing Systems 19*, 2007. MIT Press.

M. Seeger. *Bayesian Gaussian Process Models: PAC-Bayesian Generalisation Error Bounds and Sparse Approximations*. PhD thesis, University of Edinburgh, July 2003. See `www.kyb.tuebingen.mpg.de/bs/people/seeger`.

M. Seeger. Gaussian processes for machine learning. *International Journal of Neural Systems*, 14(2):69–106, 2004.

M. Seeger. Cross-validation optimization for large scale hierarchical classification kernel methods. In Schölkopf et al. (2007), pages 1233–1240.

B. Shahbaba and R. Neal. Improving classification when a class hierarchy is available using a hierarchy-based prior. *Bayesian Analysis*, 2(1):221–238, 2007.

Y. Shen, A. Ng, and M. Seeger. Fast Gaussian process regression using KD-trees. In Weiss et al. (2006).

E. Snelson and Z. Ghahramani. Sparse Gaussian processes using pseudo-inputs. In Weiss et al. (2006).

Y.-W. Teh, M. Seeger, and M. I. Jordan. Semiparametric latent factor models. In Z. Ghahramani and R. Cowell, editors, *Workshop on Artificial Intelligence and Statistics 10*, 2005.

G. Wahba. *Spline Models for Observational Data*. CBMS-NSF Regional Conference Series. Society for Industrial and Applied Mathematics, 1990.

Y. Weiss, B. Schölkopf, and J. Platt, editors. *Advances in Neural Information Processing Systems 18*, 2006. MIT Press.

C. Williams and C. Rasmussen. Gaussian processes for regression. In D. Touretzky, M. Mozer, and M. Hasselmo, editors, *Advances in Neural Information Processing Systems 8*. MIT Press, 1996.

C. K. I. Williams and D. Barber. Bayesian classification with Gaussian processes. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(12):1342–1351, 1998.

C. Yang, R. Duraiswami, and L. Davis. Efficient kernel machines using the improved fast Gauss transform. In L. Saul, Y. Weiss, and L. Bottou, editors, *Advances in Neural Information Processing Systems 17*, pages 1561–1568. MIT Press, 2005.