Cross-Validation Optimization for Structured Hessian Kernel Methods

Matthias Seeger
Max Planck Institute for Biological Cybernetics
P.O. Box: 21 69
72012 Tübingen, Germany
seeger@tuebingen.mpg.de

December 1, 2006

Abstract

We propose a highly efficient framework for kernel multi-class models with a large and structured set of classes, and more general for penalized likelihood kernel methods. 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 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 directing the focus of code optimization to these primitives.

Kernel parameters are learned automatically by maximizing the cross-validation log likelihood, and predictive probabilities are estimated. We demonstrate our approach on large scale text classification tasks with hierarchical structure on throusands of classes, achieving state-of-the-art results in an order of magnitude less time than previous work. We also discuss an extension to label sequence learning with kernel conditional random fields.

1 Introduction

In recent years, Machine Learning researchers started to address problems with kernel machines which cannot be represented reasonably with standard models involving a single latent function. These demand to fit a model with a large number of dependent variables to a training sample with very many cases. For example, for multi-way classification models with a hierarchically structured label space [4], modern applications demand predictions on thousands of classes, and very large datasets become available. If n and C denote dataset size and number of classes respectively, nonparametric kernel methods like SVMs or Gaussian processes typically scale superlinearly in $n\,C$, if dependencies between the latent class functions are properly represented.

The prediction of label sequences or other label structures can be seen as multi-way classification with a label space which is exponentially large, yet has a specific underlying structure

[12]. Looking at such models from another angle (as is propagated in this paper) reveals that there exist still only a polynomial number of latent variables, but it is their dependence relations which involve sums of exponential size. However, in such structured label models, we still have to deal with very many variables, with the added complexity of an involved coupling of these variables through a likelihood or loss function.

Furthermore, most large scale kernel methods proposed so far refrain from solving the problem of learning hyperparameters (kernel or loss function parameters). The user has to run cross-validation schemes, which require frequent human interaction and are not suitable for learning more than a few hyperparameters. However, novel applications demand fitting many hyperparameters through gradient-based optimization.

In this paper, we propose a general framework for learning in probabilistic kernel classification models. While the basic models are standard, 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 (such as complete or partial dualization). In this paper, we focus on the opposite approach of trying to find directions of descent which lead to fast descent, no matter how many of the variables are involved. The culmination of this idea is Newton's method, and a main objective of this paper is to show how approximative Newton directions can be found 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. This approach is generally favoured in the Optimization community for problems such as kernel methods fitting, which do not come with a natural decomposition into parts. We discuss this point in more detail in Section 9.2.

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, as will be demonstrated.

We show how to choose hyperparameters *automatically* by minimizing 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 fitting method is done by gradient-based optimization, where the bulk of work required to compute the gradient does not scale with the number of hyperparameters at all. Therefore, our approach can be used to learn a large number of hyperparameters and does not need 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 classification with C classes would require. We test our framework on the same tasks treated in [4], achieving comparable results in at least an order of magnitude less time. We also motivate how our framework can be applied to label sequence learning with a kernel conditional random field [12].

Note that our proposal to use approximate Newton methods is certainly not novel. The Newton method, or a variant of it called *Fisher scoring*, is the standard approach for fitting generalized linear models in Statistics [9, 14]. Our primary fitting method for flat multi-way classification (see Section 3) appeared in [30]. However, we demonstrate that this principle can be superior even on a much larger scale, and show how model structure can be exploited in this context. Furthermore, we demonstrate how secondary tasks such as hyperparameter learning can be reduced to the same underlying "large step" primitives.

The structure of the paper is as follows. The general structure of our framework is detailed in Section 2, motivating the reductions to matrix-vector multiplications. Flat multi-way classification is discussed in Section 3, and the setup is extended to hierarchical classification in Section 4. We discuss hyperparameter learning by cross-validation log likelihood maximization in Section 5. Essential details for previous Sections are collected in Section 6, and further applications of the framework are motivated in Section 7. Experimental results for flat and hierarchical classification are given in Section 8. We close with a discussion in Section 9, where arguments explaining the superior efficiency of global direction methods are given.

Highly optimized C++ software for our framework is available as part of the *LHOTSE* toolbox for adaptive statistical models, which may be downloaded from `www.kyb.tuebingen.mpg.de/bs/people/seeger/lhotse/` and used for non-commercial purposes. The implementation contains the linear kernel case used in Section 8.2 (see Appendix D.2), as well as a generic representation described in Section 6.2, with which the experiments in Section 8.1 have been done. New kernels or kernel MVM implementations can be included easily.

2 Learning with Structured Hessian

In this Section, we describe the general structure of our framework, relating it to similar previous approaches.

We focus on probabilistic multi-way classification methods which are fitted to data by minimizing a penalized log likelihood criterion in what amounts to an unconstrained convex optimization. In nonparametric models, this minimization is done over latent *functions*, which play the same role as vector-valued random variables do in conventional parametric models. Penalization is done by adding quadratic regularizers to the log likelihood criterion, where the (positive definite) matrices of these terms come from applying a covariance (or kernel) function to the data. From a Bayesian viewpoint, these penalization terms come from Gaussian process priors on the latent functions. In this paper, we concentrate on *estimating* the latent functions in a robust way, rather than doing Bayesian inference over them. Our arguments extend to methods with nondifferentiable loss functions, such as *support vector machines* (SVM), which are usually understood in terms of constrained convex optimization.

These models are parameterized in terms of *primary parameters* $\boldsymbol{\alpha}$, being coefficients in kernel expansions of latent function estimates $u_c(\cdot)$, and *hyperparameters* $\boldsymbol{h}$ (kernel and noise model parameters). The general structure of models considered in this paper, is given in Figure 1. The specific meaning of variables becomes clear in concrete examples such as flat (Section 3) and hierarchical classification (Section 4). In general, a set of latent functions $\breve{u}_p(\cdot)$ with independent priors (or penalizers) is mapped to a set of latent functions $u_c(\cdot)$,

which introduces prior dependencies (or coupled penalizers) in general. This is referred to as *prior mixing*. The latent $u_c(\cdot)$ feed into a log likelihood, giving rise to *likelihood coupling*. These two types of coupling is what we refer to as *structure* in the model. On the other hand, the coupling between different values of $\breve{u}_p(\cdot)$ is unstructured and given by kernel matrices.

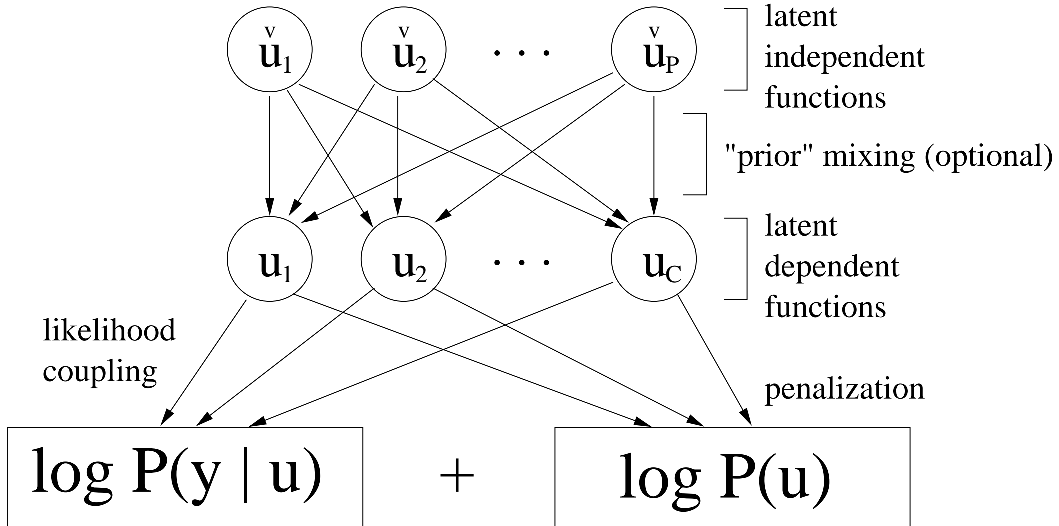


Figure 1: Structure of penalized likelihood optimization.

The term “fitting” is in general reserved for the minimization of the penalized likelihood criterion Φ w.r.t. $\boldsymbol{\alpha}$, for fixed $\boldsymbol{h}$. This can be done using the *Newton-Raphson* (NR) algorithm, in which we iteratively compute search directions and find new values of $\boldsymbol{\alpha}$ along these. Newton directions are computed by solving **L**inear **S**ystems $\boldsymbol{H}\boldsymbol{x} = \boldsymbol{r}$, where $\boldsymbol{H}$ is the **H**essian of Φ (LSH), evaluated at the current $\boldsymbol{\alpha}$.

Adjusting $\boldsymbol{h}$ to data is typically harder. The straightforward approach of minimizing Φ w.r.t. $\boldsymbol{h}$ as well fails, due to the “overfitting” problem, which states that valid generalization with reasonably complex models is not obtained by fitting finite data ever better. Observed data is noisy, and overfitting means that noise is represented along with signal, while generalization requires that noise is filtered out. *Cross-validation* (CV) is a general technique frequently used to adjust hyperparameters. It works by leaving out some part of the training data, fitting $\boldsymbol{\alpha}$ on the rest, then computing the log likelihood on the left-out data. This is done in turn for a partition of the training set, summing up the partial scores. This way, we test generalization fairly directly, detecting overfitting using the held-out data part which was not used for the fitting.

In the context of this paper, CV has the advantage of being easily reduced to fitting, and therefore to LSH. If the partition has q sets (or folds), we can compute the CV log likelihood by calling the fitting subroutine q times, working on subsets of the data. Interestingly, the gradient of the CV criterion can be computed analytically, requiring the solution of a linear system for each fold, with the system matrix being the Hessian of the fitting criterion, thus LSH. The CV criterion can be minimized using a gradient-based optimizer.

For the problems we consider here, the Hessian is typically orders of magnitude larger than what could be stored in memory, and solving LSH directly is not an option. We reduce LSH to *matrix-vector multiplication* (MVM) computations $\boldsymbol{s} \mapsto \boldsymbol{H}\boldsymbol{s}$, employing the *linear conjugate gradients* (LCG) method, well-known in numerical mathematics where very large sparse systems are solved that way [20]. LCG is also used fairly routinely in Machine Learning [8, 11]. This reduction is approximate, and its accuracy for a fixed running time does depend on specifics of the model in a complicated way. Nevertheless, the empirical fact that it is often accurate enough in practice, can be motivated along several directions, as will be done below for concrete applications.

MVM with $\boldsymbol{H}$ is further reduced to more elementary problems, using two different approaches. First, the coupling between the $\boldsymbol{\alpha}$ variables (in the example of multi-way classification, there are $n\,C$ of them) does have structure due to specific likelihood coupling, and this structure is inherited by the gradient and Hessian of Φ. Hessian structure depends on structures in Φ, and a canonical recipe cannot be given. Yet, the cases we consider here (logistic regression, label sequence learning) cover general instances likelihood coupling.

Second, the presence of quadratic penalization terms in Φ means that these "kernel matrices" turn up in the Hessian explicitly, so the computation of $\boldsymbol{H}\boldsymbol{s}$ does require computations of kernel MVMs $\boldsymbol{K}\boldsymbol{t}$. The structure of $\boldsymbol{K}$ is determined by prior coupling, and $\boldsymbol{K}$-MVM should be envisioned as consisting of post- and pre-computations coming from the prior coupling structure, with a kernel MVM application $\breve{\boldsymbol{K}}\boldsymbol{r}$ in between. Here, $\breve{\boldsymbol{K}}$ is block-diagonal, owing to the prior independence of the latent $\breve{u}_p(\cdot)$. In the applications considered here, these inner kernel MVMs require the vast majority of running time. For the CV gradient computation, we also require MVMs with the kernel derivative matrices $\partial\breve{\boldsymbol{K}}^{(c)}/\partial h_p$.

If kernel MVMs consume the majority of running time, then all effort of code optimization should be spent on these primitives, they may even be approximated themselves (see Section 9.3). In special cases, such as for the linear kernel (see Appendix D.2), kernel MVMs can be done much faster than in $O(n^2)$. Importantly, these computational benefits are the more pronounced, the *larger* the primary variables are, which is why we do *not* propose a method which accesses smaller chunks of kernel matrices. This point is discussed in more detail in Section 9.2. The step-wise reductions in our framework are shown schematically in Figure 2.

None of the methods we propose a combination of here, are entirely novel in Machine Learning, although most previous work has addressed different problems. In the remainder of this Section, we provide some general context.

Efficient Gradient Computation We have $\boldsymbol{H}(\boldsymbol{x})\boldsymbol{v} = \lim_{\varepsilon\to 0}\varepsilon^{-1}(\boldsymbol{g}(\boldsymbol{x}+\varepsilon\boldsymbol{v}) - \boldsymbol{g}(\boldsymbol{x}))$. If $\boldsymbol{g}(\cdot)$ can be computed very efficiently, so can be the Hessian-vector product, even exactly [16, 21]. These methods are useful for applying our framework to label sequence learning (see Section 7.1).

Sparse Hessian When $\boldsymbol{H}(\boldsymbol{x})$ is sparse, multiplying it by a vector is very efficient. Sparse Hessians may appear in kernel methods with compactly supported kernels or in graphical methods with factorization (conditional independence) assumptions. In the latter case, we often have *structured* sparsity which can be exploited even more directly than unstructured

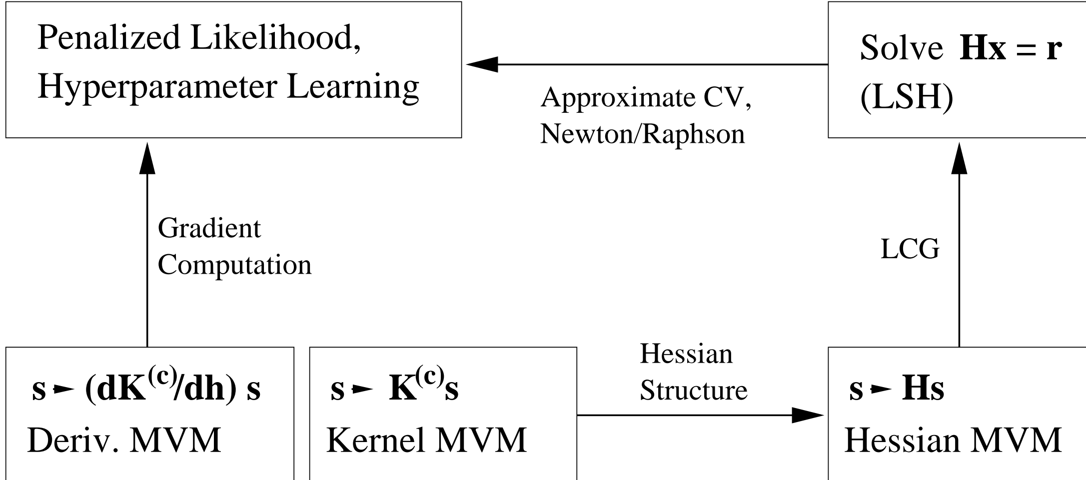


Figure 2: Reductions in our algorithm.

one. Our hierarchical classification application is an example of underlying block-diagonal structure.

Low-Rank Hessian If $\boldsymbol{H}(\boldsymbol{x})$ can be written as the sum of a diagonal matrix and a matrix of low rank, MVM can be done in $O(qr)$, r the small rank. In sparse approximations to kernel methods [24, 13], the kernel matrix is approximated as low-rank, and the likelihood gives rise to a diagonal matrix (see Section 9.3 and Section 7.4).

Fast Multipole and KD-Tree Methods For isotropic kernels, the MVM primitive with the kernel matrix can be approximated very efficiently (for low-dimensional input points) using specialized nearest neighbour data structures [31, 23] (see Section 9.3).

3 Penalized Multiple Logistic Regression

In this Section, we show how to apply our framework to multi-way classification with C classes, when structures between classes is not modelled. We refer to this setup as *flat classification*, in that the label set is flat.

Our problem is to predict $y \in \{1, \dots, C\}$ from $\boldsymbol{x} \in \mathcal{X}$, given some i.i.d. data $D = \{(\boldsymbol{x}_i, \boldsymbol{y}_i) \,|\, i = 1, \dots, n\}$. We use zero-one coding, *i.e.* $\boldsymbol{y}_i \in \{0,1\}^C$, $\mathbf{1}^T\boldsymbol{y}_i = 1$. We employ the *multiple logistic regression model*, consisting of C latent (unobserved) class functions u_c feeding into the multiple logistic (or softmax) likelihood $P(y_{i,c} = 1|\boldsymbol{x}_i, \boldsymbol{u}_i) = e^{u_c(\boldsymbol{x}_i)}/(\sum_{c'} e^{u_{c'}(\boldsymbol{x}_i)})$.

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

log likelihood

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

which we minimize for primary fitting. $\|\cdot\|_c$ is the RKHS norm for kernel $K^{(c)}$. As noted in Section 2, the model can also be understood in a Bayesian context, where the penalization terms come from Gaussian process priors on the functions f_c, and $\boldsymbol{b}$ has a Gaussian prior. From this viewpoint, we do a *maximum a-posteriori* (MAP) approximation here. Details on penalized likelihood kernel methods can be found in [9], the link to Gaussian process models is explained in [22].

Our notation for nC vectors[1] (and matrices) uses the ordering $\boldsymbol{y} = (y_{1,1}, y_{2,1}, \dots, y_{n,1}, y_{1,2}, \dots)$. We set $\boldsymbol{u} = (u_c(\boldsymbol{x}_i)) \in \mathbb{R}^{nC}$. $\otimes$ denotes the Kronecker product, $\mathbf{1}$ is the vector of all ones. Selection indexes I are applied to i only: $\boldsymbol{y}_I = (y_{i,c})_{i\in I,c} \in \mathbb{R}^{|I|C}$.

Since the likelihood depends on the f_c only through the values $f_c(\boldsymbol{x}_i)$, every minimizer of Φ must be a kernel expansion: $f_c = \sum_i \alpha_{i,c} K^{(c)}(\cdot, \boldsymbol{x}_i)$. This fact is known as "representer theorem", see [9, 27]. Plugging this in, the regularizer becomes

$$(1/2)\boldsymbol{\alpha}^T \boldsymbol{K}\boldsymbol{\alpha} + (1/2)\sigma^{-2}\|\boldsymbol{b}\|^2,$$

where $\boldsymbol{K}^{(c)} = (K^{(c)}(\boldsymbol{x}_i, \boldsymbol{x}_j))_{i,j} \in \mathbb{R}^{n,n}$, $\boldsymbol{K} = \operatorname{diag}(\boldsymbol{K}^{(c)})_c$ is block-diagonal. We refer to this setup as *flat classification* model.

We show in Section 6.1.1 that the b_c may be eliminated as $\boldsymbol{b} = \sigma^2(\boldsymbol{I} \otimes \mathbf{1}^T)\boldsymbol{\alpha}$. Thus, if $\tilde{\boldsymbol{K}} = \boldsymbol{K} + \sigma^2(\boldsymbol{I} \otimes \mathbf{1})(\boldsymbol{I} \otimes \mathbf{1}^T)$, then Φ becomes

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

Importantly, Φ is strictly convex in $\boldsymbol{\alpha}$, being a sum of linear, quadratic, and *logsumexp* terms (of the form $\log \mathbf{1}^T \exp(\boldsymbol{u}_i)$, see [2]), so it has a unique minimum point $\hat{\boldsymbol{\alpha}}$. The corresponding kernel expansions are

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

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 correct, although better estimates could be obtained by a more Bayesian treatment.

We note that this setup is related to the multi-class SVM [5], where $-\log P(y_i|\boldsymbol{u}_i)$ is replaced by the margin loss $-u_{y_i}(\boldsymbol{x}_i) + \max_c\{u_c(\boldsymbol{x}_i) + 1 - \delta_{c,y_i}\}$. 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.

As mentioned in Section 2, we minimize Φ using the Newton-Raphson (NR) algorithm. Details are provided in Section 6.1. The complexity of our fitting algorithm is dominated by $k_1(k_2 + 2)$ matrix-vector multiplications with $\boldsymbol{K}$, where k_1 is the number of NR iterations, k_2 the number of linear conjugate gradient (LCG) steps for computing each Newton direction. Since NR is a second-order convergent method, k_1 is generally small. k_2 determines

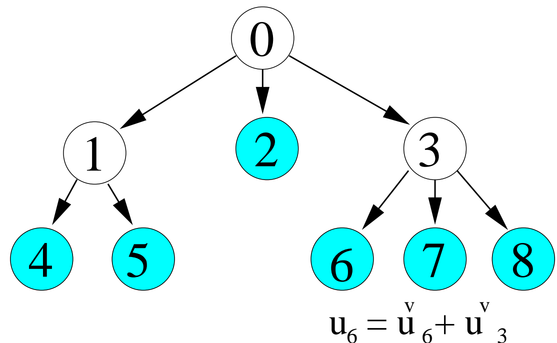
[1] In `Matlab`, `reshape(y,n,C)` would give the matrix $(y_{i,c}) \in \mathbb{R}^{n,C}$.

the quality of each Newton direction, and again, fairly small values seem sufficient (see Section 8.2). We conjecture that this is the case, because the coupling between the $\boldsymbol{\alpha}_c$ of different classes is fairly weak, given only through the normalization in the likelihood. This leads to Hessians which are close to block-diagonal, thus to fast convergence of LCG. This point is taken up at the end of Section 4.

4 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[2], 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.

In flat classification, the latent class functions u_c are modelled as *a priori* independent, in that the regularizer (which plays the role of a log prior) is a sum of individual terms for each u_c, without any interaction terms. *Analysis of variance* (ANOVA) models go beyond this independent design, they have previously been applied to text classification by [4] (although they did not call them by their name). 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. The class functions to be fed into the classification likelihood (which is the same as in Section 3) are the $u_{L(c)}$ of the leafs. This setup represents similarities conditioned on 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 regularization 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 related as $\boldsymbol{u} = (\boldsymbol{\Phi} \otimes \boldsymbol{I})\breve{\boldsymbol{u}}$, $\boldsymbol{\Phi} \in \{0,1\}^{P,P}$. Importantly, $\boldsymbol{\Phi}$ has a simple 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

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

because $u_{cs_p+j} = u_p + \breve{u}_{cs_p+j}$ for the functions of the children of p. Formally,

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

where the lower left $\boldsymbol{I} \in \mathbb{R}^{n_p,n_p}$. Note that $\boldsymbol{\Phi}$ is lower triangular with $\operatorname{diag}\boldsymbol{\Phi} = \boldsymbol{I}$. This recursive definition immediately implies Algorithm 1 for computing $\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{x}$. The MVM primitive $\boldsymbol{y} = \boldsymbol{\Phi}^T\boldsymbol{x}$ can be computed as shown in Algorithm 2. More details about the hierarchical case can be found in Section 6.4.

Algorithm 1 Matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{\Phi}\boldsymbol{x}$

$\boldsymbol{y} \leftarrow ()$. $y_0 := 0$.
for $p = 0, \dots, P$ **do**
  if $n_p > 0$ (p not a leaf node) **then**
    Let $J(p) = \{cs_p + 1, \dots, cs_p + n_p\}$.
    $\boldsymbol{y} \leftarrow (\boldsymbol{y}^T,\ y_p \mathbf{1}^T + \boldsymbol{x}_{J(p)}^T)^T$.
  end if
end for

Algorithm 2 Matrix-vector multiplication $\boldsymbol{y} = \boldsymbol{\Phi}^T\boldsymbol{x}$

$\boldsymbol{z} \leftarrow \boldsymbol{x}$.
for $p = P, \dots, 0$ **do**
  if $n_p > 0$ (p not a leaf node) **then**
    Let $J(p) = \{cs_p + 1, \dots, cs_p + n_p\}$, $J(<p) = \{1, \dots, cs_p\}$.
    $\boldsymbol{y}_{J(p)} \leftarrow \boldsymbol{z}_{J(p)}$.
    $\boldsymbol{z} \leftarrow \boldsymbol{z}_{J(<p)} + (\mathbf{1}^T \boldsymbol{z}_{J(p)})\boldsymbol{\delta}_p \in \mathbb{R}^{cs_p}$.
  end if
end for

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 $\boldsymbol{K}$,

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

into the flat classification model of Section 3. 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 the important point is that we are still able to do 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.

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 case (see Section 5). The cost of a kernel MVM is increased 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. The subtle issue is that the primary fitting becomes more costly, precisely because there is more coupling between the variables. In the flat case, the Hessian of

Φ 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, both LCG and NR need more iterations to attain the same accuracy, although the underlying primitive of multiplying with the Hessian comes at a cost close to what is required 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[3] for the system of Eq. 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.

Finally, it is interesting to observe that in our setup, the marginal prior distribution of $u_p(\cdot)$ is a zero-mean Gaussian process with a covariance function being the sum of $\breve{K}^{(p')}$ for nodes p' between p and the root (if our model is viewed in a Bayesian way, see Section 2). Much work has been done on combining elementary kernels in sums and learning the prefactors $v_{p'}$ (CITE!!!!). In our setup, there is a clear interpretation for such a combination. For example, if certain features of the covariate $\boldsymbol{x}$ are relevant only starting from certain levels of the class hierarchy, they may be included only in the corresponding kernels $\breve{K}^{(p)}$ at these levels. However, note that our setting is more powerful than simply assuming that the $u_p(\cdot)$ are *a priori* independent with kernels being sums of $\breve{K}^{(p')}$. In our case, different $u_p(\cdot)$ are actually *dependent* (as random variables) under the model, if they share an inner node other than the root.

Also, note that the $O(C\,n^3)$ variant of exact computation for the flat case (see Section 3) does not have an obvious counterpart in the hierarchical extension.

5 Hyperparameter Learning

As discussed in Section 2, our framework comes with an automatic method for setting free hyperparameters $\boldsymbol{h}$, by gradient-based maximization of the cross-validation (CV) log likelihood, calling primary fitting as a subroutine. Note that such nested strategies are commonplace in Bayesian Statistics, where (marginal) inference is typically used as subroutine for parameter learning.

Recall that primary fitting consists of minimizing Φ of Eq. 1 w.r.t. $\boldsymbol{\alpha}$. For CV, let $\{I_k\}$ be a partition of $\{1,\dots,n\}$, 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 primary criterion on the subset J_k of the data. Here, $\boldsymbol{u}_{[J_k]} = \tilde{\boldsymbol{K}}_{J_k}\boldsymbol{\alpha}_{[J_k]}$. The $\boldsymbol{\alpha}_{[J_k]}$

[3] So far, we explored two types of preconditioner $\tilde{\boldsymbol{A}}$ with negative results: one of tensor product structure supposedly taylored to the coupling structure for hierarchical classification (see Section 4), the other being a low rank approximation to $\boldsymbol{A}$ obtained by an incomplete Cholesky factorization. The former did not lead to faster LCG convergence, while the latter requires precomputations which are often more costly than solving the system (with diagonal preconditioning) in the first place.

are independent variables, *not* part of a common[4] $\boldsymbol{\alpha}$. The CV 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 $\boldsymbol{\alpha}_{[J_k]}$ minimizes Φ_{J_k}. Since for each k, we fit and evaluate on disjoint parts of $\boldsymbol{y}$, Ψ is an unbiased estimator of the test negative log likelihood, and minimizing Ψ should be robust to overfitting.

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 keep the set $\{\boldsymbol{\alpha}_{[J_k]}\}$ of primary variables, and iterate between re-fitting those for each fold I_k, and computing Ψ and $\nabla_{\boldsymbol{h}}\Psi$. The latter 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 6.3.

As for the 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 outer 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 kernel derivative MVM part consists of q derivative MVM calls for each independent component of $\boldsymbol{h}$, see Section 9.3. As opposed to the accumulation part, this part consists of a simple large matrix operation and can be run very efficiently using specialized numerical linear algebra code. Given the method for computing Ψ and $\nabla_{\boldsymbol{h}}\Psi$, we plug these into a custom optimizer such as Quasi-Newton in order to learn $\boldsymbol{h}$.

As shown in Section 6.4, the extension of hyperparameter learning to the hierarchical case of Section 4 is simply done by wrapping the accumulation part, the coding and additional memory effort being minimal.

6 Computational Details

In this Section, we provide details for the Sections above. The techniques given here do characterize our framework, and they may be useful in many other contexts as well. For simplicity, we present them in the context of the applications introduced above, but they do apply to other instances just as well.

Further, more specific details of our implementation can be found in Appendix D.

6.1 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 3. Note that this fitting method appeared in [30] in the context of approximate Gaussian process inference, although some fairly

[4] Which is why they are *not* referred to as $\boldsymbol{\alpha}_{J_k}$.

essential ideas here are novel (symmetrization of Newton system, pair optimization line search, numerical stability considerations).

Recall that we want to minimize the strictly convex criterion Φ (Eq. 1) w.r.t. $\boldsymbol{\alpha}$, $\boldsymbol{b}$, 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 6.1.1, commenting on the line searches afterwards in Section 6.1.2 (it turns out that it basically comes for free). An overview of the fitting algorithm is given in Section 6.1.3.

6.1.1 Computing the Newton Direction

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

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

Here, $\boldsymbol{D} = \operatorname{diag} \boldsymbol{\pi}$, and gradient and Hessian are taken w.r.t. $\boldsymbol{u}$ (*not* w.r.t. $\boldsymbol{\alpha}$). Note that our component ordering convention on vectors implies that in matrices $\boldsymbol{A} \otimes \boldsymbol{B}$ we use here, the factors $\boldsymbol{A}$ and $\boldsymbol{B}$ scale with C and n respectively. For example, in $\boldsymbol{1} \otimes \boldsymbol{I}$ we have $\boldsymbol{1} \in \mathbb{R}^C$, $\boldsymbol{I} \in \mathbb{R}^{n,n}$. Also note that $(\boldsymbol{1}^T \otimes \boldsymbol{I})\boldsymbol{a} = \sum_c \boldsymbol{a}^{(c)}$, and $(\boldsymbol{1} \otimes \boldsymbol{I})\boldsymbol{a}$ is $\boldsymbol{a}$ stacked C times on top of each other:

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

The form of $\boldsymbol{W}$ can be understood by noting that $\boldsymbol{W}$ is block-diagonal in a *different* ordering, which uses c as inner and i as outer index, then switching to our ordering.

It is easy to compute gradient and Hessian of Φ w.r.t. $\boldsymbol{\alpha}$, $\boldsymbol{b}$. The Newton direction is given by the equations

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

For simplicity in the formulae, we work in terms of variables $\boldsymbol{\alpha}'$, $\boldsymbol{b}'$ the classical Newton method would use *instead* of doing a line search. The Newton 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 $b_c = \sigma^2 \sum_i \alpha_{i,c}$. This is the meaning of "eliminating of $\boldsymbol{b}$" in Section 3. While we could optimize σ^2 as a

hyperparameter, we consider it fixed and given for simplicity[5]. 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 of Eq. 4 exactly if we can tolerate a scaling of $O(n^3\, C)$. Note that this scaling is linear rather than cubic in C. The exact solution is derived in Appendix B. In the remainder of this Section, we focus on approximate computations.

Although we could solve the system approximately 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. In order to justify our independent treatment of these functions (their priors or penalty functionals are independent), 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 Eq. 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 $\boldsymbol{g} \in \operatorname{ran} \boldsymbol{W}$ and $\boldsymbol{\alpha}' \in \operatorname{ran} \boldsymbol{W}$. 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 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 of Eq. 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 Eq. 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 of Eq. 6 for $\boldsymbol{\beta}$. This is a symmetric positive definite system with the well-conditioned matrix $\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}$, and

[5]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.

can be solved efficiently using the linear conjugate gradients (LCG) algorithm. The cost of each step is dominated by the MVM $\boldsymbol{s} \mapsto \boldsymbol{K}\boldsymbol{s}$, which scales linearly in C, due to the block-diagonal structure of $\boldsymbol{K}$. Third, set $\boldsymbol{\alpha}' = \boldsymbol{V}\boldsymbol{\beta}$.

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 adressed in Appendix A. Furthermore, the LCG algorithm is hardly ever run without some sort of preconditioning. Our present implementation uses diagonal preconditioning, as described in Appendix A. We have already noted in Section 4 that a non-diagonal preconditioning strategy could be very valuable, but this is subject to future work.

6.1.2 The Line Search

The textbook NR algorithms proceeds doing full steps $\boldsymbol{\alpha} \to \boldsymbol{\alpha}'$, but this is not in general a globally convergent method, because it is not guaranteed to decrease the criterion in each step. The usual remedy is to combine it with a line search routine. 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.

6.1.3 Overview of the Optimization Algorithm

In Algorithm 3, we give a schematic overview of the fitting algorithm. We use primitives for the functions $\boldsymbol{s} \mapsto \boldsymbol{K}\boldsymbol{s}$ and $\boldsymbol{s} \mapsto \tilde{\boldsymbol{A}}^{-1}\boldsymbol{s}$ for the (diagonal) preconditioner of the LCG run. The latter can be configured with the current $\boldsymbol{\pi}$ vector. For simplicity, we do not include the measures discussed in Appendix A to increase numerical stability.

6.2 A Generic Kernel Matrix Representation

A kernel matrix representation is some data structure which allows to compute kernel matrix MVMs $\boldsymbol{s} \mapsto \boldsymbol{K}^{(c)}\boldsymbol{s}$ efficiently, which are the principal primitives of our primary fitting

Algorithm 3 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 Φ.

if relative improvement in Φ 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 of Eq. 6 approximately. The CG code is configured by a primitive to compute $\boldsymbol{s} \mapsto (\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V})\boldsymbol{s}$, which in turns call the primitive for $\boldsymbol{s} \mapsto \boldsymbol{K}\boldsymbol{s}$.

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

method. Further requirements arise if additional features of our framework are used. For example, if hyperparameters are to be learned as well, derivative MVMs $\boldsymbol{s} \mapsto (\partial\boldsymbol{K}^{(c)}/\partial h_p)\boldsymbol{s}$ are required as well, and "covariance shuffling" should be possible (see Section 6.3).

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.2. 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 $K^{(c)} = v_c M^{(l_c)}$ 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 diagonals $\operatorname{diag} \boldsymbol{M}^{(l)}$ are stored separately, and whenever a specific $\boldsymbol{M}^{(l)}$ is required explicitly, the diagonal is copied into the rectangular block. It is important to note that the BLAS (see Section 9.2) 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 on a matrix, even if the matrix is stored contiguously. This gives another indication that in general, optimization

methods which require small parts of kernel matrices very often, do have a performance deficit on modern architectures (see Section 9.2 for more on this point).

6.3 Details for Hyperparameter Learning

In this Section, we provide details for the CV hyperparameter learning scheme, introduced in Section 5.

The gradient of the CV criterion Ψ of Eq. 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 $\boldsymbol{\alpha}_{[J]} + \boldsymbol{g}_{[J]} = \mathbf{0}$ (all terms of subscript $[J]$ are as in Section 6.1.1, but for the subset J of the data, and w.r.t. $\boldsymbol{\alpha}_{[J]}$). Taking derivatives gives

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

We obtain a system for $d\boldsymbol{\alpha}_{[J]}$ which is symmetrized as in Section 6.1.1:

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

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

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

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

If we collect these vectors as columns of $\boldsymbol{E}$, $\boldsymbol{F} \in \mathbb{R}^{nC,q}$, 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 5. It involves a loop over folds, in which $\boldsymbol{\alpha}_{[J_k]}$ is determined by NR optimization, starting from its previous value, then the $\boldsymbol{s}$ (column of $\boldsymbol{F}$) is computed by solving one more Newton direction system. Importantly, the accumulation is independent of the number of hyperparameters. The gradient computation then requires to compute Eq. 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 9.3), and one may be able to share computations between the different gradient components. Importantly, the computation of Eq. 7 is easily broken down into large numerical linear algebra primitives, for which very efficient code may be used (see Section 9.2). This is a significant advantage in the presence of many hyperparameters. For few hyperparameters, the accumulation clearly dominates the CV criterion and gradient computation.

The dominating part of the accumulation is the reoptimization of the $\boldsymbol{\alpha}_{[J]}$, which are done by calling the optimized code for primary fitting (Section 6.1) as subroutine. Here, a feature of our implementation becomes very important. Instead of representing each of $\boldsymbol{K}_{J_k}$ for kernel MVMs, we represent $\boldsymbol{K}$ only, as if we were interested in fitting the complete data. The representation depends on the covariance function, and in general on how kernel MVMs are

actually done. A generic representation is described in Section 6.2. In order to work on the data subset J_k, we shuffle the representation such that in the shuffled 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 having to map matrix coordinates through an index, which is many times faster. Details on "covariance shuffling" are given in Appendix D.1.

As mentioned in Section 6.1.1 and detailed in Appendix B, we can 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 B. Exact computations lead to more robust behaviour, and may actually run faster for small to moderate n. In our implementation, we use exact computations for debugging purposes.

6.4 Details for Hierarchical Classification

In this Section, we provide details for hierarchical classification method, introduced in Section 4.

Recall that $\boldsymbol{u} = \boldsymbol{\Phi}\breve{\boldsymbol{u}}$ for an indicator matrix $\boldsymbol{\Phi}$ of simple structure, and that MVM with $\boldsymbol{\Phi}$ or $\boldsymbol{\Phi}^T$ can be computed easily in $O(n\,C)$, without having to store $\boldsymbol{\Phi}$. Since the $\breve{u}_p$ are given independent priors (or regularizers), the corresponding kernel matrix $\breve{\boldsymbol{K}}$ is block-diagonal, and the induced covariance matrix $\boldsymbol{K}$ over $\boldsymbol{u}_L$ is given by Eq. 2, and hierarchical classification differs from flat one principally in that this non-block-diagonal matrix is used.

The MVM primitive $\boldsymbol{s} \mapsto \boldsymbol{K}\boldsymbol{s}$ 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 generalizing the algorithms in Section 4 to $\boldsymbol{S} \mapsto \boldsymbol{S}\boldsymbol{\Phi}$, $\boldsymbol{S} \mapsto \boldsymbol{S}\boldsymbol{\Phi}^T$ 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. If some of the $\breve{\boldsymbol{K}}^{(p)}$ are the same up to a prefactor (see Section 6.2), this can be used to code MVM with $\breve{\boldsymbol{K}}$ more efficiently[6].

The diagonal preconditioning (see Section 6.1.1) requires the computation of $\mathrm{diag}\,\boldsymbol{K} \in \mathbb{R}^{nC}$. If p is an inner node with $p = i_c$, then

$$\boldsymbol{K}_{(i,c),(i,c)} = (\boldsymbol{\delta}_p^T\boldsymbol{\Phi} \otimes \boldsymbol{\delta}_i^T)\breve{\boldsymbol{K}}(\boldsymbol{\Phi}^T\boldsymbol{\delta}_p \otimes \boldsymbol{\delta}_i) = \boldsymbol{\delta}_p^T\boldsymbol{\Phi}(\mathrm{diag}(\breve{\boldsymbol{K}}_i^{(p')})_{p'})\boldsymbol{\Phi}^T\boldsymbol{\delta}_p,$$

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 that

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

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

Hyperparameter learning (see Section 6.3) is easily extended to the hierarchical case, by noting Eq. 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 6.3). The gradient components of Eq. 7 translate to $\mathrm{tr}\,\tilde{\boldsymbol{E}}^T(d\breve{\boldsymbol{K}})\tilde{\boldsymbol{F}}$, 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 $\boldsymbol{E}$, $\boldsymbol{F}$ during accumulation. We then transform them to $\tilde{\boldsymbol{E}}$, $\tilde{\boldsymbol{F}}$ using in-place computations.

[6] In the notation of Section 6.2, we collect all columns to be multiplied with a single $\boldsymbol{M}^{(l)}$ in a matrix. Again, this makes use of the fact that optimized code of the BLAS is in general more efficient if applied to larger matrices.

It should be clear that the step from flat to hierarchical classification requires minor modifications of existing code. We only need to write wrappers for MVM and the other primitives which essentially pre- and postmultiply the input with $\boldsymbol{\Phi}$ and $\boldsymbol{\Phi}^T$ respectively, then call the existing "flat" primitives for the inner covariance matrix $\breve{\boldsymbol{K}}$), which is block-diagonal with P blocks rather than C.

7 Further Extensions

In this Section, we collect suggestions for further extensions and applications of our framework, some of which are currently explored in further detail.

7.1 Label Sequence Learning

HIER!!! Write stuff about kernel CRF, and how the Hessian-vector product can be computed efficiently!

7.2 Modelling Dependencies between Classes

In the flat classification application of Section 3, we do not explicitly represent dependencies between classes. This is done in the hierarchical classification application of Section 4, but the dependency structure is fixed *a priori*. In this Section, we motivate how dependencies between classes may 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 MVM with $\boldsymbol{K}$, because MVM with $(\boldsymbol{B} \otimes \boldsymbol{I})$ can be done in $O(C^2 n)$.

We are also interested in learning $\boldsymbol{B}$ whose elements are taken as hyperparameters. The corresponding gradient is obtained in the same way as described in Section 6.3, only that $d\boldsymbol{K}_{[B]}$ now further decomposes into parts for $d\boldsymbol{B}$ and for $d\boldsymbol{K}$. The details are not given here.

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 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 in [26].

7.3 Uncertain Targets in Hierarchical Classification

In this Section, we discuss an extension of the hierarchical classification application of Section 4. 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 I_p the set of leaf nodes of the subtree rooted at p, *i.e.* $I_p = \{p\}$ for a leaf node (with $p \in L$), and $I_0 = I$.

We can allow for such uncertain target information by letting $y_i \in \{1, \dots, P\}$, meaning that the corresponding likelihood factor is

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

Note that the log likelihood is not a concave function anymore, whenever $|I_{y_i}| > 1$. However, an *expectation-maximization* (EM) approach can be used to deal with uncertain targets. Namely, in E steps we compute

$$q_{i,c} \propto \mathrm{I}_{\{c \in I_{y_i}\}} P(y = c | \boldsymbol{u}(\boldsymbol{x}_i))$$

for the current $\boldsymbol{u}(\cdot)$, where $\boldsymbol{q}_i = (q_{i,c})_c$ is a distribution. M steps consists of Newton-Raphson iterations as before, but using $\sum_c q_{i,c} \log P(y = c | \boldsymbol{u}(\boldsymbol{x}_i))$ 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 use the properties $\boldsymbol{1}^T \boldsymbol{y}_i = 1,\ \boldsymbol{y}_i \geq \boldsymbol{0}$ above, which is true for $\boldsymbol{q}$ just as well.

7.4 Low Rank Approximations

Recall our generic kernel matrix representation from Section 6.2. 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 9.3). 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*[7] (ICF), in which case we have that $\boldsymbol{L}^{(l)}_{1 \dots d_l, \cdot}$ 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

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

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

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

Snelson and Ghahramani [25] motivate this diagonal correction in another context. It is clear that MVM with $\tilde{\boldsymbol{M}}^{(l)}$ can be done in $O(n\, d_l)$.

[7] 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

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

In order to learn parameters of the $M^{(l)}$ functions in this low rank setup, we assume that the choice of I_l does not depend on these kernel parameters. Under this assumption, we can compute the required gradient terms $\operatorname{tr}\boldsymbol{E}_p{}^T(d\tilde{\boldsymbol{M}}^{(l)})\boldsymbol{F}_p$, as is shown in Appendix C.

Finally, the reader may guess that by applying low-rank approximations, one can get around a basic limitation of our approach, namely that at least some vectors of size $C\,n$ have to be maintained in memory. This is not true, however, at least if no further approximations are done. While the application of low-rank techniques lead to $\boldsymbol{\alpha}$ of reduced size, $\boldsymbol{u}$ and $\boldsymbol{\pi}$ are still of size $C\,n$ (or $P\,n$ in the hierarchical case), and not maintaining them explicitly results in significant performance drops.

8 Experiments

In this Section, we provide experimental results for our framework on data from remote sensing (flat classification), and on a set of large text classification tasks with very many classes (hierarchical classification).

8.1 Flat Classification: Remote Sensing

We use the *satimage* remote sensing task from the *statlog* repository.[8] This task has been used in the extensive SVM multi-class study of [10], 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.

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

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

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), *1rest* (one-against-rest: C binary classifiers are trained separately to discriminate c from all others, they are voted by log probability upon prediction; 12 hyperparameters)[9]. Since we use the same task as [10], our results can be directly compared to theirs[10].

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.

[8] Available at `http://www.niaad.liacc.up.pt/old/statlog/`.

[9] We use our method with $C = 2$ in order to implement the binary classifiers for *1rest*. This can be done more efficiently.

[10] Although one should note that the results in [10] required a lot of user interaction, while ours are obtained completely automatically.

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 were 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. ROC curves[11] for the different methods are shown in Figure 3.

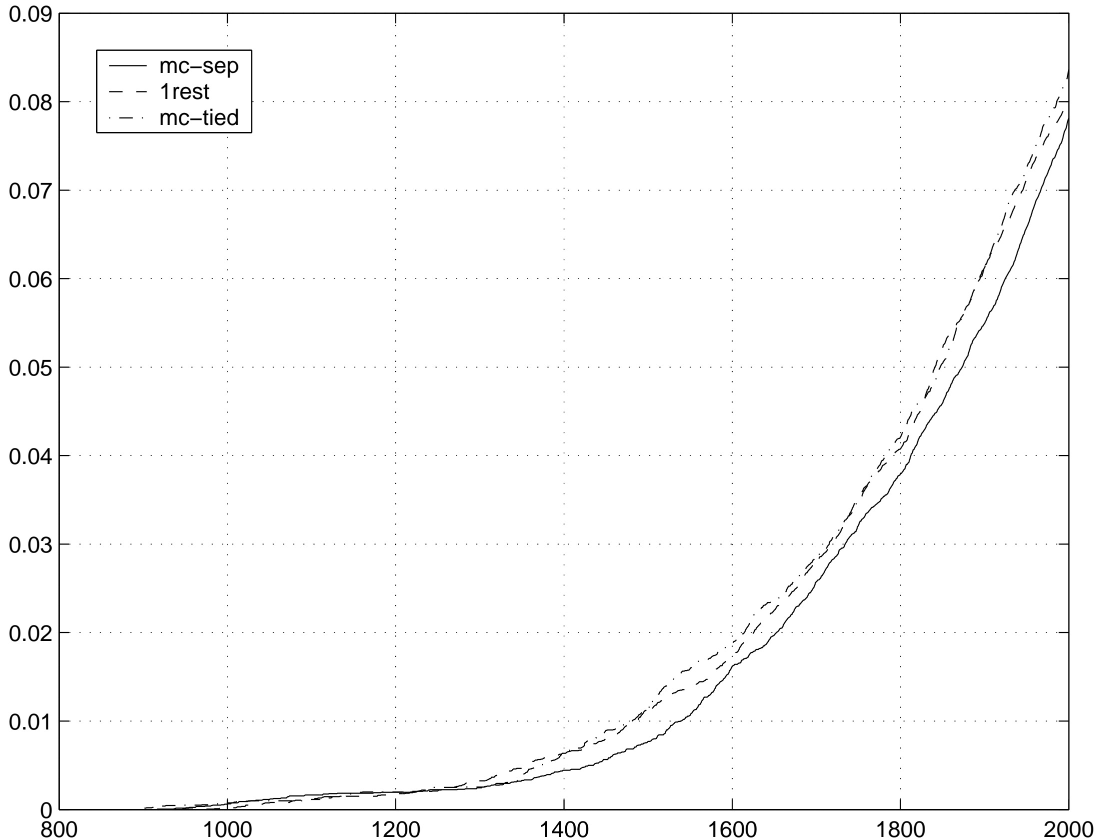


Figure 3: ROC curves for different methods on the SATIMAGE remote sensing task.

Our method *mc-sep* does significantly better than *1rest* or *mc-tied*. The test error of *mc-sep* is 0.0781 ± 0.0016 vs. 0.0801 ± 0.0005 for *1rest*. The result for *mc-sep* is state-of-the-art, for example the best SVM technique tested in [10] attained 0.0765, and SVM "one-against-rest" attained 0.0830 in this study. We see that constraining all kernels to be the same results in worse performance and cannot be recommended for this task, but although *1rest* is allowed to use a separate kernel for each class, it does not make good use of this feature, in contrast to our joint method. Although both methods have the ability of learning individual kernel parameters, the way in which this is done is significant.

In Figure 4, we plot the difference between ROC *mc-sep* and *1rest*, indicating the regime (of reject fraction of the test set) for which the advantage of *mc-sep* is most pronounced:

[11] We use the following definition for ROC curve: the value at $k \le m$ is obtained by requiring a prediction on k test points, allowing the method to reject $m - k$ points, then dividing the number of errors in the predictions done by m. The right-most value is the conventional empirical test error.

Figure 4: Difference ROC curves *1rest* and *mc-sep*.

namely when up to 10% may be rejected. This result indicates that while our joint method already improves upon *1rest* in test error, the advantage becomes even more pronounced in the presence of a reject option. This may be due to the fact that the joint model does a better job in estimating the true uncertainties $P(y_*|\boldsymbol{x}_*)$.

8.2 Hierarchical Classification: Patent Text Classification

We use the WIPO-alpha collection, which can be obtained from `www.wipo.int/ibis/datasets`, the label hierarchy is described at `www.wipo.int/classifications/en`. 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 of occuring words. All cases $\boldsymbol{x}_i$ were scaled to unit norm. Many thanks to Peter Gehler for helping us with the preprocessing.

These tasks have previously been studied in [4], 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 [4], 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.2) 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 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 $y(\boldsymbol{x}_j) = \mathrm{argmax}_c\, p_{j,c}$ maximizes expected accuracy, classes are ranked as $r_j(c) \le r_j(c')$ iff $p_{j,c} \ge p_{j,c'}$. The test scores we use here are the same as in [4]: *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 node of $L(c)$. 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 in [4]. For taxo-loss and parent accuracy, we better choose $y(\boldsymbol{x}_j)$ to minimize expected loss[12], which is different in general than the standard prediction.

We compare methods F1, F2, H1, H2 (F: flat; 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 3 parameters determining the running time (see Section 3, Section 5). 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)[13]. For the final fitting: $k_1 = 25, k_2 = 12$ (F1, F2); $k_1 = 30, k_2 = 17$ (H1, H2). The optimization is started from $v_c = 5$ for all methods. Results are given in Table 1.

The hierarchical model outperforms the flat one consistently, especially in 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 8.1). The results are very similar to the ones of [4]. However, for our method, the recommendation in [4] to use $v_c = 1$ leads to significantly worse results in all scores, the v_c chosen by our methods are generally larger.

In Table 2, we present running times[14] for the final fitting and for a single fold during hyperparameter optimization (5 of these are required for Ψ, $\nabla_{\boldsymbol{h}}\Psi$). For example, Cai and Hofmann [4] quote a final fitting time of $2200s$ on the D section, while we require $119s$ (more than 18 times faster). It is precisely this high efficiency of primary fitting which allows us to use it as inner loop for automatic hyperparameter learning. Possible reasons for the performance difference are given in Section 9.2.

9 Discussion

We have presented a general framework for learning kernel-basic 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, requiring few large steps for convergence. The steps

[12]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 $p(j)$.

[13]Except for section C, where $k_1 = 14, k_2 = 6, k_3 = 35$.

[14]Processor time on 64bit 2.33GHz AMD machines.

	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 $\operatorname{argmax}_c p_{j,c}$ 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. CV Fold: Re-optimization of $\boldsymbol{\alpha}_{[J]}$, gradient accumulation for single fold.

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 9.2. Dependencies between classes can be encoded *a prior* 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 again reduced to the same MVM primitives. This means that within our framework, all code optimization effort can be concentrated on these essential primitives (see also Section 9.3), rather than having to tune a set of further heuristics.

9.1 Related Work

Some less directly related work has been mentioned in Section 2. Or primary fitting optimization for flat multi-way classification appeared in [30], although some numerically essential features seem novel here, and they did not consider large scale problems or class structures. Empirical Bayesian criteria such as the marginal likelihood are routinely used for hyperparameter learning [30, 29]. However, in cases other than regression estimation with Gaussian noise, the marginal likelihood for a Gaussian process model cannot be computed analytically, and approximations differ strongly in terms of accuracy and computational complexity. For the multi-class model, [30] 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).

The idea of optimizing approximations to a cross-validation score is not new [6, 18]. Our approach is different, in that the CV score and gradient computations are reduced to elementary steps of the primary fitting method, making it amenable to the same approximations. In constrast, scores like GCV [6] or second order marginal likelihood [30] come in terms of the form $\mathrm{tr}\, \boldsymbol{H}^{-1}$ 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. [18] propose an interesting approach of approximating *leave-one-out (LOO)* CV using *expectation propagation (EP)* (the idea already appears in [15]) 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.

Many different multi-class SVM techniques have been proposed (see [5, 10] 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 8.1 is a prominent example), with the advantage that good methods and 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 often not large, and in these cases, commonly used chunking algorithms for multi-class SVM can be very inefficient (see Section 9.2). 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 (CITE CHAPELLE, KEERTHI!!!). SVM methods typically do not come with efficient automatic kernel parameter learning schemes, and they do not provide estimates of predictive probabilities which are asymptotically correct.

9.2 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[15]. 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 one of the most important characteristics[16], 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 [1], 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[17], yet much smaller than the whole, and often the steps can be solved analytically. Ignoring the aspect of scheduling, such methods are very 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 may have to be corrected if constraints are present (conditional gradient or gradient projection methods). If the Newton direction cannot be computed feasibly, approximations are required.

Such Newton-like methods are certainly vastly preferred in 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, and this Section focusses

[15] An almost low-rank kernel matrix translates into a coupling of a simple structure, but the dominant couplings are typically strong.

[16] We mean speed of convergence in the real world, which includes memory usage.

[17] 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.

our arguments. 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 [30, 19], albeit both typically happens on problems of smaller scale.

To be clear, we do *not* advocate to reduce Machine Learning problems directly to standard Optimization problems, then to employ black box optimizers. Machine Learning problems are different from canonical Optimization problems, and glossing over these differences leads to inaccurate models or wastes efficiency. Furthermore, priorities are different in Machine Learning and in Optimization. Still, many aspects of problems are shared between the disciplines, and Machine Learning researchers should know about and use Optimization wisdom when addressing such aspects.

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 [17], 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 very easily implemented. On the other hand, as SVMs are increasingly applied to hard structured label problems which 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. Quoting Bertsekas [1], such 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." ([1], p. 269) In kernel methods, such a decomposable structure is not present, because the penalization terms couple all variables strongly via the kernel matrices. 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 very similar to a number of cases in Machine Learning and Statistics. Iterative proportional fitting [7] is a BCD method for learning the potentials of an undirected graphical model (Markov random field, or conditional random field) which enjoyed enormous popularity due to its simple implementation, yet is all but superseded now by modern global direction methods such as limited memory Quasi-Newton which run 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 vast majority of cache accesses are misses (REF TO EXAMPLE OF DIMENSION FLIPPING!). This is not a problem which can be addressed by using a better optimizing compiler, but is an inherent characteristic of the method.

In well-designed global direction methods, the bottleneck operations, where most of the real-world running time is concentrated, are large vectorized mappings which access memory contiguously. Even better, these operations should fall in a standard class, for which highly optimized implementations for the computational architecture are readily available. In our case here, the bottleneck operations are standardized numerical linear algebra primitives from the *basic linear algebra subroutines* (BLAS), which is a standardized interface for high-performance dense linear algebra code. Very efficient implementations of the BLAS have been implemented for all major computer architectures, and with ATLAS, a self-tuning implementation is freely available[18].

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 still ongoing efforts to formulate otherwise tractable Machine Learning problems as semidefinite programs and "solving" them using $O(n^6)$ SDP packages. This association is wrong, as already mentioned above. 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 of 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.

9.3 Matrix-Vector Multiplication

The computational load in our framework is given by application of the MVM primitives $\boldsymbol{s} \mapsto \boldsymbol{K}^{(c)}\boldsymbol{s}$ and $\boldsymbol{s} \mapsto (\partial \boldsymbol{K}^{(c)}/\partial h_p)\boldsymbol{s}$. A user only needs to provide those for a kernel of choice. The generic representation of Section 6.2 applies to general covariance functions, but much more efficient alternatives may be used in special cases (REF TO LIN. KERNEL).

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 [31, 23] such as the Gaussian of Eq. 8. For such kernels, the derivative MVM can often be addressed using the same

[18] See `http://math-atlas.sourceforge.net/`.

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.

Another general approximation technique is to replace $\boldsymbol{K}$ by a low-rank matrix, namely by $\boldsymbol{K}^{(S)} = \boldsymbol{K}_{\cdot,S}\boldsymbol{K}_S^{-1}\boldsymbol{K}_{S,\cdot}$ for some $S \subset \{1,\dots,n\}$ [24, 13]. This allows MVM to be computed in $O(n\,|S|)$. We can use

$$d\boldsymbol{K}^{(S)} = 2\,\mathrm{sym}(d\boldsymbol{K}_{\cdot,S})\boldsymbol{R}^T - \boldsymbol{R}(d\boldsymbol{K}_S)\boldsymbol{R}^T, \boldsymbol{R} = \boldsymbol{K}_{\cdot,S}\boldsymbol{K}_S^{-1} \quad (9)$$

for computing the derivative MVMs. The selection of the "active set" S is a problem to be solved independently. Some more details about low-rank approximations are given in Section 7.4.

9.4 Extensions and Future Work

The extension noted in Section 7 have not been explored properly yet, but this is work in progress (the extension to label sequence learning is maybe the most interesting one). In general, we think that most structured label kernel methods proposed in the SVM context can be addressed in our framework as well. We also plan to address hierarchical multi-label problems, which differ from hierarchical multi-class in that each instance can have multiple associated labels.

In general, it is not yet clear whether our CV criterion is useful in order to learn a large number of hyperparameters, or whether severe local minima problems or overfitting takes place in such a case. Additional experiments, with suitable regularization of the hyperparameters, are required to gain clarity 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.

References

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

[2] S. Boyd and L. Vandenberghe. *Convex Optimization.* Cambridge University Press, 2002. Available online at `www.stanford.edu/~boyd/cvxbook.html`.

[3] C. Brodley, editor. *International Conference on Machine Learning 21.* Morgan Kaufmann, 2004.

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

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

[6] 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.

[7] 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.

[8] Mark N. Gibbs. *Bayesian Gaussian Processes for Regression and Classification*. PhD thesis, University of Cambridge, 1997.

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

[10] 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.

[11] 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.

[12] J. Lafferty, X. Zhu, and Y. Liu. Kernel conditional random fields: Representation and clique selection. In Brodley [3].

[13] N. D. Lawrence, M. Seeger, and R. Herbrich. Fast sparse Gaussian process methods: The informative vector machine. In S. Becker, S. Thrun, and K. Obermayer, editors, *Advances in Neural Information Processing Systems 15*, pages 609–616. MIT Press, 2003.

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

[15] Manfred Opper and Ole Winther. Gaussian processes for classification: Mean field algorithms. *Neural Computation*, 12(11):2655–2684, 2000.

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

[17] John C. 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.

[18] Y. Qi, T. Minka, R. Picard, and Z. Ghahramani. Predictive automatic relevance determination by expectation propagation. In Brodley [3].

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

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

[21] N. Schraudolph. Fast curvature matrix-vector products. In Georg Dorffner, Horst Bischof, and Kurt Hornik, editors, *ICANN*, pages 19–26. Springer Verlag, 2001.

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

[23] Y. Shen, A. Ng, and M. Seeger. Fast Gaussian process regression using KD-trees. In Weiss et al. [28].

[24] A. Smola and P. Bartlett. Sparse greedy Gaussian process regression. In T. Leen, T. Dietterich, and V. Tresp, editors, *Advances in Neural Information Processing Systems 13*, pages 619–625. MIT Press, 2001.

[25] E. Snelson and Z. Ghahramani. Sparse Gaussian processes using pseudo-inputs. In Weiss et al. [28].

[26] 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.

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

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

[29] 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.

[30] 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.

[31] 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.

A Details for Primary Fitting Algorithm

In this Section, we discuss some details of the primary fitting algorithm of Section 3, in addition to Section 6.1.

In a general-purpose implementation such as ours, it is wise to deal with 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 $\log \pi_{i,c} = u_{i,c} - l_i$. We make use of a threshold $\kappa < 0$ and define

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

The indices in I can be problematic due to the corresponding component $\tilde{g}_{i,c} \approx y_{i,c}/\pi_{i,c}^{1/2}$ 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 in Eq. 6 we see that

$$\boldsymbol{D}^{1/2}\boldsymbol{\beta} = \boldsymbol{D}\boldsymbol{A}^T\left(\boldsymbol{u} - \tilde{\boldsymbol{K}}\boldsymbol{\alpha}'\right) - \boldsymbol{g}.$$

Therefore, if $(i,c) \in I$, then $(\boldsymbol{D}^{1/2}\boldsymbol{\beta})_{i,c} \approx -g_{i,c} \approx y_{i,c}$. 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 $\tilde{g}_{i,c} \approx 0$, leading to $\beta_{i,c} \approx 0$ from Eq. 6. The corresponding components in the starting value $\boldsymbol{\beta}$ can therefore be set to 0.

Next, the linear conjugate gradients algorithm used to solve systems of the form Eq. 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 $\boldsymbol{s} \mapsto \tilde{\boldsymbol{A}}^{-1}\boldsymbol{s}$ can be computed efficiently (essentially in linear time in the size of $\boldsymbol{s}$), 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 typically converges faster and less erratically.

Our implementation involves preconditioning with the diagonal of the system matrix $\boldsymbol{I} + \boldsymbol{V}^T\tilde{\boldsymbol{K}}\boldsymbol{V}$. Note that $\boldsymbol{V}\boldsymbol{\delta}_{ic} = \pi_{i,c}^{1/2}(\boldsymbol{\delta}_c - \boldsymbol{\pi}_i) \otimes \boldsymbol{\delta}_i$, so that

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

Therefore, the diagonal can be computed based on the $\operatorname{diag}\boldsymbol{K}^{(c)}$ vectors.

If the joint kernel matrix $\boldsymbol{K}$ is not block-diagonal (as in hierarchical classification, see Section 4), $\operatorname{diag}\boldsymbol{K}$ 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

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

B Solving Systems Exactly

CHANGE THIS!!!!! NOT UP TO DATE!!

In this Section we show how to implement our scheme using exact rather than approximate solutions of linear systems. The scaling requirements are $O(n^3\,C)$ time and $O(n^2\,C)$ memory. However, in the computation of the CV score and gradient, the dominating $O(n^3\,C)$ computation has to be done only once, as opposed to the approximate case where a number of systems have to be solved from scratch. Thus, for moderate n, the exact computation may actually be more efficient than the approximate one.

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

$$\boldsymbol{\alpha}'=\boldsymbol{D}^{1/2}\left(\boldsymbol{A}^{-1}+\boldsymbol{A}^{-1}\boldsymbol{U}(\boldsymbol{U}^T\boldsymbol{A}^{-1}\boldsymbol{U})^{-1}\boldsymbol{U}^T(\boldsymbol{I}-\boldsymbol{A}^{-1})\right)\boldsymbol{D}^{-1/2}\boldsymbol{r}. \tag{10}$$

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

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

For the exact computation, we compute $\boldsymbol{A}^{(c)-1}$ for all c, which is best done by first computing the Cholesky decompositions. Furthermore, we compute the Cholesky decomposition of $\boldsymbol{U}^T\boldsymbol{A}^{-1}\boldsymbol{U}$.

In order to compute the CV criterion and its gradient, we need to solve systems with reduced matrices as well. Let $\boldsymbol{H}=\boldsymbol{I}+\boldsymbol{W}\tilde{\boldsymbol{K}}$. Since $\boldsymbol{W}_{J,I}=\boldsymbol{0}$ for disjoint I,J, we have that $\boldsymbol{H}_J=\boldsymbol{I}+\boldsymbol{W}_J\tilde{\boldsymbol{K}}_J$, the reduced system matrix. The partitioned inverse equations render

$$\boldsymbol{H}_J^{-1}=(\boldsymbol{H}^{-1})_J-(\boldsymbol{H}^{-1})_{J,I}(\boldsymbol{H}^{-1})_I^{-1}(\boldsymbol{H}^{-1})_{I,J}. \tag{11}$$

If $\boldsymbol{A}^{-1}$ is given explicitly, as well as $\boldsymbol{R}^T\boldsymbol{R}=\boldsymbol{U}^T\boldsymbol{A}^{-1}\boldsymbol{U}$, the required terms can be computed relatively easily. We give details BELOW!!

TO BE CHANGED!! NOT UP TO DATE!!

The computation of the CV criterion and gradient in the exact case are slightly simpler, as shown in the following. First,

$$(\boldsymbol{I}+\boldsymbol{W}_J\tilde{\boldsymbol{K}}_J)\boldsymbol{\beta}'_J=\boldsymbol{W}_J\tilde{\boldsymbol{K}}_{J,I}\boldsymbol{\alpha}_I,\quad \boldsymbol{\alpha}'_J=\boldsymbol{\alpha}_J+\boldsymbol{\beta}'_J.$$

Note that $\boldsymbol{\beta}'_J$ is not the same as in the approximate case. Next, $(d\boldsymbol{W}_J)\boldsymbol{a}=\boldsymbol{M}_3(\boldsymbol{a})(d\boldsymbol{\pi}_J)$ with

$$\boldsymbol{M}_3(\boldsymbol{a})=(\boldsymbol{I}-\boldsymbol{D}_J\boldsymbol{P}_{cls})\boldsymbol{\Lambda}-(\operatorname{diag}\boldsymbol{P}_{cls}\boldsymbol{D}_J\boldsymbol{a}),\quad \boldsymbol{\Lambda}=\operatorname{diag}\boldsymbol{a}.$$

Furthermore, $d\boldsymbol{\pi}_J=\boldsymbol{W}_J((d\boldsymbol{K}_{J,\cdot})\boldsymbol{\alpha}+\tilde{\boldsymbol{K}}_{J,\cdot}(d\boldsymbol{\alpha}))$. Since $\tilde{\boldsymbol{K}}_{J,I}\boldsymbol{\alpha}_I-\tilde{\boldsymbol{K}}_J\boldsymbol{\beta}'_J=\tilde{\boldsymbol{K}}_{J,\cdot}\boldsymbol{q}$, we have that

$$(\boldsymbol{I}+\boldsymbol{W}_J\tilde{\boldsymbol{K}}_J)(d\boldsymbol{\beta}'_J)=\boldsymbol{M}_3(\tilde{\boldsymbol{K}}_{J,\cdot}\boldsymbol{q})\boldsymbol{W}_J\left((d\boldsymbol{K}_{J,\cdot})\boldsymbol{\alpha}+\tilde{\boldsymbol{K}}_{J,\cdot}(d\boldsymbol{\alpha})\right)+\boldsymbol{W}_J(d\boldsymbol{K}_{J,\cdot})\boldsymbol{q}$$
$$+\boldsymbol{W}_J\tilde{\boldsymbol{K}}_{J,I}\boldsymbol{I}_{I,\cdot}(d\boldsymbol{\alpha}).$$

With

$$\boldsymbol{E}_1=\boldsymbol{M}_3(\tilde{\boldsymbol{K}}_{J,\cdot}\boldsymbol{q})\boldsymbol{W}_J,\ \boldsymbol{E}_2=(\boldsymbol{I}+\boldsymbol{W}_J\tilde{\boldsymbol{K}}_J)^{-1}\boldsymbol{E}_1,\ \boldsymbol{E}_3=(\boldsymbol{I}+\boldsymbol{W}_J\tilde{\boldsymbol{K}}_J)^{-1}\boldsymbol{W}_J$$

we have that

$$d\boldsymbol{\alpha}'_J=\boldsymbol{E}_2(d\boldsymbol{K}_{J,\cdot})\boldsymbol{\alpha}+\boldsymbol{E}_3(d\boldsymbol{K}_{J,\cdot})\boldsymbol{q}+\left(\boldsymbol{E}_2\tilde{\boldsymbol{K}}_{J,\cdot}+\boldsymbol{E}_3\tilde{\boldsymbol{K}}_{J,I}\boldsymbol{I}_{I,\cdot}+\boldsymbol{I}_{J,\cdot}\right)(d\boldsymbol{\alpha}).$$

Now, $(\boldsymbol{I}+\boldsymbol{W}\tilde{\boldsymbol{K}})(d\boldsymbol{\alpha}) = -\boldsymbol{W}(d\boldsymbol{K})\boldsymbol{\alpha}$. With $\boldsymbol{r} = \tilde{\boldsymbol{K}}_{J,I}(\boldsymbol{\pi}'_I - \boldsymbol{y}_I)$, we have that $d\Phi_I = (\boldsymbol{\pi}'_I - \boldsymbol{y}_I)^T(d\boldsymbol{K}_{I,J})\boldsymbol{\alpha}'_J + \boldsymbol{r}^T(d\boldsymbol{\alpha}'_J)$. Define

$$\boldsymbol{s}_2 = \boldsymbol{E}_2^T\boldsymbol{r} = \boldsymbol{E}_1^T\boldsymbol{s}_3, \quad \boldsymbol{s}_3 = (\boldsymbol{I}+\tilde{\boldsymbol{K}}_J\boldsymbol{W}_J)^{-1}\boldsymbol{r},$$
$$\boldsymbol{s}_1 = -\boldsymbol{W}(\boldsymbol{I}+\tilde{\boldsymbol{K}}\boldsymbol{W})^{-1}\left(\boldsymbol{I}_{\cdot,J}\boldsymbol{r} + \tilde{\boldsymbol{K}}_{\cdot,J}\boldsymbol{s}_2 + \boldsymbol{I}_{\cdot,I}\tilde{\boldsymbol{K}}_{I,J}\boldsymbol{W}_J\boldsymbol{s}_3\right).$$

It is clear that systems with the matrix $\boldsymbol{I}+\tilde{\boldsymbol{K}}\boldsymbol{W}$ can be solved using a straightforward variant of the procedure described above. Then,

$$d\Phi_I = \operatorname{tr}\left(\boldsymbol{I}_{\cdot,J}\boldsymbol{\alpha}'_J(\boldsymbol{\pi}'_I - \boldsymbol{y}_I)^T\boldsymbol{I}_{I,\cdot} + \boldsymbol{\alpha}(\boldsymbol{s}_1 + \boldsymbol{I}_{\cdot,J}\boldsymbol{s}_2)^T + \boldsymbol{q}(\boldsymbol{V}_J\boldsymbol{s}_3)^T\boldsymbol{I}_{J,\cdot}\right)(d\boldsymbol{K})$$

just as in the approximate case.

CHANGE THIS:

If $\boldsymbol{H} = \boldsymbol{I}+\boldsymbol{W}\tilde{\boldsymbol{K}}$, then $\boldsymbol{H}^{-1}$ is given in Eq. 10. The reduced $\boldsymbol{H}_J^{-1}$ can be obtained using Eq. 11. In this Section, we work out the details for solving $\boldsymbol{\alpha}' = \boldsymbol{H}_J^{-1}\boldsymbol{r}$.

We first compute $(\boldsymbol{H}^{-1})_I$. Let $\boldsymbol{V}_1^{(c)} = \boldsymbol{D}^{(c)1/2}(\boldsymbol{A}^{(c)-1})_{\cdot,I}\boldsymbol{D}_I^{(c)-1/2}$ and $\boldsymbol{V}_1 = \operatorname{diag}(\boldsymbol{V}_1^{(c)})_c$. Let $\boldsymbol{V}_2^{(c)} = \boldsymbol{R}^{-1}\boldsymbol{R}^{-T}(\boldsymbol{I}_{\cdot,I} - \boldsymbol{V}_1^{(c)})$ and $\boldsymbol{V}_2 = (\boldsymbol{V}_2^{(c)})_c \in \mathbb{R}^{n,C|I|}$. Then,

$$(\boldsymbol{H}^{-1})_I = (\boldsymbol{V}_1)_{I,\cdot} + \boldsymbol{D}_I^{1/2}(\boldsymbol{A}^{-1})_{I,\cdot}\boldsymbol{D}^{1/2}(\boldsymbol{1}\otimes\boldsymbol{I})\boldsymbol{V}_2$$
$$= \left(\boldsymbol{D}_I^{(c_1)1/2}(\boldsymbol{A}^{(c_1)-1})_{I,\cdot}\boldsymbol{D}^{(c_1)1/2}\boldsymbol{V}_2^{(c_2)} + \delta_{c_1,c_2}(\boldsymbol{V}_1^{(c_1)})_{I,\cdot}\right)_{c_1,c_2}.$$

Let $(\boldsymbol{H}^{-1})_I = \tilde{\boldsymbol{Q}}\tilde{\boldsymbol{R}}$ (QR decomposition). Now, let $\boldsymbol{v}_1 = \boldsymbol{D}^{1/2}(\boldsymbol{A}^{-1})_{\cdot,J}\boldsymbol{D}_J^{-1/2}\boldsymbol{r}$, $\boldsymbol{v}_2 = \boldsymbol{R}^{-1}\boldsymbol{R}^{-T}(\boldsymbol{1}^T\otimes\boldsymbol{I})(\boldsymbol{I}_{\cdot,J}\boldsymbol{r} - \boldsymbol{v}_1)$, and $\boldsymbol{v}_3 = (\boldsymbol{H}^{-1})_{\cdot,J}\boldsymbol{r} = \boldsymbol{D}^{1/2}(\boldsymbol{A}^{-1})\boldsymbol{U}\boldsymbol{v}_2 + \boldsymbol{v}_1$. Then, $\boldsymbol{\alpha}' = (\boldsymbol{v}_3)_J - (\boldsymbol{H}^{-1})_{J,I}\tilde{\boldsymbol{R}}^{-1}\tilde{\boldsymbol{Q}}^T(\boldsymbol{v}_3)_I$. To compute the rest, go through the same procedure replacing $\boldsymbol{r}$ by $\tilde{\boldsymbol{R}}^{-1}\tilde{\boldsymbol{Q}}^T(\boldsymbol{v}_3)_I$ and exchanging I, J. It is clear that $\boldsymbol{H}_J^{-T}\boldsymbol{r}$ may be computed along the same lines, noting that $(\boldsymbol{H}^{-T})_I = (\boldsymbol{H}^{-1})_I^T = \tilde{\boldsymbol{R}}^T\tilde{\boldsymbol{Q}}^T$.

C Gradient Computation for Low Rank Approximation

CHANGE THIS!! NOT UP TO DATE!!

In this Section, we show how to compute the gradient terms $\operatorname{tr}\tilde{\boldsymbol{E}}_p^T(d\tilde{\boldsymbol{M}}^{(l)})\tilde{\boldsymbol{F}}_p$ for the low rank approximation $\tilde{\boldsymbol{M}}^{(l)}$ of the correlation matrix $\boldsymbol{M}^{(l)}$ (see Section 7.4). In the remainder of this Section, we drop the l superscript, and denote $\tilde{\boldsymbol{E}}_p$, $\tilde{\boldsymbol{F}}_p$ by $\boldsymbol{E}$, $\boldsymbol{F} \in \mathbb{R}^{n,q}$.

Recall that I denotes the active set of size d, and that

$$\tilde{\boldsymbol{M}} = (\operatorname{diag}^2\boldsymbol{M}) + \boldsymbol{P}(\boldsymbol{B} - (\operatorname{diag}^2\boldsymbol{B}))\boldsymbol{P}^T, \quad \boldsymbol{B} = \boldsymbol{L}\boldsymbol{L}^T,$$

with $\boldsymbol{L} \in \mathbb{R}^{n,d}$. If $\tilde{\boldsymbol{L}} = \boldsymbol{L}_{1\dots d,\cdot}$, then $\tilde{\boldsymbol{L}}$ is the Cholesky factor of $\boldsymbol{M}_I$. Note that $\boldsymbol{P}\boldsymbol{B}\boldsymbol{P}^T = \boldsymbol{M}_{\cdot,I}\boldsymbol{M}_I^{-1}\boldsymbol{M}_{\cdot,I}^T$, so that $\tilde{\boldsymbol{M}}$ depends on I only, but not on all of $\boldsymbol{P}$. Also, $\boldsymbol{P}\boldsymbol{L} = \boldsymbol{M}_{\cdot,I}\tilde{\boldsymbol{L}}^{-T}$.

If $\boldsymbol{R} = \boldsymbol{L}\tilde{\boldsymbol{L}}^{-1}$, we have that

$$d\boldsymbol{P}\boldsymbol{B}\boldsymbol{P}^T = 2\operatorname{sym}(d\boldsymbol{M}_{\cdot,I})\boldsymbol{R}^T\boldsymbol{P}^T - \boldsymbol{P}\boldsymbol{R}(d\boldsymbol{M}_I)\boldsymbol{R}^T\boldsymbol{P}^T,$$

using Eq. 9. Defining

$$\boldsymbol{e} = \operatorname{diag} \boldsymbol{F}\boldsymbol{E}^T, \quad \tilde{\boldsymbol{E}} = \boldsymbol{R}^T\boldsymbol{P}^T\boldsymbol{E}, \quad \tilde{\boldsymbol{F}} = \boldsymbol{R}^T\boldsymbol{P}^T\boldsymbol{F},$$

we have that

$$\operatorname{tr} \boldsymbol{E}^T(d\tilde{\boldsymbol{M}})\boldsymbol{F} = \boldsymbol{e}^T((d \operatorname{diag} \boldsymbol{M}) - (\operatorname{diag} d\boldsymbol{B})) + \operatorname{tr} \boldsymbol{E}^T(d\boldsymbol{M}_{\cdot,I})\tilde{\boldsymbol{F}} + \operatorname{tr} \boldsymbol{F}^T(d\boldsymbol{M}_{\cdot,I})\tilde{\boldsymbol{E}} \\ - \operatorname{tr} \tilde{\boldsymbol{E}}^T(d\boldsymbol{M}_I)\tilde{\boldsymbol{F}}.$$

Some algebra gives

$$\operatorname{tr} \boldsymbol{E}^T(d\tilde{\boldsymbol{M}})\boldsymbol{F} = \boldsymbol{e}^T(d \operatorname{diag} \boldsymbol{M}) \\ + \operatorname{tr}\left(\boldsymbol{E}\tilde{\boldsymbol{F}}^T + \boldsymbol{F}\tilde{\boldsymbol{E}}^T - 2\boldsymbol{P}(\operatorname{diag} \boldsymbol{e})\boldsymbol{R} + \boldsymbol{I}_{\cdot,I}\left(\boldsymbol{R}^T(\operatorname{diag} \boldsymbol{e})\boldsymbol{R} - \tilde{\boldsymbol{E}}\tilde{\boldsymbol{F}}^T\right)\right)^T (d\boldsymbol{M}_{\cdot,I}).$$

Suppose now that $p_1, \dots, p_K$ are the nodes corresponding to l, and let $\boldsymbol{\pi} = (v_{p_k})_k \in \mathbb{R}^K$. Now, $\boldsymbol{E} = (\boldsymbol{E}_{p_k})_k \in \mathbb{R}^{n,qK}$, and $\boldsymbol{F}$ respectively. If we redefine $\boldsymbol{e} = \operatorname{diag} \boldsymbol{F}\boldsymbol{\Pi}\boldsymbol{E}^T$ with $\boldsymbol{\Pi} = \operatorname{diag} \boldsymbol{1} \otimes \boldsymbol{\pi}$, $\boldsymbol{1} \in \mathbb{R}^q$, we have that

$$\sum_k v_{p_k} \operatorname{tr} \boldsymbol{E}_{p_k}^T(d\tilde{\boldsymbol{M}})\boldsymbol{F}_{p_k} = \boldsymbol{e}^T(d \operatorname{diag} \boldsymbol{M}) + \operatorname{tr}\Big(\boldsymbol{E}\boldsymbol{\Pi}\tilde{\boldsymbol{F}}^T + \boldsymbol{F}\boldsymbol{\Pi}\tilde{\boldsymbol{E}}^T - 2\boldsymbol{P}(\operatorname{diag} \boldsymbol{e})\boldsymbol{R} \\ + \boldsymbol{I}_{\cdot,I}\left(\boldsymbol{R}^T(\operatorname{diag} \boldsymbol{e})\boldsymbol{R} - \tilde{\boldsymbol{E}}\boldsymbol{\Pi}\tilde{\boldsymbol{F}}^T\right)\Big)^T (d\boldsymbol{M}_{\cdot,I}).$$

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 cacheing architectures (see Section 9.2).

D.1 Shuffling the Kernel Matrix Representation

Covariance matrix shuffling has been motivated in Section 6.3. It is required during hyperparameter optimization, because the MVM primitives for submatrices $\boldsymbol{K}_{J_k}$ have to be driven from a representation of the complete $\boldsymbol{K}$ (note that each $\boldsymbol{K}_{J_k}$ is of size $(q-1)/qn$, 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 a 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. For this to work, it is important that the underlying BLAS explicitly allows working on submatrices within upper left corners of larger frames, with virtually no loss in efficiency[19]. In the generic

[19]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.

representation of Section 6.2, 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 $\boldsymbol{K}$.

Note that only few methods of a representation are required when it is in shuffled state, namely the kernel MVM primitives (since only these are called during re-optimizations of $\boldsymbol{\alpha}_{[j_k]}$). In case of hierarchical classification (Section 4), we apply the shuffling to the inner representation of $\check{\boldsymbol{K}}$, noting that the wrapping applications of $\boldsymbol{\Phi}$ do not depend on the ordering of datapoints.

D.2 The Linear Kernel

Our application described in Section 8.2 uses the linear kernel $K^{(c)}(\boldsymbol{x}, \boldsymbol{x}') = v_c \boldsymbol{x}^T \boldsymbol{x}'$, where $\boldsymbol{x}$ is very high-dimensional (word counts over a dictionary), but also extremely sparse (by far the most counts are zero). The linear kernel fits the setup of Section 6.2 with a single $\boldsymbol{M}^{(1)} = \boldsymbol{M} = \boldsymbol{X}\boldsymbol{X}^T$, 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 (the one employed by *Matlab*).

An MVM is done as $\boldsymbol{s} \mapsto v_c(\boldsymbol{X}(\boldsymbol{X}^T \boldsymbol{s}))$, requiring two matrix-vector multiplications, one with $\boldsymbol{X}^T$ and one with $\boldsymbol{X}$. More generally, we do $\boldsymbol{S} \mapsto \boldsymbol{X}\boldsymbol{X}^T\boldsymbol{S}$ with large matrices $\boldsymbol{S}$. In this context, it is interesting to remark a finding which underlines the arguments in Section 9.2 about cache structures and contiguous memory. 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 $\boldsymbol{v} = (v_{1,1}, v_{2,1}, v_{3,1}, \dots)^T$ to $(v_{1,1}, v_{1,2}, v_{1,3}, \dots)^T$, before major kernel MVM computations are done, meaning that $\boldsymbol{S}^T$ is available for the primitives above. Even to our surprise, this simple modification led to a direct five-times speedup! This example should underline the importance of keeping memory contiguous in the bottleneck computations of a method, since only this ensures that cache hierarchies can be used optimally.