CS-E4830 Kernel Methods in Machine Learning

Lecture 10: Learning with multiple kernels

Juho Rousu

22. November, 2017

Task: Integrating different data sources

- In many application, different set of measurements of the same phenomenon might be available
- e.g Voice and Video, Text and Images, (Protein) sequence and structure
- How to learn effectively with several data sources?

Integrating data sources

Three main ways to tackle multiple data sources in machine learning:

- Early fusion (e.g. Combining features):
  - Combine the data sources first, e.g. by a human expert
  - Learn a single model using the combined data source
  - Can be used with any machine learning method, including kernels
- Intermediate fusion (e.g. Learning a combination of sources)
  - Input all data sources
  - Learn a combination (weights) of data sources while learning the model
  - Multi-view learning, e.g. Multiple Kernel Learning belongs to this category
- Late fusion (Combining classifiers):
  - Learn a model using each data source independently
  - Combine the predictions, e.g. using majority voting
  - Ensemble learners are based on this approach (not discussed on this course)

Early fusion with kernels

Using closure properties

- Assume our data sources are represented by kernels $\kappa_1, \ldots, \kappa_P$
- We can make simple combinations of the kernels by using the closure properties of kernels:
  - If $\kappa(\mathbf{x}, \mathbf{z})$ is a kernel and $c > 0$, $c\kappa(\mathbf{x}, \mathbf{z})$ is a kernel
  - If $\kappa_1(\mathbf{x}, \mathbf{z})$ and $\kappa_2(\mathbf{x}, \mathbf{z})$ are kernels, $\kappa_1(\mathbf{x}, \mathbf{z}) + \kappa_2(\mathbf{x}, \mathbf{z})$ is a kernel
  - If $\kappa_1(\mathbf{x}, \mathbf{z})$ and $\kappa_2(\mathbf{x}, \mathbf{z})$ are kernels, the product $\kappa_1(\mathbf{x}, \mathbf{z}) \cdot \kappa_2(\mathbf{x}, \mathbf{z})$ is a kernel
- These properties can be combined iteratively construct combinations of several kernels

Sum of kernels

- Simple integration strategy: sum of kernels

$$\kappa(\mathbf{x}, \mathbf{z}) = \sum_{m=1}^{P} \kappa_m(\mathbf{x}, \mathbf{z})$$

- Corresponds to concatenation feature vectors:

$$\phi(\mathbf{x}) = \left(\phi_1(\mathbf{x})', \ldots, \phi_P(\mathbf{x})'\right)'$$

with kernel as ther inner product of the concatenated vectors:

$$\langle \phi(\mathbf{x}_i), \phi(\mathbf{x}_j) \rangle = \sum_{m=1}^{P} \kappa_m(\mathbf{x}_i^m, \mathbf{x}_j^m)$$

Itemwise product of kernels

- Itemwise product of two kernels :

$$\kappa_{prod}(\mathbf{x}, \mathbf{z}) = \kappa_1(\mathbf{x}, \mathbf{z})\kappa_2(\mathbf{x}, \mathbf{z})$$

- Corresponds to a tensor product feature map (c.f. pairwise kernels from the last lecture)

$$\phi_{prod}(\mathbf{x}) = \phi^{(1)}(\mathbf{x}) \otimes \phi^{(2)}(\mathbf{x}),$$

where $\phi^{(1)} \in \mathbb{R}^{N_1}$ and $\phi^{(2)} \in \mathbb{R}^{N_2}$

- Feature map contains all pairwise features

$$\phi_{prod,h}(\mathbf{x}) = \phi_i^{(1)}(\mathbf{x})\phi_j^{(2)}(\mathbf{x}),$$

where $h = h(i, j)$ is a bijective mapping between indices h and index pairs (i, j), e.g. $h(i, j) = N_1(j - 1) + i$, for all $i = 1, \ldots, N_1$, $j = 1, \ldots, N_2$

Itemwise product of P kernels

- Itemwise product of P kernels :

$$\kappa_{prod}(\mathbf{x}, \mathbf{z}) = \kappa_1(\mathbf{x}, \mathbf{z})\kappa_2(\mathbf{x}, \mathbf{z}) \dots \kappa_P(\mathbf{x}, \mathbf{z}) = \prod_{m=1}^{P} \kappa_m(\mathbf{x}, \mathbf{z})$$

- Corresponds to a P-way tensor product feature map

$$\phi_{prod}(\mathbf{x}) = \phi^{(1)}(\mathbf{x}) \otimes \phi^{(2)}(\mathbf{x}) \otimes \cdots \otimes \phi^{(P)}(\mathbf{x})$$

- Feature map contains all products of P features, one feature from each kernel

$$\phi_{prod,h}(\mathbf{x}) = \phi^{(1)}_{i_1}(\mathbf{x})\phi^{(2)}_{i_2}(\mathbf{x}) \dots \phi^{(P)}_{i_P}(\mathbf{x})$$

where $h = h(i_1, i_2, \dots, i_P)$ is a bijective mapping between indices i and index P-tuples $(i_1, i_2, \dots, i_P)$, where $i_k = 1, \dots, N_k$ indexes the features in the k'th feature map

Polynomial combinations of kernels

- The polynomial kernel $(\kappa(\mathbf{x}, \mathbf{z}) + c)^k$ applied to the sum of kernels to induce a polynomial combination of kernels

$$\kappa_{poly}(\mathbf{x}, \mathbf{z}) = (\sum_{m=1}^{P} \kappa_m(\mathbf{x}, \mathbf{z}) + c)^k$$

- Expanding for $k = 2, c = 1$ one obtains

$$(\sum_{m=1}^{P} \kappa_m(\mathbf{x}, \mathbf{z}) + 1)^2 = \sum_{m_1, m_2 = 1}^{P} \kappa_{m_1}(\mathbf{x}, \mathbf{z}) \kappa_{m_2}(\mathbf{x}, \mathbf{z}) + 2 \sum_{m=1}^{P} \kappa_m(\mathbf{x}, \mathbf{z}) + 1$$

- First term is a sum of products of kernel pairs $\implies$ feature map contains all pairwise features $\phi_{m_1,i}(\mathbf{x})\phi(m_2, j(\mathbf{z}))$, where $1 \leq m, m_2 \leq P$ indexes the kernels and $1 \leq i \leq N_{m_1}$ and $1 \leq j \leq N_{m_2}$ index the features within the kernels
- Second term is the sum of kernels $\implies$ concatenation of all feature from all kernels

Polynomial combinations of kernels

- The polynomial kernel $(\kappa(\mathbf{x}, \mathbf{z}) + c)^k$ applied to the sum of kernels to induce a polynomial combination of kernels

$$\kappa_{poly}(\mathbf{x}, \mathbf{z}) = (\sum_{m=1}^{P} \kappa_m(\mathbf{x}, \mathbf{z}) + c)^k$$

- For general k, we get all products

$$\phi_{m_1,j_1}(\mathbf{x})\phi_{m_2,j_2}(\mathbf{x}) \cdots \phi_{m_k,j_k}(\mathbf{x})$$

  of k features, where (m_i, j_i) denotes the j_i'th feature in the m_i'th feature map (kernel)
- The same (kernel,feature) may appear several times in the product $\implies$ non-linear feature combinations within kernels and across kernels
- For $c > 0$ we also get all degree-h products of features, where $1 < h < k$

Preprocessing prior kernel combination

It is useful to preprocess the kernels $\kappa_1, \ldots, \kappa_P$ prior combining:

- Centering: to make the dataset zero mean; kernel will capture the co-variance of data items around the mean

$$\kappa_{i,c} = [\mathbf{I} - \frac{\mathbf{1}\mathbf{1}'}{\ell}]\kappa_i[\mathbf{I} - \frac{\mathbf{1}\mathbf{1}'}{\ell}]$$

- Normalization: projects the data on the unit sphere, removes the effect of the length of data vectors

$$\hat{\kappa}(\mathbf{x}, \mathbf{z}) = \frac{\kappa_i(\mathbf{x}, \mathbf{z})}{\sqrt{\kappa_i(\mathbf{x}, \mathbf{x})}\sqrt{\kappa_i(\mathbf{z}, \mathbf{z})}}$$

Intermediate fusion: learning kernel combinations

General recipe:

- Input a set of **base kernel** functions, $\{\kappa_m(\mathbf{x}_i^m, \mathbf{x}_j^m)\}_{m=1}^P$, for P feature representations of data instances
- Learn a combination function f_d to fuse the base kernels

$$\kappa_d(\mathbf{x}_i, \mathbf{x}_j) = f_d(\{\kappa_m(\mathbf{x}_i^m, \mathbf{x}_j^m)\}_{m=1}^P)$$

- The combination function, $f_d : \mathbb{R}^P \to \mathbb{R}$, can be a linear or a nonlinear function
- The combination should result in a combined kernel that is PSD

Linear kernel combinations

- To learn linear kernel combinations, we introduce kernel weights d_m:

$$\kappa_d(\mathbf{x}_i, \mathbf{x}_j) = \sum_{m=1}^{P} d_m \kappa_m(\mathbf{x}_i^m, \mathbf{x}_j^m)$$

- The weights d_m need to be constrained to ensure PSD property
- Typical restriction is non-negativity: $d_m \geq 0, m = 1, \ldots, P$: Non-negative combination of PSD base kernels is PSD.
- However, non-negativity is not strictly needed to ensure PSD property: can be seen as a computationally efficient way to guarantee PSD

Linear kernel combinations

- For non-negative combination of kernel weights $(d_1, \ldots, d_P)$, there is underlying feature representation

$$\phi_d(\mathbf{x}) = \left(\sqrt{d_1}\phi_1(\mathbf{x}), \ldots, \sqrt{d_P}\phi_P(\mathbf{x})\right)$$

with an inner product

$$\langle \phi_d(\mathbf{x}), \phi_d(\mathbf{z}) \rangle = \sum_{m=1}^{P} d_m \kappa_m(\mathbf{x}, \mathbf{z})$$

Learning linear combinations of kernels

General approaches:

- Two-step methods:
  - First approximate a target kernel through a combinations of kernels $\mathbf{K}_d \approx \mathbf{K}_y = \mathbf{y}\mathbf{y}'$, then learn a model using the approximated kernel
  - A popular approximation is to maximize the kernel alignment with the target (ALIGNF algorithm by Cortes et al. 2012)
- One-step methods:
  - Learn the kernel combination while learning the model: $f(\mathbf{x}) = \sum_{i=1}^{\ell} y_i \alpha_i \sum_{m=1}^{P} d_m \kappa_m(\mathbf{x}_i, \mathbf{x})$
  - Large number of variants, we will examine the simpleMKL algorithm (Rakotomamonjy et al., 2008)

Kernel alignment

- The alignment $A(\mathbf{K}_1, \mathbf{K}_2)$ between two kernel matrices $\mathbf{K}_1 = (\kappa_1(\mathbf{x}_i, \mathbf{x}_j))_{i,j=1}^{\ell}$ and $\mathbf{K}_2 = (\kappa_2(\mathbf{x}_i, \mathbf{x}_j))_{i,j=1}^{\ell}$ is given by

$$A(\mathbf{K}_1, \mathbf{K}_2) = \frac{\langle \mathbf{K}_1, \mathbf{K}_2 \rangle_F}{\sqrt{\langle \mathbf{K}_1, \mathbf{K}_1 \rangle_F \langle \mathbf{K}_2, \mathbf{K}_2 \rangle_F}}$$

- $\langle \mathbf{K}_1, \mathbf{K}_2 \rangle_F = \sum_{i=1}^{\ell} \sum_{j=1}^{\ell} \kappa_1(\mathbf{x}_i, \mathbf{x}_j) \kappa_2(\mathbf{x}_i, \mathbf{x}_j) = \mathrm{tr}\,(\mathbf{K}_1 \mathbf{K}_2)$ is the Frobenius inner product

Kernel alignment

- Alternative view: denote by $\mathbf{vec}(\mathbf{K})$ the concatenation of columns of $\mathbf{K}$ into a ℓ^2 -dimensional vector, then

$$A(\mathbf{K}_1, \mathbf{K}_2) = \frac{\langle \mathbf{vec}(\mathbf{K_1}), \mathbf{vec}(\mathbf{K_2}) \rangle}{\|\mathbf{vec}(\mathbf{K_1})\| \, \|\mathbf{vec}(\mathbf{K_2})\|},$$

- It can be viewed as the cosine of the angle between the matrices viewed as ℓ^2-dimensional vectors,
- Alignment can be shown to be always non-negative (Proof: Cortes et al. 2012): $0 \leq A(\mathbf{K}_1, \mathbf{K}_2) \leq 1$.

Kernel alignment

- The so called ideal target kernel for binary classification

$$\kappa_y(y_i, y_j) = y_i y_j = \begin{cases} 1 & \text{if } y_i = y_j, \\ -1 & \text{if } y_i \neq y_j \end{cases}$$

- The target kernel as a matrix

$$\mathbf{K}_y = (\kappa_y(y_i, y_j))_{i,j=1}^{\ell} = \mathbf{y}\mathbf{y}' = \begin{bmatrix} y_1 y_1 & y_1 y_2 & \cdots & y_1 y_\ell \\ y_2 y_1 & y_2 y_2 & \cdots & y_2 y_\ell \\ \vdots & \vdots & \ddots & \vdots \\ y_\ell y_1 & y_\ell y_2 & \cdots & y_\ell y_\ell \end{bmatrix}$$

  where $\mathbf{y} = (y_1, \ldots, y_\ell)$ contains the labels of the training set
- A good input kernel $\mathbf{K} \approx$ one with large $A(\mathbf{K}, \mathbf{K}_y)$
- Q: why is this ideal target?

Multiple kernel learning through Alignment maximization

$$\max_{d_m} A(\sum_{m=1}^{P} d_m \mathbf{K}_m, \mathbf{K}_y)$$
$$s.t. \sum_m d_m \mathbf{K}_m \succeq 0$$

- Objective asks for maximizing the alignment between the linear combination of input kernels and the target kernel $\mathbf{K}_y = \mathbf{y}\mathbf{y}'$
- Constraints require the resulting matrix to be positive semi-definite (symbol $\succeq$)
- This is an example of a semi-definite program (SDP), which is a convex optimization problem
- Despite being convex, SDPs are much harder to solve than, e.g. QPs, thus alternative formulations are used.

ALIGN: weighting kernels by centered kernel alignment[1]

- ALIGN uses alignment scores directly as input kernel weights $d_m = A(\mathbf{K}_{cm}, \mathbf{K}_y)$ i.e. no optimization is required
- $\mathbf{K}_{cm}$ is centered input kernel and $\mathbf{K}_{cy}$ is centered output kernel
- The combined kernel will be

$$\mathbf{K}_d = \sum_{m=1}^{P} d_m \mathbf{K}_{cm} = \sum_{m=1}^{P} A(\mathbf{K}_{cm}, \mathbf{K}_{cy}) \mathbf{K}_{cm}$$

- By definition, all d_m are non-negative, all $\mathbf{K}_m$ are PSD, thus $\mathbf{K}_d$ is PSD
- Easy computation: $A(\mathbf{K}, \mathbf{K}_y)$ takes $O(\ell^2)$ time, for $\mathbf{K}, \mathbf{K}_y \in \mathbb{R}^{\ell \times \ell}$ $\implies$ $O(P\ell^2)$ to compute $\mathbf{K}_d$

[1] Cortes et al. 2012

Centered kernel alignment

- Cortes et al. (2012) note that for best performance, one should center the kernels before alignment (in fact centering is the trick that allows better performance than unweighted sum of kernels)

$$\mathbf{K}_c = [\mathbf{I} - \frac{\mathbf{1}\mathbf{1}'}{n}]\mathbf{K}[\mathbf{I} - \frac{\mathbf{1}\mathbf{1}'}{n}]$$

- Then the centered alignment between $\mathbf{K}_1$ and $\mathbf{K}_2$ is defined by

$$A(\mathbf{K}_{c1}, \mathbf{K}_{c2}) = \frac{\langle \mathbf{K}_{c1}, \mathbf{K}_{c2} \rangle_F}{\|\mathbf{K}_{c1}\|_F \|\mathbf{K}_{c2}\|_F}.$$

- Above $\|\mathbf{K}\|_F = \sqrt{\langle \mathbf{K}, \mathbf{K} \rangle_F}$ denotes the Forbenius norm
- It can be shown that it suffices to center one of the kernels (Cortes et al 2012):

$$\langle \mathbf{K}_{c1}, \mathbf{K}_{c2} \rangle_F = \langle \mathbf{K}_{c1}, \mathbf{K}_2 \rangle_F = \langle \mathbf{K}_1, \mathbf{K}_{c2} \rangle_F$$

ALIGNF: maximizing centered kernel alignment

- Second method by Cortes et al. (2012) is based on alignment maximixation
- It solves the following optimization problem:

$$\max_{\boldsymbol{d}} \frac{\langle \sum_{m=1}^{P} d_m \mathbf{K}_{cm}, \mathbf{K}_y \rangle_F}{\|\mathbf{K}_d\|_F \, \|\mathbf{K}_y\|_F} = \max_{\boldsymbol{d}} \frac{\sum_{m=1}^{P} d_m \langle \mathbf{K}_{cm}, \mathbf{K}_y \rangle_F}{\sqrt{\sum_{m=1}^{P} \sum_{h=1}^{P} d_m d_h \langle \mathbf{K}_{cm}, \mathbf{K}_{ch} \rangle_F} \cdot \ell}$$

  where $\boldsymbol{d}$ satisfies $\|\boldsymbol{d}\|_2 = 1, d_m \geq 0$.
- The numerator asks to maximize linearly weighted alignment between input kernels and the target
- The denominator contains a weighted sum of all pairwise alignments of input kernels
- The combined kernel will be PSD due to non-negativity of $\boldsymbol{d}$ as the base kernels being PSD

Maximizing centered kernel alignment

- It can be shown (Cortes et al. 2012) that alignment maximization corresponds to standard QP

$$\begin{aligned} \text{minimize} \quad & -\boldsymbol{c}'\boldsymbol{d} + \frac{1}{2}\boldsymbol{d}'\mathbf{Q}\boldsymbol{d} \\ \text{subject to} \quad & \mathbf{1}'\boldsymbol{d} = 1 \\ & \mathbf{0} \leq \boldsymbol{d} \leq \mathbf{1} \end{aligned}$$

- Vector $\boldsymbol{c}$ contains the centered alignments between input kernels and the target
- Matrix Q contains the pairwise alignments of input kernels

$$\boldsymbol{c} = \begin{bmatrix} \langle \mathbf{K}_{c1}, \mathbf{K}_Y \rangle_F \\ \langle \mathbf{K}_{c2}, \mathbf{K}_Y \rangle_F \\ \vdots \\ \langle \mathbf{K}_{cP}, \mathbf{K}_Y \rangle_F \end{bmatrix} \quad \mathbf{Q} = \begin{bmatrix} \langle \mathbf{K}_{c1}, \mathbf{K}_{c1} \rangle_F & \langle \mathbf{K}_{c1}, \mathbf{K}_{c2} \rangle_F & \dots & \langle \mathbf{K}_{c1}, \mathbf{K}_{Pc} \rangle_F \\ \langle \mathbf{K}_{c2}, \mathbf{K}_{c1} \rangle_F & \langle \mathbf{K}_{c2}, \mathbf{K}_{c2} \rangle_F & \dots & \langle \mathbf{K}_{c2}, \mathbf{K}_{cP} \rangle_F \\ \vdots & \vdots & \ddots & \vdots \\ \langle \mathbf{K}_{cP}, \mathbf{K}_{c1} \rangle_F & \langle \mathbf{K}_{cP}, \mathbf{K}_{c2} \rangle_F & \dots & \langle \mathbf{K}_{cP}, \mathbf{K}_{cP} \rangle_F \end{bmatrix}$$

The second step: the case with SVM

- Plug in the combined kernel $\mathbf{K}_d = \sum_{m=1}^{P} d_n \mathbf{K}_m$ into dual SVM

$$\max_{\alpha} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j}^{\ell} \alpha_i \alpha_j \sum_{m=1}^{P} d_m \kappa_m(\mathbf{x}_i, \mathbf{x}_j)$$

$$s.t. 0 \leq \alpha_i \leq C, i = 1, \ldots, \ell \qquad \left(\sum_i \alpha_i y_i = 0 \right)$$

- Train SVM using any solver
- Note: the second step can also be regression or any other learning problem where suitable target kernel can be defined.

One-step approaches

- One-step approaches aim to learn the model (e.g. SVM classifier) and the kernel weights simultaneuosly
- For SVM, the MKL problem with linear kernel weights will be

$$\max_{\alpha,d} \sum_i \alpha_i - \frac{1}{2} \sum_{i,j}^{\ell} \alpha_i \alpha_j \sum_{m=1}^{P} d_m \kappa_m(\mathbf{x}_i, \mathbf{x}_j)$$

$$s.t.\ d \in H$$

$$0 \leq \alpha_i \leq C, i = 1, \ldots, \ell$$

$$\left(\sum_i \alpha_i y_i = 0 \right)$$

- Above H is some feasible set for the kernel weights, e.g. convex combination, $H = \{d \in \mathbb{R}^P | \sum_m d_m = 1, d_m \geq 0\}$

SimpleMKL (Rakotomamonjy, 2008)

- One-stage MKL method
- Learns a linear combination of kernels

$$\kappa_d(\mathbf{x}_i, \mathbf{x}_j) = \sum_{m=1}^{P} d_m \kappa_m(\mathbf{x}_i^m, \mathbf{x}_j^m)$$

- Kernel weights constrained to a convex combination $d_m \geq 0, \sum_m d_m = 1$
- Objective is to optimize SVM criteria and the optimal kernel weights together
- Bi-level optimization scheme: Uses SVM solver as a wrapper

SimpleMKL: the idea

- Can be interpreted as learning a mixture of classifiers
- One base classifier for each kernel, with kernel weight d_m, example weights α_i:

$$f_m(\mathbf{x}) = \langle \mathbf{w}_m, \phi_m(\mathbf{x}) \rangle = \sum_i \alpha_i y_i d_m \kappa_m(\mathbf{x}, \mathbf{x}_i)$$

- Above $\mathbf{w}_m = \sum_i \alpha_i y_i d_m \phi_m(\mathbf{x}_i)$ obtained from Lagrangian duality
- The full model will use a mixture of base classifiers:

$$f(\mathbf{x}) = \sum_{m=1}^{P} f_m(\mathbf{x}) = \sum_i \alpha_i y_i d_m \kappa_m(\mathbf{x}, \mathbf{x}_i)$$

SimpleMKL primal problem

$$\min_{\mathbf{w},b,\xi,d} \frac{1}{2} \sum_m \frac{1}{d_m} \|\mathbf{w}_m\|^2 + C \sum_i \xi_i$$

$$s.t.\ y_i \left(\sum_m \langle \mathbf{w}_m, \phi_m(\mathbf{x}_i) \rangle \ (+b) \right) \geq 1 - \xi_i, i = 1 \ldots, \ell$$

$$\xi_i \geq 0, i = 1 \ldots, \ell$$

$$\sum_m d_m = 1, d_m \geq 0, m = 1, \ldots, P$$

- Objective minimizes a linear combination of norms of weight vectors corresponding to different kernels plus slack for examples
- Constraints declare that the mixture classifier should achieve a large margin
- It is a convex problem (Rakotomamonjy, 2008)

Bi-level optimization problem: primal

- Reformulation as a bi-level optimization problem

$$\min_{\boldsymbol{d} \in H} J(\boldsymbol{d})$$

$$s.t.\ J(\boldsymbol{d}) = \begin{cases} \min_{\mathbf{w},b,\xi} & \frac{1}{2}\sum_m \frac{1}{d_m}\|\mathbf{w}_m\|^2 + C\sum_i \xi_i \\ s.t. & y_i\left(\sum_m \langle \mathbf{w}_m, \phi_m(\mathbf{x}_i)\rangle\ (+b)\right) \geq 1-\xi_i \\ & \xi_i \geq 0, i = 1\ldots,\ell \end{cases}$$

- In outer loop optimize kernel weights $\boldsymbol{d}$
- Inner loop corresponds to a primal soft-margin SVM using a combined kernel with current kernel weights

Bi-level optimization problem: dual

- We can plug in the dual of the SVM problem in the inner loop

$$\min_{\boldsymbol{d} \in H} \; J(\boldsymbol{d})$$

$$s.t.\; J(\boldsymbol{d}) = \begin{cases} \max_{\alpha} & \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j \sum_m d_m \kappa_m(\mathbf{x}_i, \mathbf{x}_j) \\ s.t. & 0 \leq \alpha_i \leq C \\ & (\sum_i \alpha_i y_i = 0) \end{cases}$$

- In outer loop optimize kernel weights $\boldsymbol{d}$
- Inner loop corresponds to a dual soft-margin SVM using a combined kernel with current kernel weights

Optimization in the outer loop

- Consider minimization of $J(\boldsymbol{d})$ with respect to kernel weights (keeping α fixed)

$$J(\boldsymbol{d}) = \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j \sum_m d_m \kappa_m(\mathbf{x}_i, \mathbf{x}_j)$$

- The negative gradient gives descent directions:

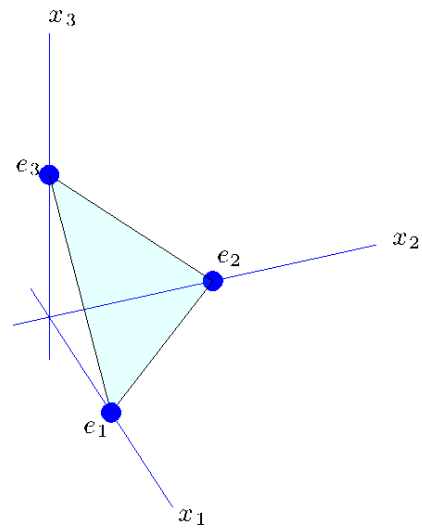
$$-\nabla J(\boldsymbol{d}) = \frac{1}{2} \left(\sum_{i,j} \alpha_i \alpha_j \kappa_m(\mathbf{x}_i, \mathbf{x}_j) \right)_{m=1}^{P}$$

Feasible set of the outer problem

- The feasible set of the outer problem is a simplex

$$H = \{\boldsymbol{d} \in \mathbb{R}^P | \sum_m d_m = 1, d_m \geq 0\},$$

- Vertices of the simplex correspond to individual kernels receiving all the weight: $d_m = 1$
- Edges and faces of the simplex correspond to linear combinations of subsets of the kernels, with some kernels at $d_m = 0$
- Interior of the simplex correspond to linear combinations of all kernels $d_m > 0$

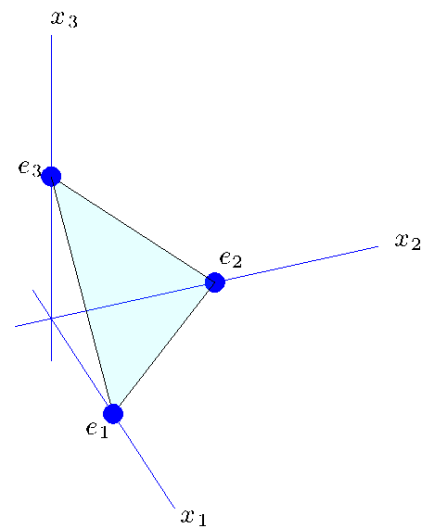


Feasible set of the outer problem

- We will look for an update $\boldsymbol{d}^{(t+1)} = \boldsymbol{d}^{(t)} + \gamma D$ that will keep us within the simplex
- We will choose the descent direction D chosen so that $\sum_m d_m = 1$ maintained regardless of step size
- Need to satisfy

$$\sum_m d_m^{(t+1)} = \sum_m \left(d_m^{(t)} + \gamma D_m \right) = 1, \ \forall$$

$$\implies \sum_m D_m = 0$$



Optimization in the outer loop

- Need to satisfy $\sum_m d_m^{(t)} + \gamma D_m = 1$ for all $t \implies \sum_m D_m = 0$
- Setting D as the **reduced gradient** satisfies it

$$D = \begin{bmatrix} -\frac{\partial J(\boldsymbol{d})}{\partial d_1} \\ \vdots \\ \sum_{m \neq k} \frac{\partial J(\boldsymbol{d})}{\partial d_m} \\ \vdots \\ -\frac{\partial J(\boldsymbol{d})}{\partial d_P} \end{bmatrix} + \begin{bmatrix} \frac{\partial J(\boldsymbol{d})}{\partial d_k} \\ \vdots \\ -(P-1)\frac{\partial J(\boldsymbol{d})}{\partial d_k} \\ \vdots \\ \frac{\partial J(\boldsymbol{d})}{\partial d_k} \end{bmatrix}$$

- Intuitively:
  - One component k is chosen that will be heavily updated in the direction of the negative gradient
  - The other components receive reduced updates by corresponding amount
- Can be shown to be a descent direction[2]

[2] D. Bertsekas, *Non-linear programming*, 1999, p. 197

Stepsize

- The component k is chosen by $k = \mathbf{argmax}_m d_m$ (for numerical stability according to the authors)
- Stepsize γ_{max} is chosen to be the largest that keeps the positivity constraints $d_m \geq 0, m = 1, \ldots, P$ satisfied
- Let us find the component $\tilde{m}$ that first hits $d_m = 0$ when we step in the update direction D:

$$\boldsymbol{d}^{new} = \boldsymbol{d}^{(t)} + \gamma_{max} D$$

- For a component d_m, the maximum stepsize γ_m is obtained from

$$d_m + \gamma_m D_m = 0 \implies \gamma_m = -d_m / D_m$$

- The component first to reach $d_m = 0$ is $\tilde{m} = \mathbf{argmin}_m \gamma_m$
- The maximum stepsize to keep all $d_m \geq 0$ is $\gamma_{max} = \gamma_{\tilde{m}}$

Updated reduced gradient

- At $\boldsymbol{d}^{new}$ we have $d_{\tilde{m}}^{new} = 0$, and we need to update the reduced gradient to exclude moving turhter in that direction
- Assuming $\tilde{m} = 1$, we set $d_{\tilde{m}} = 0$, and renormalize the reduced gradient keeping $D_{\tilde{m}}^{new} = 0$:

$$D^{new} = \begin{bmatrix} 0 \\ -\frac{\partial J(\boldsymbol{d})}{\partial d_1} \\ \vdots \\ \sum_{m \neq k, D_m \neq 0} \frac{\partial J(\boldsymbol{d})}{\partial d_m} \\ \vdots \\ -\frac{\partial J(\boldsymbol{d})}{\partial d_P} \end{bmatrix} + \begin{bmatrix} 0 \\ \frac{\partial J(\boldsymbol{d})}{\partial d_k} \\ \vdots \\ -(P-2)\frac{\partial J(\boldsymbol{d})}{\partial d_k} \\ \vdots \\ \frac{\partial J(\boldsymbol{d})}{\partial d_k} \end{bmatrix}$$

Computing updates

- The objective $J(\boldsymbol{d}^{new})$ is evaluated by SVM
- If the objective value statisfies $J(\boldsymbol{d}^{new}) < J(\boldsymbol{d}^{t})$ the update is made $\boldsymbol{d}^{t+1} = \boldsymbol{d}^{new}$, $D^{t+1} = D^{new}$ and the procedure is repeated: finding the maximal update in the direction of D^{t+1} and updating
- Otherwise, line search is conducted in the interval $[0, \gamma_{max}]$ to find the local optimum in that direction
- SVM is called for every step in the line search
  - Exact line search is not applicable, need to compute several points to find the optimum
  - E.g. Direct search: pick two new points d', d'' within the interval, find the minimum, exclude whichever outer points are not adjacent to the minimum point, make a new interval of the remaining three points, and repeat

SimpleMKL algorithm

Algorithm 1 SimpleMKL algorithm

set $d_m = \frac{1}{M}$ for $m = 1, \ldots, M$
while stopping criterion not met **do**
  compute $J(d)$ by using an SVM solver with $K = \sum_m d_m K_m$
  compute $\frac{\partial J}{\partial d_m}$ for $m = 1, \ldots, M$ and descent direction D (12).
  set $\mu = \underset{m}{\operatorname{argmax}}\, d_m$, $J^\dagger = 0$, $d^\dagger = d$, $D^\dagger = D$
  while $J^\dagger < J(d)$ **do** {descent direction update}
    $d = d^\dagger$, $D = D^\dagger$
    $\nu = \underset{\{m|D_m<0\}}{\operatorname{argmin}} -d_m/D_m$, $\gamma_{\max} = -d_\nu/D_\nu$
    $d^\dagger = d + \gamma_{\max} D$, $D^\dagger_\mu = D_\mu - D_\nu$, $D^\dagger_\nu = 0$
    compute $J^\dagger$ by using an SVM solver with $K = \sum_m d^\dagger_m K_m$
  end while
  line search along D for $\gamma \in [0, \gamma_{\max}]$ {calls an SVM solver for each γ trial value}
  $d \leftarrow d + \gamma D$
end while