Optimizing Ranking Measures for Compact Binary Code Learning

Guosheng Lin[1], Chunhua Shen[1,⋆], and Jianxin Wu[2]

[1] University of Adelaide, Australia
chunhua.shen@adelaide.edu.au
[2] Nanjing University, China

Abstract. Hashing has proven a valuable tool for large-scale information retrieval. Despite much success, existing hashing methods optimize over simple objectives such as the reconstruction error or graph Laplacian related loss functions, instead of the performance evaluation criteria of interest—multivariate performance measures such as the AUC and NDCG. Here we present a general framework (termed StructHash) that allows one to directly optimize multivariate performance measures. The resulting optimization problem can involve exponentially or infinitely many variables and constraints, which is more challenging than standard structured output learning. To solve the StructHash optimization problem, we use a combination of column generation and cutting-plane techniques. We demonstrate the generality of StructHash by applying it to ranking prediction and image retrieval, and show that it outperforms a few state-of-the-art hashing methods.

1 Introduction

The ever increasing volumes of imagery available, and the benefits reaped through the interrogation of large image datasets, have increased enthusiasm for large-scale approaches to vision. One of the simplest, and most effective means of improving the scale and efficiency of an application has been to use hashing to pre-process the data [10,24,14,20,16,13].

Depending on applications, specific measures are used to evaluate the performance of the generated hash codes. For example, information retrieval and ranking criteria [17] such as the Area Under the ROC Curve (AUC) [7], Normalized Discounted Cumulative Gain (NDCG) [6], Precision-at-K, Precision-Recall and Mean Average Precision (mAP) have been widely adopted to evaluate the success of hashing methods. However, to date, most hashing methods are usually learned by optimizing simple errors such as the reconstruction error (e.g., binary reconstruction embedding hashing [10]) or the graph Laplacian related loss [26,16,24]. These hashing methods construct a set of hash functions that map the original high-dimensional data into a much smaller binary space, typically with the goal of preserving neighbourhood relations. The resulting compact

⋆ Corresponding author.

D. Fleet et al. (Eds.): ECCV 2014, Part III, LNCS 8691, pp. 613–627, 2014.

binary encoding enables fast similarity computation between data points by using the Hamming distance, which can be carried out by rapid, often hardware-supported, bit-wise operations. Furthermore, compact binary codes are much more efficient for large-scale data storage.

To our knowledge, none of the existing hashing methods has tried to learn hash codes that *directly* optimize a multivariate performance criterion. In this work, we seek to reduce the discrepancy between existing learning criteria and the evaluation criteria (such as retrieval quality measures).

The proposed framework accommodates various complex multivariate measures. By observing that the hash codes learning problem is essentially an information retrieval problem, various ranking loss functions can and should be applied, rather than merely pairwise distance comparisons. This framework also allows to introduce more general definitions of "similarity" to hashing beyond existing ones.

In summary, our main contributions are as follows.

1. We propose a flexible binary hash codes learning framework that directly optimizes complex multivariate measures. This framework, *for the first time*, exploits the gains made in structured output learning for the purposes of hashing. Our hashing method, labelled as StructHash, is able to directly optimize various multivariate evaluation criteria, such as information retrieval measures (e.g., AUC and NDCG [6]).
2. To facilitate StructHash, we combine column generation and cutting-plane methods in order to efficiently solve the resulting optimization problem, which may involve exponentially or even infinitely many variables and constraints.
3. Applied to ranking prediction for image retrieval, the proposed method demonstrates state-of-the-art performance on hash function learning.

2 Related Work

One of the best known *data-independent* hashing methods is locality sensitive hashing (LSH) [3], which uses random projection to generate binary codes. Recently, a number of *data-dependent* hashing methods have been proposed. For example, spectral hashing (SPH) [24] aims to preserve the neighbourhood relation by optimizing the Laplacian affinity. Anchor graph hashing (AGH) [16] makes the original SPH much more scalable. Examples of supervised or semi-supervised hashing methods include binary reconstruction embedding (BRE) [10], which aims to minimize the expected distances; and the semi-supervised sequential projection learning hashing (SPLH) [22], which enforces the smoothness of similar data points and the separability of dissimilar data points.

To obtain a richer representation, kernelized LSH [11] was proposed, which randomly samples training data as support vectors, and randomly draws the dual coefficients from a Gaussian distribution. Liu et al. extended Kulis and Grauman's work to kernelized supervised hashing (KSH) [15] by learning the dual coefficients instead. Lin et al. [13] employed ensembles of decision trees as

the hash functions. Nonetheless, all of these methods do not directly optimize the multivariate performance measures of interest. We formulate hash codes learning as a structured output learning problem, in order to directly optimize a wide variety of evaluation measures.

This work is primarily inspired by recent advances in learning to rank such as the metric learning method in [17], which directly optimizes several different ranking measures. We aim to learn hash functions, which leads to a very different learning task preventing directly applying techniques in [17]. We are also inspired by the recent column generation based hashing method, column generation hashing (CGH) [12], which iteratively learns hash functions using column generation. However, their method optimizes the conventional classification-related loss, which is much simpler than the multivariate loss that we are interested in here. Moreover, the optimization of CGH relies on all triplet constraints while our method is able to use much less number of constraints without sacrificing the performance.

Our framework is built upon the structured SVM [21], which has been applied to many applications for complex structured output prediction, e.g., image segmentation, action recognition and so on.

Notation. Let $\{(\boldsymbol{x}_i; \boldsymbol{y}_i)\}$, $i = 1, 2 \cdots$, denote a set of input-output pairs. The discriminative function for structured output prediction is[1] $F(\boldsymbol{x}, \boldsymbol{y}) : \mathcal{X} \times \mathcal{Y} \mapsto \mathbb{R}$, which measures the compatibility of the input and output pair $(\boldsymbol{x}, \boldsymbol{y})$. Given a query $\boldsymbol{x}_i$, we use $\mathcal{X}_i^+$ and $\mathcal{X}_i^-$ to denote the subsets of relevant and irrelevant data points in the training data. Given two data points: $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, $\boldsymbol{x}_i \prec_{\boldsymbol{y}} \boldsymbol{x}_j$ ($\boldsymbol{x}_i \succ_{\boldsymbol{y}} \boldsymbol{x}_j$) means that $\boldsymbol{x}_i$ is placed before (after) $\boldsymbol{x}_j$ in the ranking $\boldsymbol{y}$.

2.1 Structured SVM

First we provide a brief overview of structured SVM. Structured SVM enforces that the score of the "correct" model $\boldsymbol{y}''$ should be larger than all other "incorrect" model $\boldsymbol{y}$, $\forall \boldsymbol{y} \neq \boldsymbol{y}''$, which writes:

$$\forall \boldsymbol{y} \in \mathcal{Y}: \quad \boldsymbol{w}^\top [\psi(\boldsymbol{x}, \boldsymbol{y}'') - \psi(\boldsymbol{x}, \boldsymbol{y})] \geq \Delta(\boldsymbol{y}, \boldsymbol{y}'') - \xi. \tag{1}$$

Here ξ is a slack variable (soft margin) corresponding to the hinge loss. $\psi(\boldsymbol{x}, \boldsymbol{y})$ is a vector-valued joint feature mapping. It plays a key role in structured learning and specifies the relationship between an input $\boldsymbol{x}$ and output $\boldsymbol{y}$. $\boldsymbol{w}$ is the model parameter. The label loss $\Delta(\boldsymbol{y}, \boldsymbol{y}'') \in \mathbb{R}$ measures the discrepancy of the predicted $\boldsymbol{y}$ and the true label $\boldsymbol{y}''$. A typical assumption is that $\Delta(\boldsymbol{y}, \boldsymbol{y}) = 0, \Delta(\boldsymbol{y}, \boldsymbol{y}'') > 0$ for any $\boldsymbol{y} \neq \boldsymbol{y}''$, and $\Delta(\boldsymbol{y}, \boldsymbol{y}'')$ is upper bounded. The prediction $\boldsymbol{y}^*$ of an input $\boldsymbol{x}$ is achieved by

$$\boldsymbol{y}^* = \operatorname*{argmax}_{\boldsymbol{y}} F(\boldsymbol{x}, \boldsymbol{y}) = \boldsymbol{w}^\top \psi(\boldsymbol{x}, \boldsymbol{y}). \tag{2}$$

[1] To be precise, the prediction function should be written as $F(\boldsymbol{x}, \boldsymbol{y}; \boldsymbol{w})$ because it is parameterized by $\boldsymbol{w}$. For simplicity, we omit $\boldsymbol{w}$.

Optimization for the Model Parameter $\boldsymbol{w}$. For structured problems, the size of the output $|\mathcal{Y}|$ is typically very large or infinite. Considering all possible constraints in (1) is generally intractable. The cutting-plane method [9] is commonly employed, which allows to maintain a small working-set of constraints and obtain an approximate solution of the original problem up to a pre-set precision. To speed up, the 1-slack reformulation is proposed [8]. Nonetheless the cutting-plane method needs to find the most violated label (equivalent to an inference problem)

$$\underset{\boldsymbol{y}\in\mathcal{Y}}{\operatorname{argmax}} \ \boldsymbol{w}^\top \psi(\boldsymbol{x},\boldsymbol{y}) + \Delta(\boldsymbol{y},\boldsymbol{y}''). \tag{3}$$

Structured SVM typically requires: 1) a well-designed feature representation $\psi(\cdot,\cdot)$; 2) an appropriate label loss $\Delta(\cdot,\cdot)$; 3) solving inference problems (2) and (3) efficiently.

Ranking Prediction with Structured Output. In a retrieval system, given a test data point $\boldsymbol{x}$, the goal is to predict a ranking of data points in the database. For a "correct" ranking, relevant data points are expected to be placed in front of irrelevant data points. A ranking output is denoted by $\boldsymbol{y}$. Let us introduce a symbol $y_{jk} = 1$ if $\boldsymbol{x}_j \prec_{\boldsymbol{y}} \boldsymbol{x}_k$ and $y_{jk} = -1$ if $\boldsymbol{x}_j \succ_{\boldsymbol{y}} \boldsymbol{x}_k$. The ranking can be evaluated by various measures such as AUC, NDCG, mAP. These evaluation measures can be optimized directly as label losss Δ [7,17]. Here $\psi(\boldsymbol{x},\boldsymbol{y})$ can be defined as:

$$\psi(\boldsymbol{x}_i,\boldsymbol{y}) = \sum_{\boldsymbol{x}_j\in\mathcal{X}_i^+}\sum_{\boldsymbol{x}_k\in\mathcal{X}_i^-} y_{jk}\Big[\frac{\phi(\boldsymbol{x}_i,\boldsymbol{x}_j)-\phi(\boldsymbol{x}_i,\boldsymbol{x}_k)}{|\,\mathcal{X}_i^+\,|\cdot|\,\mathcal{X}_i^-\,|}\Big]. \tag{4}$$

$\mathcal{X}_i^+$ and $\mathcal{X}_i^-$ are the sets of relevant and irrelevant neighbours of data point $\boldsymbol{x}_i$ respectively. Here $|\cdot|$ is the set size. The feature map $\phi(\boldsymbol{x}_i,\boldsymbol{x}_j)$ captures the relation between a query $\boldsymbol{x}_i$ and point $\boldsymbol{x}_j$.

We have briefly reviewed how to optimize ranking criteria using structured prediction. Now we review some basic concepts of hashing before introducing our framework.

2.2 Learning-Based Hashing

Given a set of training data $\boldsymbol{x}_i$, $(i = 1, 2, \dots)$, the task is to learn a set of hash functions $[h_1(\boldsymbol{x}), h_2(\boldsymbol{x}), \dots, h_\ell(\boldsymbol{x})]$. Each hash function maps the input into a binary bit $\{0, 1\}$. So with the learned functions, an input $\boldsymbol{x}$ is mapped into a binary code of length ℓ. We use $\widetilde{\boldsymbol{x}} \in \{0,1\}^\ell$ to denote the hashed values of $\boldsymbol{x}$ i.e.,

$$\widetilde{\boldsymbol{x}} = [h_1(\boldsymbol{x}) \dots, h_\ell(\boldsymbol{x})]^\top. \tag{5}$$

Suppose that we are given the supervision information as a set of triplets: $\{(\boldsymbol{x}_i,\boldsymbol{x}_j,\boldsymbol{x}_k)\}(i = 1, 2, \dots)$, in which $\boldsymbol{x}_j$ is an relevant (similar) data point of

$\boldsymbol{x}_i$ (i.e., $\boldsymbol{x}_j \in \mathcal{X}_i^+$) and $\boldsymbol{x}_k$ is an irrelevant (dissimilar) neighbor of $\boldsymbol{x}_i$ (i.e., $\boldsymbol{x}_k \in \mathcal{X}_i^-$). These triplets encode the relative similarity information. After applying the hashing, the distance of two hash codes (of length ℓ) can be calculated using the weighted hamming distance:

$$d_{\mathsf{hm}}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{w}^\top |\widetilde{\boldsymbol{x}}_i - \widetilde{\boldsymbol{x}}_j|, \tag{6}$$

where $\boldsymbol{w} > 0$ is a non-negative vector, which can be learned. Such weighted hamming distance is used in CGH [12] and multi-dimension spectral hashing [23]. It is expected that after hashing, the distance between relevant data points should be smaller than the distance between irrelevant data points. That is

$$d_{\mathsf{hm}}(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq d_{\mathsf{hm}}(\boldsymbol{x}_i, \boldsymbol{x}_k),$$

for $\boldsymbol{x}_j \in \mathcal{X}_i^+, \boldsymbol{x}_k \in \mathcal{X}_i^-, \forall i = 1, 2, \cdots$. One can then define the margin $\rho = d_{\mathsf{hm}}(\boldsymbol{x}_i, \boldsymbol{x}_k) - d_{\mathsf{hm}}(\boldsymbol{x}_i, \boldsymbol{x}_j)$. It is then possible to plug this margin into the large-margin learning framework to optimize for the parameter $\boldsymbol{w}$ as well as the hash functions, as shown in [12].

3 The Proposed StructHash Algorithm

For the time being, let us assume that we have already learned all the hashing functions. In other words, *given a data point $\boldsymbol{x}$, we assume that we have access to its corresponding hashed values $\widetilde{\boldsymbol{x}}$, as defined in* (5). Later we will show how this mapping can be explicitly learned using column generation. Now let us focus on how to optimize for the weight $\boldsymbol{w}$. When the weighted hamming distance is used, we aim to learn an optimal weight $\boldsymbol{w}$ defined in (6). Distances are calculated in the learned space and ranked accordingly. A natural choice for the vector-valued mapping function ϕ in Equ. (4) is

$$\phi(\boldsymbol{x}_i, \boldsymbol{x}_j) = -|\widetilde{\boldsymbol{x}}_i - \widetilde{\boldsymbol{x}}_j|. \tag{7}$$

Note that we have flipped the sign, which preserves the ordering in the standard structured SVM. Due to this change of sign, sorting the data by ascending $d_{\mathsf{hm}}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is equivalent to sorting by descending $\boldsymbol{w}^\top \phi(\boldsymbol{x}_i, \boldsymbol{x}_j) = -\boldsymbol{w}^\top |\widetilde{\boldsymbol{x}}_i - \widetilde{\boldsymbol{x}}_j|$.

The loss function $\Delta(\cdot, \cdot)$ depends on the metric, which we will discuss in detail in the next section. For ease of exposition, let us define

$$\delta\psi_i(\boldsymbol{y}) = \psi(\boldsymbol{x}_i, \boldsymbol{y}_i) - \psi(\boldsymbol{x}_i, \boldsymbol{y}), \tag{8}$$

with $\psi(\boldsymbol{x}_i, \boldsymbol{y})$ defined in (4). We consider the following problem,

$$\min_{\boldsymbol{w} \geq 0, \boldsymbol{\xi} \geq 0} \ \|\boldsymbol{w}\|_1 + \tfrac{C}{m} \sum_{i=1}^{m} \xi_i \tag{9a}$$

$$\text{s.t.: } \forall i = 1, \ldots, m \text{ and } \forall \boldsymbol{y} \in \mathcal{Y}:$$

$$\boldsymbol{w}^\top \delta\psi_i(\boldsymbol{y}) \geq \Delta(\boldsymbol{y}_i, \boldsymbol{y}) - \xi_i. \tag{9b}$$

Unlike standard structured SVM, here we use the ℓ_1 regularisation (instead of ℓ_2) and enforce that $\boldsymbol{w}$ to be non-negative. This is aligned with boosting methods [2,18], and enables us to learn hash functions efficiently.

Algorithm 1. StructHash: Column generation for hash function learning

1: **Input:** training examples $(\boldsymbol{x}_1; \boldsymbol{y}_1), (\boldsymbol{x}_2; \boldsymbol{y}_2), \cdots$; parameter C; the maximum iteration number (bit length ℓ).
2: **Initialise:** working set of hashing functions $\mathcal{W}_{\mathrm{H}} \leftarrow \emptyset$; initialise $\lambda_{(\boldsymbol{c},\boldsymbol{y})} = C/m$ by randomly picking m pairs of $(\boldsymbol{c}, \boldsymbol{y})$ and the rest is set to 0.
3: **Repeat**
4: – Find a new hashing function $h^*(\cdot)$ by solving Equ. (15).
5: – add h^* into the working set of hashing functions: $\mathcal{W}_{\mathrm{H}}$.
6: – Solve the structured SVM problem (9) or the equivalent (10) using cutting-plane as discussed in Sec. 3.1.
7: **Until** the maximum iteration is reached.
8: **Output:** Learned hash functions and $\boldsymbol{w}$.

3.1 Learning Weights $\boldsymbol{w}$ via Cutting-Plane

Here we show how to learn $\boldsymbol{w}$. Inspired by [8], we first derive the 1-slack formulation of the original m-slack formulation (9):

$$\min_{\boldsymbol{w}\geq 0, \xi \geq 0} \quad \|\boldsymbol{w}\|_1 + C\xi \tag{10a}$$

$$\text{s.t.:} \ \forall \boldsymbol{c} \in \{0,1\}^m \text{ and } \forall \boldsymbol{y} \in \mathcal{Y}, i = 1, \cdots, m:$$

$$\frac{1}{m}\boldsymbol{w}^\top \left[\sum_{i=1}^{m} c_i \cdot \delta\psi_i(\boldsymbol{y})\right] \geq \frac{1}{m}\sum_{i=1}^{m} c_i \Delta(\boldsymbol{y}_i, \boldsymbol{y}) - \xi. \tag{10b}$$

Here $\boldsymbol{c}$ enumerates all possible $\boldsymbol{c} \in \{0,1\}^m$. As in [8], cutting-plane methods can be used to solve the 1-slack primal problem (10) efficiently. Specifically, we need to solve a maximization for every $\boldsymbol{x}_i$ in each cutting-plane iteration to find the most violated constraint of (10b), given a solution $\boldsymbol{w}$:

$$\boldsymbol{y}_i^* = \operatorname*{argmax}_{\boldsymbol{y}} \ \Delta(\boldsymbol{y}_i, \boldsymbol{y}) - \boldsymbol{w}^\top \delta\psi_i(\boldsymbol{y}). \tag{11}$$

We now know how to efficiently learn $\boldsymbol{w}$ using cutting-plane methods. However, it remains unclear how to learn hash functions (or features). Thus far, we have taken for granted that the hashed values $\widetilde{\boldsymbol{x}}$ (or $h(\cdot)$) are given. We would like to learn the hash functions and $\boldsymbol{w}$ in a single optimization framework. Next we show how this is possible using the column generation technique from boosting.

3.2 Learning Hash Functions Using Column Generation

Note that the dimension of $\boldsymbol{w}$ is the same as the dimension of $\widetilde{\boldsymbol{x}}$ (and of $\phi(\cdot,\cdot)$, see Equ. (7)), which is the number of hash bits by the definition (5). If we were able to access all hash functions, it may be possible to select a subset of them and learn the corresponding $\boldsymbol{w}$ due to the sparsity introduced by the ℓ_1 regularization in (9). Unfortunately, the number of possible hash functions can be infinitely large. In this case it is in general infeasible to solve the optimization problem exactly.

Column generation [2] can be used to approximately solve the problem by adding variables iteratively into the master optimization problems. Column generation was originally invented to solve extremely large-scale linear programming problem, which mainly works on the dual problem. The basic concept of column generation is to add one constraint at a time to the dual problem until an optimal solution is identified. Columns[2] are generated and added to the problem iteratively to approach the optimality. In the primal problem, column generation solves the problem on a subset of primal variables ($\boldsymbol{w}$ in our case), which corresponds to a subset of constraints in the dual. This strategy has been widely employed to learn weak learners in boosting [18,2,19].

To learn hash functions via column generation, we derive the dual problem of the above 1-slack optimization, which is,

$$\max_{\boldsymbol{\lambda}\geq 0} \quad \sum_{\boldsymbol{c},\boldsymbol{y}} \lambda_{(\boldsymbol{c},\boldsymbol{y})} \sum_{i=1}^{m} c_i \Delta(\boldsymbol{y}_i, \boldsymbol{y}) \tag{12a}$$

$$\text{s.t.:} \quad \frac{1}{m} \sum_{\boldsymbol{c},\boldsymbol{y}} \lambda_{(\boldsymbol{c},\boldsymbol{y})} \left[\sum_{i=1}^{m} c_i \cdot \delta\psi_i(\boldsymbol{y}) \right] \leq \mathbf{1}, \tag{12b}$$

$$0 \leq \sum_{\boldsymbol{c},\boldsymbol{y}} \lambda_{(\boldsymbol{c},\boldsymbol{y})} \leq C. \tag{12c}$$

We denote by $\lambda_{(\boldsymbol{c},\boldsymbol{y})}$ the 1-slack dual variable associated with one constraint in (10b). Note that (12b) is a set of constraints because $\delta\psi(\cdot)$ is a vector of the same dimension as $\phi(\cdot,\cdot)$ as well as $\widetilde{\boldsymbol{x}}$, which can be infinitely large. One dimension in the vector $\delta\psi(\cdot)$ corresponds to one constraint in (12b). Finding the most violated constraint in the dual form (12) of the 1-slack formulation for generating one hash function is to maximise the l.h.s. of (12b).

The calculation of $\delta\psi(\cdot)$ in (8) can be simplified as follows. Because of the subtraction of $\psi(\cdot)$ (defined in (4)), only those incorrect ranking pairs will appear in the calculation. Recall that the true ranking is $\boldsymbol{y}_i$ for $\boldsymbol{x}_i$. We define $\mathcal{S}_i(\boldsymbol{y})$ as a set of incorrectly ranked pairs: $(j,k) \in \mathcal{S}_i(\boldsymbol{y})$, in which the incorrectly ranked pair (j,k) means that the true ranking is $\boldsymbol{x}_j \prec_{\boldsymbol{y}_i} \boldsymbol{x}_k$ but $\boldsymbol{x}_j \succ_{\boldsymbol{y}} \boldsymbol{x}_k$. So we have

$$\begin{aligned} \delta\psi_i(\boldsymbol{y}) &= \tfrac{2}{|\,\mathcal{X}_i^+\,||\,\mathcal{X}_i^-\,|} \sum_{(j,k)\in\mathcal{S}_i(\boldsymbol{y})} \big[\phi(\boldsymbol{x}_i,\boldsymbol{x}_j) - \phi(\boldsymbol{x}_i,\boldsymbol{x}_k)\big] \\ &= \tfrac{2}{|\,\mathcal{X}_i^+\,||\,\mathcal{X}_i^-\,|} \sum_{(j,k)\in\mathcal{S}_i(\boldsymbol{y})} \big(|\widetilde{\boldsymbol{x}}_i - \widetilde{\boldsymbol{x}}_k| - |\widetilde{\boldsymbol{x}}_i - \widetilde{\boldsymbol{x}}_j|\big). \end{aligned} \tag{13}$$

[2] A column is a variable in the primal and a corresponding constraint in the dual.

With the above equations and the definition of $\widetilde{\boldsymbol{x}}$ in (5), the most violated constraint in (12b) can be found by solving the following problem:

$$h^*(\cdot) = \operatorname*{argmax}_{h(\cdot)} \sum_{\boldsymbol{c},\boldsymbol{y}} \lambda_{(\boldsymbol{c},\boldsymbol{y})} \sum_i \frac{2c_i}{|\,\mathcal{X}_i^+\,||\,\mathcal{X}_i^-\,|} \cdot \sum_{(j,k)\in\mathcal{S}_i(\boldsymbol{y})} \big(|h(\boldsymbol{x}_i) - h(\boldsymbol{x}_k)| - |h(\boldsymbol{x}_i) - h(\boldsymbol{x}_j)|\big). \tag{14}$$

By exchanging the order of summations, the above optimization can be further written in a compact form:

$$h^*(\cdot) = \operatorname*{argmax}_{h(\cdot)} \sum_{i,\boldsymbol{y}} \sum_{(j,k)\in\mathcal{S}_i(\boldsymbol{y})} \mu_{(i,\boldsymbol{y})} \big(|h(\boldsymbol{x}_i) - h(\boldsymbol{x}_k)| - |h(\boldsymbol{x}_i) - h(\boldsymbol{x}_j)|\big), \tag{15}$$

$$\text{where, } \mu_{(i,\boldsymbol{y})} = \frac{2}{|\,\mathcal{X}_i^+\,||\,\mathcal{X}_i^-\,|} \sum_{\boldsymbol{c}} \lambda_{(\boldsymbol{c},\boldsymbol{y})} c_i. \tag{16}$$

The objective in the above optimization is a summation of weighted triplet (i, j, k) ranking scores, in which $\mu_{(i,\boldsymbol{y})}$ is the triplet weighting value. Solving the above optimization provides the best hash function for the current solution $\boldsymbol{w}$. Once a hash function is generated, we learn $\boldsymbol{w}$ using cutting-plane in Sec. 3.1. The column generation procedure for hash function learning is summarised in Algorithm 1.

The form of hash function $h(\cdot)$ can be any function that outputs a binary value. For a decision stump as the hash function, usually we can exhaustively enumerate all possibility and find the globally best one. However globally solving (15) is generally difficult. In most of our experiments, we use the linear perceptron hash function with the output in $\{0, 1\}$:

$$h(\boldsymbol{x}) = 0.5(\text{sign}(\boldsymbol{v}^\top \boldsymbol{x} + b) + 1). \tag{17}$$

The non-smooth function sign$(\cdot)$ here brings the difficulty for optimization. Similar to [12], we replace the sign$(\cdot)$ function by a smooth sigmoid function, and then locally solve the above optimization (15) (e.g., LBFGS [27]) for learning the parameters of a hash function. We can apply a few heuristics to initialize for solving (15). For example, similar to LSH, we can generate a set of random projection planes and then choose the best one that maximizes the objective in (15) as the initialization. We can also train a decision stump by searching a best dimension and threshold to maximize the objective on the quantized data. Alternatively, one can employ the spectral relaxation method [16] which drops the sign$(\cdot)$ function and solves a generalized eigenvalue problem to obtain an initial point. In our experiments, we use the spectral relaxation method for initialization.

Next, we discuss some widely-used information retrieval evaluation criteria, and show how they can be seamlessly incorporated into StructHash.

4 Ranking Measures

Here we discuss a few ranking measures for loss functions, including AUC, NDCG, Precision-at-K, and mAP. Following [17], we define the loss function over two rankings $\Delta \in [0\ 1]$ as:

$$\Delta(\boldsymbol{y}, \boldsymbol{y}') = 1 - \text{score}(\boldsymbol{y}, \boldsymbol{y}'). \tag{18}$$

Here $\boldsymbol{y}'$ is the ground truth ranking and $\boldsymbol{y}$ is the prediction. We define $\mathcal{X}^+_{\boldsymbol{y}'}$ and $\mathcal{X}^-_{\boldsymbol{y}'}$ as the indexes of relevant and irrelevant neighbours respectively in the ground truth ranking $\boldsymbol{y}'$.

AUC. The area under the ROC curve is to evaluate the performance of correct ordering of data pairs, which can be computed by counting the proportion of correctly ordered data pairs:

$$\text{score}_{\text{AUC}}(\boldsymbol{y}, \boldsymbol{y}') = \frac{1}{|\,\mathcal{X}^+_{\boldsymbol{y}'}\,||\,\mathcal{X}^-_{\boldsymbol{y}'}\,|} \sum_{i \in \mathcal{X}^+_{\boldsymbol{y}'}} \sum_{j \in \mathcal{X}^-_{\boldsymbol{y}'}} \delta(i \prec_{\boldsymbol{y}} j). \tag{19}$$

$\delta(\cdot) \in \{0, 1\}$ is the indicator function. For using this AUC loss, the maximization inference in (11) can be solved efficiently by sorting the distances of data pairs, as described in [7]. Note that the loss of a wrongly ordered pair is not related to their positions in the ranking list, thus AUC is a position insensitive measure.

Precision-at-K. Precision-at-K is to evaluate the quality of top-K retrieved examples in a ranking. It is computed by counting the number of relevant data points within top-K positions and divided by K:

$$\text{score}_{\text{P@K}}(\boldsymbol{y}, \boldsymbol{y}'') = \frac{1}{K} \sum_{i=1}^{K} \delta(\boldsymbol{y}(i) \in \mathcal{X}^+_{\boldsymbol{y}''}). \tag{20}$$

Here $\boldsymbol{y}(i)$ is the example index on the i-th position of a ranking $\boldsymbol{y}$; $\delta(\cdot)$ is an indicator. An algorithm for solving the inference in (11) is proposed in [7].

NDCG. Normalized Discounted Cumulative Gain [6] is to measure the ranking quality of the first K returned neighbours. A similar measure is Precision-at-K which is the proportion of top-K relevant neighbours. NDCG is a position-sensitive measure which considers the positions of the top-K relevant neighbours. Compared to the position-insensitive measure: AUC, NDCG assigns different importances on the ranking positions, which is a more favorable measure for a general notion of a "good" ranking in real-world applications. In NDCG, each position of the ranking is assigned a score in a decreasing way. NDCG can be computed by accumulating the scores of top-K relevant neighbours:

$$\text{score}_{\text{NDCG}}(\boldsymbol{y}, \boldsymbol{y}') = \frac{1}{\sum_{i=1}^{K} S(i)} \sum_{i=1}^{K} S(i) \delta(\boldsymbol{y}(i) \in \mathcal{X}^+_{\boldsymbol{y}'}). \tag{21}$$

Here $\boldsymbol{y}(i)$ is the example index on the i-th position of a ranking $\boldsymbol{y}$. $S(i)$ is the score assigned to the i-th position of a ranking. $S(1) = 1$, $S(i) = 0$ for $i > k$ and

Table 1. Results using NDCG measure (64 bits). We compare our StructHash using AUC (StructH-A) and NDCG (StructH-N) loss functions with other supervised and un-supervised methods. Our method using NDCG loss performs the best in most cases.

Dataset	StructH-N	StructH-A	CGH	SPLH	STHs	BREs	ITQ	SPHER	MDSH	AGH	LSH
	NDCG ($K = 100$)										
STL10	**0.435**	0.374	0.375	0.404	0.214	0.289	0.337	0.318	0.313	0.310	0.228
USPS	**0.905**	0.893	0.900	0.816	0.688	0.777	0.804	0.762	0.735	0.741	0.668
MNIST	0.851	0.798	**0.867**	0.804	0.594	0.805	0.856	0.806	0.100	0.793	0.561
CIFAR	0.335	0.259	0.258	**0.357**	0.178	0.273	0.314	0.297	0.283	0.286	0.168
ISOLET	**0.881**	0.839	0.866	0.629	0.766	0.483	0.623	0.518	0.538	0.536	0.404

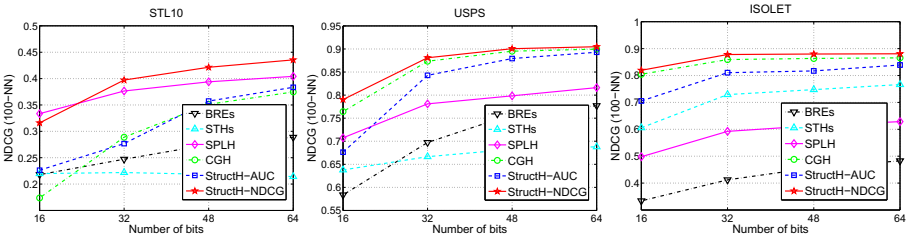

Fig. 1. NDCG results on 3 datasets. Our StructHash performs the best.

$S(i) = 1/\log_2(i)$ for other cases. A dynamic programming algorithm is proposed in [1] for solving the maximization inference in (11).

mAP. Mean average precision (mAP) is the averaged precision-at-K scores over all positions of relevant data points in a ranking, which is computed as:

$$\text{score}_{\text{mAP}}(\boldsymbol{y}, \boldsymbol{y}') = \frac{1}{|\mathcal{X}^+_{\boldsymbol{y}'}|} \sum_{i=1}^{|\mathcal{X}^+_{\boldsymbol{y}'}|+|\mathcal{X}^-_{\boldsymbol{y}'}|} \delta(\boldsymbol{y}(i) \in \mathcal{X}^+_{\boldsymbol{y}'}) \text{score}_{\text{P@K}(K=i)}(\boldsymbol{y}, \boldsymbol{y}'). \quad (22)$$

For using this mAP loss, an efficient algorithm for solving the inference in (11) is proposed in [25].

5 Experiments

Our method is in the category of supervised method for learning compact binary codes. Thus we mainly compare with 4 supervised methods: column generation hashing (CGH) [12],supervised binary reconstructive embeddings (BREs) [10], supervised self-taught hashing (STHs) [26], semi-supervised sequential projection learning hashing (SPLH) [22].

For comparison, we also run some unsupervised methods: locality-sensitive hashing (LSH) [3], anchor graph hashing (AGH) [16], spherical hashing (SPHER) [5], multi-dimension spectral hashing (MDSH) [23], and iterative quantization (ITQ) [4]. We carefully follow the original authors' instruction for parameter setting. For SPLH, the regularization parameter is picked from 0.01 to 1. We use the hierarchical variant of AGH. The bandwidth parameters of Gaussian affinity

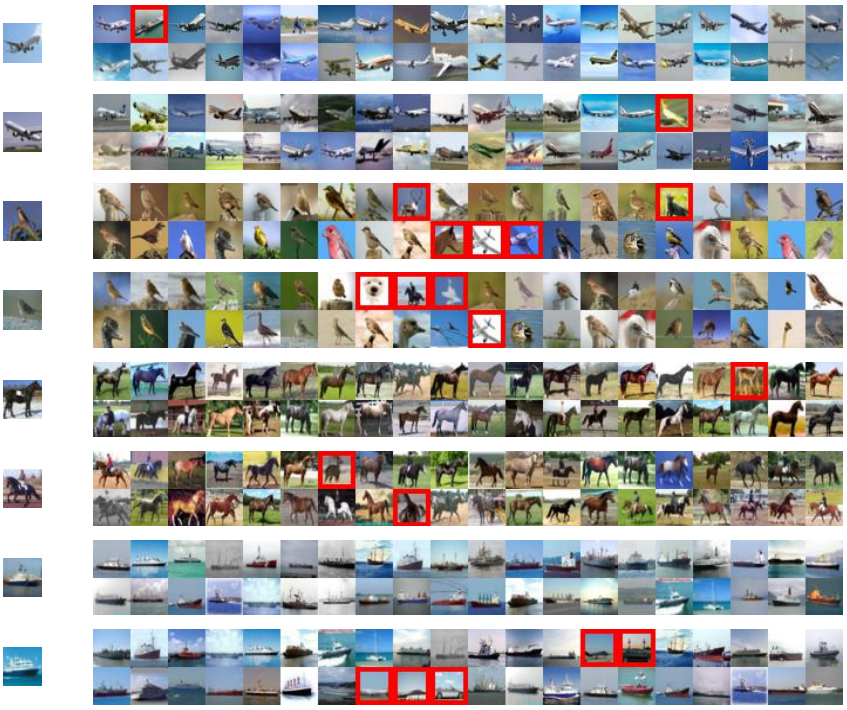

Fig. 2. Some ranking examples of our method. The first column shows query images, and the rest are retrieved images. False predictions are marked by red boxes.

in MDSH is set as $\sigma = t\bar{d}$. Here $\bar{d}$ is the average Euclidean distance of top 100 nearest neighbours and t is picked from 0.01 to 50. For supervised training of our StructHash and CGH, we use 50 relevant and 50 irrelevant examples to construct similarity information for each data point.

We use 9 datasets for evaluation, including one UCI dataset: ISOLET, 4 image datasets: CIFAR10[3], STL10[4], MNIST, USPS, and another 4 large image datasets: Tiny-580K [4], Flickr-1M[5], SIFT-1M [22] and GIST-1M[6]. CIFAR10 is a subset of the 80-million tiny images and STL10 is a subset of Image-Net. Tiny-580K consists of 580, 000 tiny images. Flick-1M dataset consists of 1 million thumbnail images. SIFT-1M and GIST-1M datasets contain 1 million SIFT and GIST features respectively.

We follow a common setting in many supervised methods [10,15,12] for hashing evaluation. For multi-class datasets, we use class labels to define the relevant and irrelevant semantic neighbours by label agreement. For large datasets: Flickr-1M, SIFT-1M, GIST-1M and Tiny-580K, the semantic ground truth is

[3] http://www.cs.toronto.edu/~kriz/cifar.html
[4] http://www.stanford.edu/~acoates/stl10/
[5] http://press.liacs.nl/mirflickr/
[6] http://corpus-texmex.irisa.fr/

Table 2. Results using ranking measures of Precision-at-K, Mean Average Precision and Precision-Recall (64 bits). We compare our method using AUC (StructH-A) and NDCG (StructH-N) loss functions with other supervised and un-supervised methods. Our method using NDCG loss performs the best on these measures.

Dataset	StructH-N	StructH-A	CGH	SPLH	STHs	BREs	ITQ	SPHER	MDSH	AGH	LSH
	Precision-at-K ($K = 100$)										
STL10	**0.431**	0.376	0.376	0.396	0.208	0.279	0.325	0.303	0.298	0.301	0.222
USPS	**0.903**	0.894	0.898	0.805	0.667	0.755	0.780	0.730	0.698	0.711	0.637
MNIST	0.849	0.807	**0.862**	0.797	0.579	0.790	0.842	0.788	0.100	0.780	0.540
CIFAR	0.336	0.259	0.261	**0.354**	0.174	0.264	0.301	0.286	0.270	0.281	0.164
ISOLET	**0.875**	0.844	0.859	0.604	0.755	0.448	0.589	0.477	0.493	0.493	0.370
	Mean Average Precision (mAP)										
STL10	**0.331**	0.326	0.322	0.299	0.155	0.211	0.233	0.193	0.178	0.162	0.162
USPS	**0.868**	0.851	0.848	0.689	0.456	0.582	0.566	0.451	0.405	0.333	0.418
MNIST	**0.802**	0.790	0.789	0.684	0.397	0.558	0.585	0.510	0.119	0.505	0.343
CIFAR	0.294	**0.300**	0.298	0.289	0.147	0.204	0.215	0.204	0.181	0.201	0.149
ISOLET	**0.836**	0.796	0.815	0.518	0.653	0.340	0.484	0.357	0.348	0.298	0.267
	Precision-Recall										
STL10	**0.267**	0.248	0.248	0.246	0.130	0.181	0.200	0.174	0.164	0.145	0.138
USPS	**0.776**	0.760	0.760	0.609	0.401	0.520	0.508	0.424	0.379	0.326	0.375
MNIST	**0.591**	0.574	0.582	0.445	0.165	0.313	0.323	0.246	0.018	0.197	0.143
CIFAR	0.105	0.093	0.091	**0.110**	0.042	0.066	0.074	0.069	0.064	0.061	0.042
ISOLET	**0.759**	0.709	0.737	0.445	0.563	0.301	0.429	0.321	0.320	0.275	0.238

defined according to the ℓ_2 distance [22]. Specifically, a data point is labeled as a relevant data point of the query if it lies in the top 2 percentile points in the whole dataset. We generated GIST features for all image datasets except MNIST and USPS. we randomly select 2000 examples for testing queries, and the rest is used as database. We sample 2000 examples from the database as training data for learning models. For large datasets, we use 5000 examples for training. To evaluate the performance of compact bits, the maximum bit length is set to 64, as similar to the evaluation settings in other supervised hashing methods [10,12].

We report the result of the NDCG measure in Table 1. We compare our StructHash using AUC and NDCG loss functions with other supervised and unsupervised methods. Our method using NDCG loss function performs the best in most cases. We also report the result of other common measures in Table 2, including the result of Precision-at-K, Mean Average Precision (mAP) and Precision-Recall. Precision-at-K is the proportion of true relevant data points in the returned top-K results. The Precision-Recall curve measures the overall performance in all positions of the prediction ranking, which is computed by varying the number of nearest neighbours. It shows that our method generally performs better than other methods on these evaluation measures. As described before, compared to the AUC measure which is position insensitive, the NDCG measure assigns different importance on ranking positions, which is closely related to many other position sensitive ranking measures (e.g., mAP). As expected, the result shows that on the Precision-at-K, mAP and Precision-recall measures, optimizing the position sensitive NDCG loss performs better than the AUC loss. The triplet loss based method CGH actually helps to reduce the AUC loss. This may explain the reason that our method, which aims to reduce the NDCG loss,

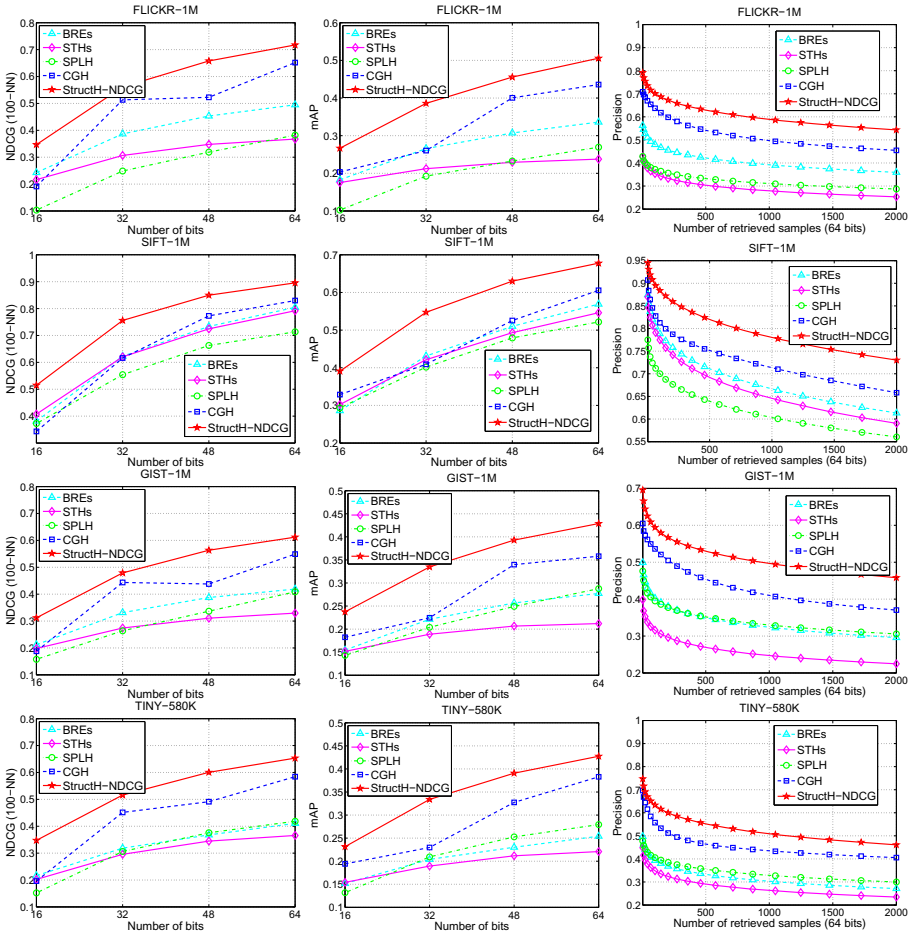

Fig. 3. Results on 4 large datasests: Flickr-1M (1 million Flickr images), Sift-1M (1 million SIFT features), Gist-1M (1 million GIST features) and Tiny580K (580, 000 Tiny image dataset). We compare with several supervised methods. The results of 3 measures (NDCG, mAP and precision of top-K neighbours) are show here. Our StructHash outperforms others in most cases.

is able to outperform CGH in these measures. We also plot the NDCG results on several datasets in Fig. 1 by varying the number of bits. Some retrieval examples are shown in Fig. 2.

We further evaluate our method on 4 large-scale datasets (Flickr-1M, SIFT-1M, GIST-1M and Tiny-580K). The results of NDCG, mAP and the precision of top-K neighbours are shown in Fig. 3. The NDCG and mAP results are shown by varying the number of bits. The precision of top-K neighbours is shown by varying the number of retrieved examples. In most cases, our method outperforms

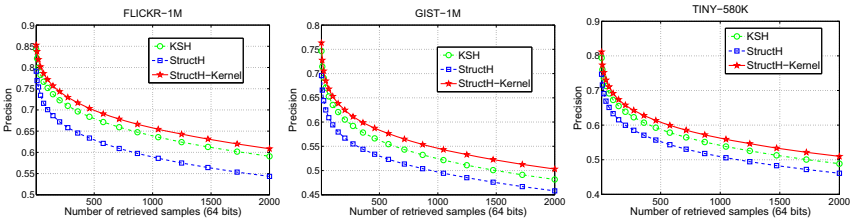

Fig. 4. Comparison on large datasets of our kernel StructHash (StructHash-Kernel) with our non-kernel StructHash and the relevant method KSH [15]. Our kernel version is able to achieve better results.

other competitors. Our method with NDCG loss function succeeds to achieve good performance both on NDCG and other measures.

Applying the kernel technique in KLSH [11] and KSH [15] further improves the performance of our method. As describe in [15], we perform a pre-processing step to generate the kernel mapping features: we randomly select a number of support vectors (300) then compute the kernel response on data points as input features. Note that here we simply follow KSH for the kernel parameter setting. We evaluate this kernel version of our method in Fig. 4 and compare to KSH. Our kernel version is able to achieve better results.

6 Conclusion and Future Work

We have developed a hashing framework that allows us to directly optimize multivariate performance measures. The fact that the proposed method outperforms comparable hashing approaches is to be expected, as it more directly optimizes the required loss function. It is anticipated that the success of the approach may lead to a range of new hashing-based applications with task-specified targets. Extension to make use of the power of more sophisticated hash functions such as kernel functions or decision trees is of future work.

References

1. Chakrabarti, S., Khanna, R., Sawant, U., Bhattacharyya, C.: Structured learning for non-smooth ranking losses. In: Proc. ACM Knowledge Discovery & Data Mining (2008)
2. Demiriz, A., Bennett, K.P., Shawe-Taylor, J.: Linear programming boosting via column generation. Mach. Learn. (2002)
3. Gionis, A., Indyk, P., Motwani, R.: Similarity search in high dimensions via hashing. In: Proc. Int. Conf. Very Large Data Bases (1999)
4. Gong, Y., Lazebnik, S., Gordo, A., Perronnin, F.: Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. IEEE Trans. Patt. Anal. & Mach. Intelli. (2012)
5. Heo, J., Lee, Y., He, J., Chang, S., Yoon, S.: Spherical hashing. In: Proc. Int. Conf. Comp. Vis. & Patt. Recogn. (2012)

6. Järvelin, K., Kekäläinen, J.: IR evaluation methods for retrieving highly relevant documents. In: Proc. ACM Conf. SIGIR (2000)
7. Joachims, T.: A support vector method for multivariate performance measures. In: Proc. Int. Conf. Mach. Learn. (2005)
8. Joachims, T.: Training linear SVMs in linear time. In: Proc. ACM Knowledge Discovery & Data Mining (2006)
9. Kelley Jr., J.E.: The cutting-plane method for solving convex programs. J. Society for Industrial & Applied Math. (1960)
10. Kulis, B., Darrell, T.: Learning to hash with binary reconstructive embeddings. Proc. Adv. Neural Info. Process. Syst. (2009)
11. Kulis, B., Grauman, K.: Kernelized locality-sensitive hashing. IEEE Trans. Patt. Anal. & Mach. Intelli. (2012)
12. Li, X., Lin, G., Shen, C., van den Hengel, A., Dick, A.: Learning hash functions using column generation. In: Proc. Int. Conf. Mach. Learn. (2013)
13. Lin, G., Shen, C., Shi, Q., van den Hengel, A., Suter, D.: Fast supervised hashing with decision trees for high-dimensional data. In: Proc. Int. Conf. Comp. Vis. & Patt. Recogn. Columbus, Ohio, USA (2014), `https://bitbucket.org/chhshen/fasthash/src`
14. Lin, G., Shen, C., Suter, D., van den Hengel, A.: A general two-step approach to learning-based hashing. In: Proc. Int. Conf. Comp. Vis., Sydney, Australia (2013)
15. Liu, W., Wang, J., Ji, R., Jiang, Y., Chang, S.: Supervised hashing with kernels. In: Proc. Int. Conf. Comp. Vis. & Patt. Recogn. (2012)
16. Liu, W., Wang, J., Kumar, S., Chang, S.F.: Hashing with graphs. In: Proc. Int. Conf. Mach. Learn. (2011)
17. McFee, B., Lanckriet, G.: Metric learning to rank. In: Proc. Int. Conf. Mach. Learn. (2010)
18. Shen, C., Li, H.: On the dual formulation of boosting algorithms. IEEE Trans. Patt. Anal. & Mach. Intelli. (2010)
19. Shen, C., Lin, G., van den Hengel, A.: StructBoost: Boosting methods for predicting structured output variables. IEEE Trans. Patt. Anal. & Mach. Intelli. (2014)
20. Shen, F., Shen, C., Shi, Q., van den Hengel, A., Tang, Z.: Inductive hashing on manifolds. In: Proc. Int. Conf. Comp. Vis. & Patt. Recogn., Oregon, USA (2013)
21. Tsochantaridis, I., Hofmann, T., Joachims, T., Altun, Y.: Support vector machine learning for interdependent and structured output spaces. In: Proc. Int. Conf. Mach. Learn. (2004)
22. Wang, J., Kumar, S., Chang, S.: Semi-supervised hashing for large scale search. IEEE Trans. Patt. Anal. & Mach. Intelli. (2012)
23. Weiss, Y., Fergus, R., Torralba, A.: Multidimensional spectral hashing. In: Proc. Eur. Conf. Comp. Vis. (2012)
24. Weiss, Y., Torralba, A., Fergus, R.: Spectral hashing. In: Proc. Adv. Neural Info. Process. Syst. (2008)
25. Yue, Y., Finley, T., Radlinski, F., Joachims, T.: A support vector method for optimizing average precision. In: Proc. ACM Conf. SIGIR (2007)
26. Zhang, D., Wang, J., Cai, D., Lu, J.: Extensions to self-taught hashing: Kernelisation and supervision. In: Proc. ACM Conf. SIGIR Workshop (2010)
27. Zhu, C., Byrd, R.H., Lu, P., Nocedal, J.: Algorithm 778: L-BFGS-B: Fortran subroutines for large-scale bound-constrained optimization. ACM T. Math. Softw. (1997)