A Unified Approach to Learning Task-Specific Bit Vector Representations for Fast Nearest Neighbor Search

Vinod Nair
Yahoo! Labs Bangalore
vnair@yahoo-inc.com

Dhruv Mahajan
Yahoo! Labs Bangalore
dkm@yahoo-inc.com

S. Sundararajan
Yahoo! Labs Bangalore
ssrajan@yahoo-inc.com

ABSTRACT

Fast nearest neighbor search is necessary for a variety of large scale web applications such as information retrieval, nearest neighbor classification and nearest neighbor regression. Recently a number of machine learning algorithms have been proposed for representing the data to be searched as (short) bit vectors and then using hashing to do rapid search. These algorithms have been limited in their applicability in that they are suited for only one type of task – e.g. Spectral Hashing learns bit vector representations for retrieval, but not say, classification. In this paper we present a unified approach to learning bit vector representations for many applications that use nearest neighbor search. The main contribution is a single learning algorithm that can be customized to learn a bit vector representation suited for the task at hand. This broadens the usefulness of bit vector representations to tasks beyond just conventional retrieval.

We propose a learning-to-rank formulation to learn the bit vector representation of the data. LambdaRank algorithm is used for learning a function that computes a task-specific bit vector from an input data vector. Our approach outperforms state-of-the-art nearest neighbor methods on a number of real world text and image classification and retrieval datasets. It is scalable and learns a 32-bit representation on 1.46 million training cases in two days.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval; I.2.6 [**Artificial Intelligence**]: Learning

Keywords

nearest neighbor search, hashing, learning to rank

1. INTRODUCTION

Nearest neighbor (NN) methods are widely used in several web applications such as web objects (e.g., documents, and images) classification, and retrieval. In retrieval applications, availability of large volume of web data makes it possible to retrieve very *similar* objects for almost any given query object, and these similar objects are *nearest neighbors* under some suitably defined similarity measure. Similarly, NN methods such as k-Nearest Neighbor (kNN) classifiers achieve high accuracy as very similar objects often belong to the same category. These classifiers are attractive due to their simplicity and speed independence in the number of classes[1].

Since NN methods work well in very large data set scenarios, the ability to handle large volume of data during training and testing is a necessity. For example, in a naive implementation, linear search over the entire training data is needed, and this is expensive for web scale applications. Therefore, development of algorithms, data structures (e.g., KD-Trees) and representations (e.g., bit vectors) that facilitate fast search are of paramount importance. Furthermore, task-specific performance (e.g., accuracy in classification tasks, and ranking measures such as normalized discounted cumulative gain (NDCG), precision@k in retrieval tasks) achieved is dependent on distance metric or similarity measure that is used in finding the neighbors. Although Euclidean distance is a useful metric in some applications, it is often not optimal. Hence, learning a task-specific distance metric is also necessary.

Keeping above requirements in mind, we propose a unified framework in which task-specific fast NN methods that achieve high performance can be developed. We demonstrate our method on classification and retrieval tasks, and experimental results show superior performance over state-of-the-art NN classification and retrieval methods.

1.1 Nearest Neighbor Approaches

In recent years, there has been a flurry of research activity on nearest neighbor methods for web classification and information retrieval applications. To motivate our approach, we briefly discuss several popular methods and differentiate them along three key aspects (capturing the above mentioned requirements): 1) speed (training and testing), 2) learning (representations/similarity measure with or without additional information (e.g., labels of objects, pairwise similar or dissimilar objects), and 3) optimizing a task-specific performance measure during learning. In Table 1 we present a summary of various methods, and Figure 1 shows different configurations to which these methods adhere to. We observe that all the existing methods fall short in covering at least one of the aspects, and we attempt to address this issue.

Hashing methods are approximate nearest neighbor search methods, and are popular due to their speed. In these methods data points (objects) are represented as bit vectors, and such a representation helps in finding the neighbors quickly[2]. Similarity between two objects is measured by the Hamming distance, and nearest

---

[1] Although standard classifiers such as support vector machines or logistic regression models have been quite successfully used in the web context, the dependence of model and test time complexities on the number of classes makes them less attractive when the number of classes is very large.

[2] An important advantage of using a bit vector representation is that the search time can be made *constant with respect to the size of the search set*. For example, with 32 bit representation, one can build an inverted index with $2^{32} = 4294967296$ bins
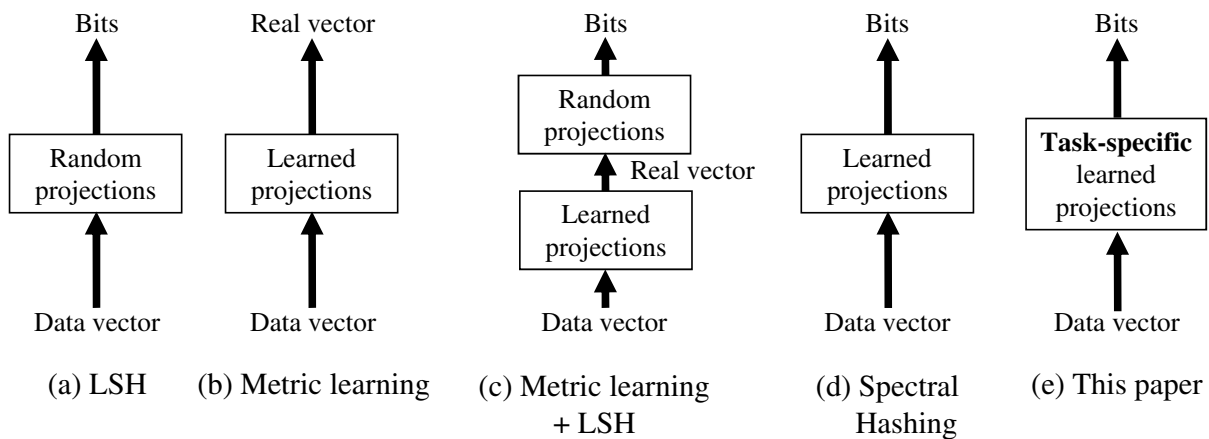
Figure 1: Comparison of various approaches with respect to our algorithm.

neighbors are identified as the points that are within a user specified Hamming distance from a given query.

Locality Sensitive Hashing (LSH) [1] is a simple method (figure 1(a)) in which bit vector representation for a data point (object) is obtained from projecting the data vector on several random directions, and converting the projected values to $\{0, 1\}$ by thresholding. This method does not make use of data to *learn* the representation. Learning is important because it is useful to find compact and better representations, which in turn results in improved speed and task-specific performance. Hence, attempts have been made to develop better hashing methods. Sophisticated hashing methods such as Semantic Hashing [19] and Spectral Hashing [23] (figure 1(d)) learn compact bit vector representations (codes) with the desirable property that similar objects have similar codes. In Semantic Hashing, each object in the training database is represented by a compact binary code, and the code is computed using a feed forward neural network, with the network weights learned by minimizing data reconstruction error. In Spectral Hashing [23], k-bit compact binary codes are obtained by minimizing the sum of weighted Hamming distance between data points that are similar, and the weight represents a similarity score computed using the euclidean distance in a suitable input feature space. A key disadvantage of these methods is that they do not make use of any task-specific information such as object labels, query-document pair relevancy score, etc. (when available); and, they do not explicitly optimize task-specific performance measures. So these methods suffer on actual task-specific performance such as classification accuracy, NDCG, etc.

There are several popular methods that learn a distance metric (e.g., Mahalanobis distance). These methods make use of additional information and learn the distance metric by a optimizing task-specific objective function. Methods such as Neighborhood component analysis (NCA) [9] and large margin nearest neighbor (LMNN) [22] (figure 1(b)) classification learn the metric using class label information, keeping nearest neighbor classification as the goal. Metric learning has also been seen as a special case of ranking problem, from the view point that *good neighbors appear at the top of the list and bad neighbors appear at the bottom.* McFee et al. [16] propose a structural SVM learning framework (figure 1(b)) for learning the metric where various ranking measures such as precision@k, NDCG, etc., can be optimized (referred as MLR in Table 1). Chechik et al. [6] propose an online algorithm

for scalable image similarity (OASIS) that learns a similarity measure (figure 1(b)) with the property of scoring the more relevant pair of objects higher; it uses labeled information in the form of *relative similarity* of different pairs of objects. A major drawback with these methods is that they do not provide bit vector representations. Therefore, finding nearest neighbors is expensive unless the input representation is sparse (which is not the case in many applications). Also, learning is computationally expensive with NCA, LMNN and MLR approaches[3]. Therefore, they are not suitable for very large-scale problems.

Jain et al. [12] propose a method (figure 1(c)) that applies LSH on a learned metric (referred as M+LSH in Table 1). However, this method does not use task-specific objective function for learning the metric; more importantly, it does not learn the bit vector representation *directly*. Although LSH can be applied on the projected data using a metric learned via NCA or LMNN, any such *independent* two stage method will be *sub-optimal* in getting a good bit vector representation.

1.2 Our Contributions

We propose a unified framework (Section 3) to develop fast nearest neighbor methods for various applications such as classification, retrieval, regression, and user recommendations. In our method we *learn* the bit vector representation directly by *explicitly* optimizing task-specific performance. This is done by solving a learning-to-rank problem. Our approach has several advantages.

- Using bit vector representation helps in finding nearest neighbors fast using Hamming distance as the measure.

- High performance is achieved by *direct* bit vector learning and *explicit* optimization of task-specific performance measure.

- It is easy to adapt the method for different applications via choices of performance measure and ranking. LambdaRank [5] can be used to optimize various non-smooth task-specific performance measures (e.g., precision@k and nearest neighbor classification accuracy) (Section 3).

---

where each bin contains the data points that were mapped to the corresponding bit vector.

[3]In LMNN and MLR, computationally complex matrix constraints such as positive definiteness (PD) are used. Chechik et al. [6] argue that PD constraints are not necessary for large datasets, and they do not use such constraints. In NCA, the computational cost is quadratic in the number of training points; there are no positive definite constraints since the projection matrix is learned directly.
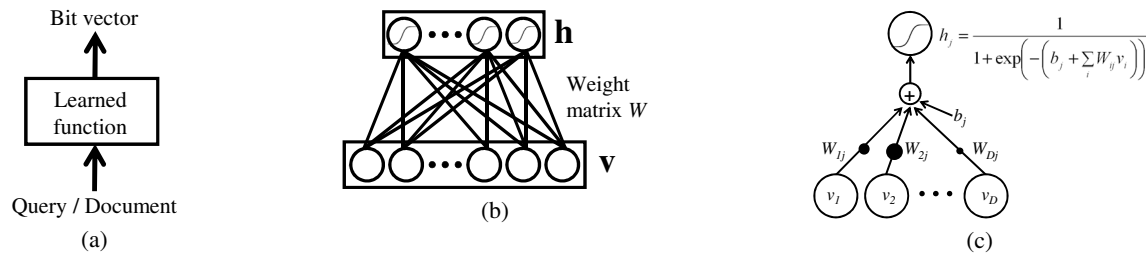
Figure 2: (a) The algorithm we are proposing learns a function that takes a query or a document as input and computes a bit vector as output. (b) The function multiplies the input vector $\mathbf{v}$ by the matrix W, adds the biases b_j to the result, and then applies the *sigmoid function* to compute a vector $\mathbf{h}$ whose components are real values between 0 and 1. The elements of W and the biases b_j are the learnable parameters of the function. (c) The sigmoid is applied component-wise to the result of the linear projection.

Method	Learning	Representation	Task-Specific
LSH Fig. 1(a)	No	bit vector	No
Spectral Fig. 1(d)	Yes	bit vector	No
Semantic Fig. 1(d)	Yes	bit vector	No
NCA Fig. 1(b)	Yes	real vector	Yes (C)
LMNN Fig. 1(b)	Yes	real vector	Yes (C)
OASIS Fig. 1(b)	Yes	real vector	Yes (IR)
MLR Fig. 1(b)	Yes	real vector	Yes (IR)
M-LSH Fig. 1(c)	Yes	bit vector	No
Ours Fig. 1(e)	Yes	bit vector	Yes (C, IR, R)

Table 1: Comparison of Methods (see text for details). The abbreviations C, IR and R stand for classification, information retrieval and regression tasks respectively, and PD stands for positive definite matrix constraint. Compact bit vector representation is preferable for faster nearest neighbor search during training and testing. Constraints such as positive definite affect training speed. Task-specific performance optimization is important to get improved performance. Jain et al. [12] use information theoretic criterion for distance metric learning. However, LSH can be combined with any other task-specific distance metric learning methods such as NCA, LMNN, etc.

- From an optimization view point, our formulation and implementation (Section 4) are simple and scalable. We observed that our algorithm, parallelized over 8 cores, completes learning in 2 days on a 1.46 million image dataset. At test time on the same dataset, 100000 query searches over 1.46 million vectors with a 32-bit representation using an inverted index are completed in 1.37 seconds (13.7 microseconds per query on average) on a single core of an Intel Xeon 2.33GHz machine with 4GB RAM.

We conduct detailed experimental study (Section 5) on several benchmark datasets for two applications: 1) classification and 2) information retrieval. Comparisons with state-of-the-art NN methods show that our method performs significantly better. We discuss applicability of our framework to other problems such as hierarchical classification and regression (Section 6). We also show how the speed of our method can be significantly improved *further* with a multi-stage (cascaded) implementation; here, speed improvement is achieved by allocating fewer bits in earlier stages, and reducing the nearest neighbor search space progressively (Section 6).

2. OVERVIEW

Our algorithm learns a function that takes a query or document vector as input and computes its corresponding bit vector representation as output (see figure 2a). Here we describe at a high level how the function is learned, and after learning, how it is used for nearest neighbor search.

Training data: As mentioned before, we adopt a learning-to-rank approach. So each training example has the form (*query, document list*), where *document list* specifies how its elements should be ordered with respect to that *query*. In this paper we assume that both the query and the document are the same type of data, using the same representation (e.g., both are images of the same size, or both are text documents that use the same bag-of-words representation)[4]. Therefore the same learned function can be applied to both a query and a document to compute the corresponding bit vector.

Parameterization of the function: The function applies linear projections to the input vector, followed by the *sigmoid function* to map the output of the linear projections into real numbers in the range $[0, 1]$ (see figure 2b and 2c). Making the function differentiable allows for gradient descent learning. At test time, when a proper bit vector is required for fast nearest neighbor search, the output of the sigmoid function is thresholded at 0.5 to convert its value from a real number in $[0, 1]$ into 0 or 1. The linear projection coefficients are the learnable parameters of the function. They are represented as a matrix of size (number of inputs) × (number of bits). See section 3 for more details.

Learning: LambdaRank [5] is the algorithm we use for learning. It is based on the RankNet algorithm [4], which learns a pairwise ranking function. Given a triplet (*query, document1, document2*), the ranking function computes the probability that *document 1* should be ranked higher than *document 2* for that *query*. The parameters of the function are learned iteratively using the gradient of the log probability of the true pairwise ranking for a set of training triplets.

While RankNet learning is limited to pairwise ranking, LambdaRank can optimize more general, *listwise* ranking evaluation scores. Suppose LambdaRank is optimizing a ranking function for some

---

[4]It is easy to modify our algorithm to handle the case where the query and the document are two distinct types of inputs, but we do not consider it here.

evaluation score S (e.g. NDCG). Given a query, the basic version of LambdaRank defines the gradient to be that function's RankNet gradient for a pair of documents, multiplied by the change in the score's value $|\Delta S|$ if those two documents swapped positions in the full listwise ordering computed by the current ranking function. Optimizing with the LambdaRank gradient has been shown empirically to converge to a local optimum of various non-smooth IR evaluation scores [7].

Adapting LambdaRank to learn a bit vector representation requires several modifications. These are 1) the parameterization of the ranking function, 2) the procedure for finding pairwise swaps that give nonzero $|\Delta S|$, and 3) the definition of appropriate evaluation scores for various tasks. In addition, we show how to apply LambdaRank to non-retrieval tasks such as classification and regression with bit vectors. Details are in section 3.

2.1 Nearest neighbor search with bit vectors

After learning, the bit vector representation is used to find nearest neighbors for a query vector within a set of documents. The query is first converted into a bit vector using the learned function, with the output of the sigmoid function thresholded at 0.5. The bit vectors for the documents to be searched can be precomputed and stored in memory. The query bit vector is then compared against the document bit vectors in the search set with Hamming distance as the metric. Documents with the lowest Hamming distances are returned as neighbors.

Two important choices that need to be made in bit vector-based nearest neighbor search are 1) the type of thresholding to use on the Hamming distance, and 2) how to resolve ties in the ranking. These choices are task-specific, so we discuss them in section 3.

3. PROBLEM FORMULATION

Now we explain how the training data for different tasks are constructed to fit the learning-to-rank framework. Then we define the function for computing the bit vector representation, how it is used for ranking, and how it is learned from data using the RankNet and LambdaRank algorithms.

3.1 Training data for different tasks

Learning requires triplets, each consisting of a query and two documents, along with the desired pairwise ranking. Depending on the task (e.g. retrieval, nearest neighbor classification, nearest neighbor regression, etc.), how these triplets are created differs.

NN Classification: Given a classification dataset consisting of (input, label) pairs, we define triplets as follows: the input for which the class label needs to be predicted is the query. Inputs in the training set that are used as candidate neighbors are the documents. Given a query belonging to a particular class, we want documents of the same class to get ranked higher than documents of all other classes. We will denote input vectors by $\mathbf{v}$ and their labels by l. So three class-labeled inputs $(\mathbf{v}_1, l_1)$, $(\mathbf{v}_2, l_2)$, and $(\mathbf{v}_3, l_3)$, where $l_1 = l_2$ and $l_1 \neq l_3$ are turned into a triplet $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ where $\mathbf{v}_1$ is the query, $\mathbf{v}_2$ and $\mathbf{v}_3$ form the document pair.

Retrieval: A retrieval dataset may already be in the desired query-document format, where each query has a corresponding list of documents ordered by a relevance label. In such a case, it is clear how to construct triplets. In image retrieval applications (such as the ones used in our experiments) we may simply be given for each query a set of documents that are equally relevant (i.e. the relevance label is either 0 or 1). Here we can follow the same strategy as in the classification setting and create triplets of the form $(\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3)$ where $\mathbf{v}_1$ is the query, $\mathbf{v}_2$ is a relevant document for the query, and $\mathbf{v}_3$ is an irrelevant document.

3.2 Computing the bit vector representation

Let $\mathbf{v} \in \Re^D$ be the D-dimensional input vector (either a query or a document) for which we want to compute the bit vector. First we define a function for computing a real-valued B-dimensional vector $\mathbf{h}$ whose components have values between 0 and 1, $\mathbf{h} \in [0, 1]^B$, from $\mathbf{v}$. The bit vector is subsequently computed from $\mathbf{h}$ by rounding. The jth component of $\mathbf{h}$ is given by

$$h_j = \frac{1}{1 + \exp\left(-\left(b_j + \sum_{i=1}^{D} W_{ij} v_i\right)\right)}, \qquad (1)$$

where v_i is the ith component of $\mathbf{h}$, W_{ij} is a learnable *weight* parameter, b_j is a learnable bias, and $1/(1 + \exp(-(\cdot)))$ is the sigmoid function. The function for computing the entire $\mathbf{h}$ vector is parameterized in terms of a $D \times B$ matrix W and the B bias parameters (figure 2b). To simplify notation we omit bias terms from the equations in the rest of the paper without loss of generality[5] The bit vector $\mathbf{b}$ is computed by rounding each component of $\mathbf{h}$ to 0 or 1. Note that $\mathbf{h}$ is differentiable with respect to W, but $\mathbf{b}$ is not because of the non-smoothness of rounding.

3.3 Smooth ranking function for learning

We want to formulate an algorithm for learning W within a learning-to-rank framework. To do this we first need to define a ranking function suited for learning a bit vector representation. It takes a query and a document as inputs and computes a scalar score with which the document can be ranked. Given a query $\mathbf{v}_q$ and a set of documents $\{\mathbf{v}_{d_1}, \mathbf{v}_{d_2}, ..., \mathbf{v}_{d_n}\}$ (all in $\Re^D$), the score can be computed for each query-document pair, followed by a sort, to rank the documents with respect to the query.

Let $\mathbf{b}_q \in \{0, 1\}^B$ be the bit vector for the query and $\mathbf{b}_i \in \{0, 1\}^B$ be the bit vector for the ith document. The ranking score s_i for the ith document is the Hamming distance between $\mathbf{b}_q$ and $\mathbf{b}_i$. Documents are ranked in ascending order of the score.

For gradient-based learning we need the ranking score to be differentiable with respect to W. The score s_i does not meet this requirement because rounding is not a differentiable function. So we compute an alternative score $\hat{s}_i$ using $\mathbf{h}$ (from equation 1) instead. Since $\mathbf{h}$ is real-valued, Hamming distance is no longer appropriate. One alternative is to use Euclidean distance, i.e. let the score for the ith document be the Euclidean distance between $\mathbf{h}_q$ and $\mathbf{h}_i$. When the components of $\mathbf{h}_q$ and $\mathbf{h}_i$ are exactly binary, the score will be the same as Hamming distance. One drawback is that the score can also be zero even when $\mathbf{h}_q$ and $\mathbf{h}_i$ are not binary, but still identical. For example, if the components of both vectors are all 0.5, then the score will be zero.

A second alternative is the following:

$$\hat{s}_i = \sum_{j=1}^{N} h_{qj}(1 - h_{ij}) + (1 - h_{qj})h_{ij}, \qquad (2)$$

where h_{qj} and h_{ij} are the jth component of $\mathbf{h}_q$ and $\mathbf{h}_i$, respectively. It 'relaxes' Hamming distance to non-binary values, and becomes equal to Hamming distance for binary values. It is *not* a proper distance metric – two identical vectors will not give a score of zero unless they are binary. We use this relaxed Hamming score in our experiments.

---

[5]The biases can be put into W by appending an extra constant element 1 to the input vector and defining W to be a $(D+1) \times B$ matrix where the extra row corresponds to the biases.

3.4 Learning with RankNet

As mentioned before, LambdaRank is based on RankNet. Now we describe RankNet in some detail because it is required for LambdaRank. Also it serves as an important baseline in our experiments to demonstrate the usefulness of task-specific optimization.

So far we have formulated a ranking function that incorporates the smooth version of the bit vector representation and is differentiable with respect to W. Now we plug this ranking function into the RankNet algorithm to learn W.

Given a query $\mathbf{v}_q$ and a pair of documents, $\mathbf{v}_{d_1}$ and $\mathbf{v}_{d_2}$, let $\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2}$ denote the event that $\mathbf{v}_{d_1}$ is ranked higher than $\mathbf{v}_{d_2}$. RankNet adopts a probabilistic view that allows for uncertainty in the pairwise ranking. The ranking scores of the two documents do not deterministically imply an ordering. Instead they define a *probability* that $\mathbf{v}_{d_1}$ is ranked higher than $\mathbf{v}_{d_2}$:

$$P(\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2}) = \frac{1}{1 + \exp(\hat{s}_1 - \hat{s}_2)}, \qquad (3)$$

where $\hat{s}_1$ and $\hat{s}_2$ are the ranking scores for $\mathbf{v}_{d_1}$ and $\mathbf{v}_{d_2}$, respectively (as computed by equations 1 and 2).

Note that if $\hat{s}_1 < \hat{s}_2$, then $P(\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2}) > 0.5$. Intuitively if one document is closer to the query than the other, it has a higher probability of being ranked higher than the farther one. And if $\hat{s}_1 << \hat{s}_2$ (document 1 is much closer to the query than document 2), then $P(\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2}) \approx 1$.

Consider the simple case where our training set consists of only one triplet $(\mathbf{v}_q, \mathbf{v}_{d_1}, \mathbf{v}_{d_2})$. Let $T_{1 \triangleright 2}$ be the target probability that $\mathbf{v}_{d_1}$ is ranked higher than $\mathbf{v}_{d_2}$ with respect to $\mathbf{v}_q$. For example, $T_{1 \triangleright 2} = 1$ if $\mathbf{v}_{d_1}$ should be ranked higher than $\mathbf{v}_{d_2}$, and 0 if the opposite is true. The cost function used by RankNet for learning is:

$$C = -T_{1 \triangleright 2} \log P(\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2}) - (1 - T_{1 \triangleright 2}) \log(1 - P(\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2})). \qquad (4)$$

C is the cross entropy between the two Bernoulli distributions $P(\mathbf{v}_{d_1} \triangleright \mathbf{v}_{d_2})$ and $T_{1 \triangleright 2}$. The parameters of the ranking function are learned via gradient descent on C. C is a smooth differentiable function of W, and an expression for $\frac{\partial C}{\partial W}$ can be derived analytically. Equation 4 considers only a single training triplet, but it can be applied to the case where multiple training triplets are available by simply minimizing the average cost of all the triplets.

Note that with RankNet only the way in which the triplets are defined differs from task to task. The training cost function and the optimization procedure stay the same across tasks. In contrast, LambdaRank allows a *different cost function* for each task, which makes it possible to customize the learning.

3.5 Learning with LambdaRank

We begin with a general description of LambdaRank, and then describe the adaptations specific to learning a bit vector representation for classification and retrieval in sections 3.5.1, 3.5.2, and 3.5.3.

We keep the parameterization of the ranking function and the definition of the ranking score the same as before, but change the learning algorithm to LambdaRank (figure 3). This affects two things – 1) the objective function used for learning, and 2) the procedure for computing its gradient.

For simplicity we present LambdaRank here as a modified version of RankNet, but the underlying ideas are more general. Consider again a RankNet training triplet $(\mathbf{v}_q, \mathbf{v}_{d_1}, \mathbf{v}_{d_2})$ where $\mathbf{v}_q$ is a query, and $\mathbf{v}_{d_1}$ and $\mathbf{v}_{d_2}$ are two documents to be ranked with respect to $\mathbf{v}_q$. Suppose that we want the learning to optimize the ranking function for an evaluation score S. S can be a listwise ranking score, e.g. NDCG. Then LambdaRank modifies the RankNet

gradient as follows:

$$\frac{\partial C}{\partial W} |\Delta S| \qquad (5)$$

where C is the RankNet cost function (equation 4) and W is the weight matrix to be learned. $|\Delta S|$ is the absolute difference in the value of S due to swapping the positions of $\mathbf{v}_{d_1}$ and $\mathbf{v}_{d_2}$ in the ordering of *all* documents, with respect to $\mathbf{v}_q$, computed by the current ranking function.

Note that LambdaRank learns on triplets, as before, but now only those triplets that produce a non-zero change in S by swapping the positions of the documents contribute to the learning. Given a query, first all documents are ordered with respect to it using the ranking function given by the current W. Evaluating this ordering will give some score S_1. Now pick any two documents and swap their positions in the ordering. Evaluating this new ordering will give some score S_2. LambdaRank then computes the RankNet gradient for those two documents and the query, and multiplies it by $|S_1 - S_2|$. Only the score function S needs to be changed from task to task in LambdaRank.

We now describe aspects of LambdaRank training that are specific to our approach.

3.5.1 Score function for classification:

Suppose that for a query $\mathbf{v}_q$, L documents $\{\mathbf{v}_{d_1}, \mathbf{v}_{d_2}, ..., \mathbf{v}_{d_L}\}$ are selected as neighbors (as explained later). Let l_q and l_{d_i} be the class labels of the query and the ith document, respectively. The score function S_C for nearest neighbor classification of a single query is:

$$S_C = [MAJORITY(l_{d_1}, l_{d_2}, ..., l_{d_L}) = l_q], \qquad (6)$$

where the $MAJORITY(\cdot)$ function picks the most frequent label in $\{l_{d_1}, l_{d_2}, ..., l_{d_L}\}$ and $[\cdot]$ is Iverson notation denoting an output 1 if the argument is true and 0 otherwise.

One disadvantage of the above score is that it does not reflect incremental progress towards the correct answer. For example, if the correct label is three votes short of becoming the majority, then a swap that increases its vote count by 1 will not change the above score. But such a swap moves the result closer to the correct answer and therefore should be used. To allow for such swaps, we use the following score:

$$S_C = \sum_{i=1}^{L} [l_{d_i} = l_q]. \qquad (7)$$

Note that without the $MAJORITY(\cdot)$ function, even swaps that would make *all* the neighbors have the same label as the query would be allowed. This is more stringent than necessary for nearest neighbor classification. A score function with a looser requirement may give better accuracy, but we have not yet explored that.

Thresholding for neighbor selection: There are two types of thresholding that can be used on the Hamming distance. An *absolute* threshold k selects all documents with Hamming distance $\leq k$ from the query as neighbors. A *relative* threshold k selects only those documents in the k nearest *non-empty* Hamming distance bins from the query as neighbors. Consider an example where the nearest documents from the query appear at say, distances 4, 7, 12, 17, 20, etc. and $k = 3$. With an absolute threshold, no documents would be selected, which makes nearest neighbor classification ambiguous. But with a relative threshold, documents at distances 4, 7, and 12 would be selected. Since a relative threshold guarantees that the set of neighbors will always be non-empty, we use it for nearest neighbor classification.

Resolving ties: If we use a B-dimensional bit vector representation, then Hammming distance can take on only $B + 1$ possible values (0 to B). If the number of documents being ranked is much greater than $B + 1$, then there will be a lot of ties in the ranking. For nearest neighbor classification, we do not break ties. Instead we use all the documents from the selected bins to vote on the class label.

3.5.2 Score function for retrieval:

In section 5 we consider two retrieval tasks where the relevance label of a document is binary (either 0 or 1), and accuracy is measured using precision (# of retrieved documents that are relevant divided by # of retrieved documents). We define the score function for this particular setting. Again consider a query $\mathbf{v}_q$, and L documents $\{\mathbf{v}_{d_1}, \mathbf{v}_{d_2}, ..., \mathbf{v}_{d_L}\}$ retrieved for that query with an absolute threshold k. Let l_{d_i} be the relevance label for the ith document. The score function S_R for retrieval is:

$$S_R = \sum_{i=1}^{L} l_{d_i}. \qquad (8)$$

We do not normalize the score by L in order to prevent queries that have a small number of retrieved documents from contributing disproportionately to the gradient.

Resolving ties: During learning, we do not break ties in the ranking. But during testing, tie-breaking may be needed. For example, consider a task that requires retrieving exactly 100 documents for a given query. If the nearest Hamming distance bin to the query contains 110 documents, then some re-ranking procedure would be needed to select exactly 100[6]. In such an application a bit vector representation is still useful for rapidly *shortlisting* a small set of documents for the re-ranker. Note that tie-breaking is needed for any approach that uses bit vectors, not just ours.

3.5.3 Procedure for finding swaps with $|\Delta S| > 0$

A key step in the per-query gradient computation of LambdaRank is to identify only those pairwise swaps that have nonzero $|\Delta S|$ (steps 3 and 4 in figure 3). At first glance, finding such swaps may appear to be a very expensive computation that requires sorting all the documents in the training set for every query. But in the case of learning a bit vector representation, a fast approximation turns out to be possible.

Given a query, there are only a fixed number of Hamming distance bins that the documents can belong to. For efficiency, we use swaps only among those documents that belong to a small subset of bins closest to the query. Finding documents that belong to the nearest bins can be done efficiently using a heap structure, without having to compute the *full* Hamming distance and sorting over the entire training set.

4. IMPLEMENTATION DETAILS

Subsampling: To significantly speed up the per-query gradient computation, we subsample the set of documents from which document pairs are selected for each query. A noisy estimate of the gradient can be computed cheaply from a small subset of the full document set. This is helpful when the training set is large and computing the gradient from the full set is too expensive.

In the case of RankNet, for each query we use only a small subset of the full training set to generate pairs for that query. Most of our experiments (section 5) use only 100 randomly chosen documents per query to generate triplets.

---

[6]One possibility is to sort by the real-valued score $\hat{s}_i$ (equation 2) and select the top 100.

> **LambdaRank learning algorithm:**
> **Training inputs:**
> - A ranking metric M (e.g. NDCG) to be optimized.
> - A set of training examples where the ith example contains:
>
>   1. Query $\mathbf{v}_q^i \in \Re^D$,
>
>   2. List L_i of N documents $\{\mathbf{v}_{d_1}^i, ..., \mathbf{v}_{d_N}^i\}$ all in $\Re^D$.
>
> - Number of bits B.
> - Learning rate parameters: step-size η, momentum m.
>
> **Training outputs:** $D \times B$ weight matrix W.
> **Initialization:** W_{ij} are sampled from zero-mean Gaussian with small (10^{-3}) variance, ΔW = zero matrix.
>
> **Weight update computed using the ith training case:**
>   1. Compute real-valued representation $\mathbf{h}_q$ for $\mathbf{v}_q^i$ using equation 1.
>
>   2. For the kth document in L_i compute:
>
>   (a) Compute real-valued representation $\mathbf{h}_k$ for $\mathbf{v}_{d_k}^i$ using equation 1.
>
>   (b) Compute score $\hat{s}_k$ using equation 2 from $\mathbf{h}_q$ and $\mathbf{h}_k$.
>
>   3. Compute an ordering O of the documents in L_i by sorting them in ascending order of the scores $\hat{s}_k$. Let S_O be the value of the evaluation score S for O. For an efficient approximation, see sections 3.5.3 and 4.
>
>   4. Find the set V of pairs of documents $(\mathbf{v}_a^i, \mathbf{v}_b^i)$ selected from L_i such that swapping the positions of $(\mathbf{v}_a^i$ and $\mathbf{v}_b^i)$ in O results in a new ordering O' for which $|S_O - S_{O'}| > 0$.
>
>   5. For each element $(\mathbf{v}_a^i, \mathbf{v}_b^i) \in V$ compute $P(\mathbf{v}_a^i \rhd \mathbf{v}_b^i)$ using equation 3.
>
>   6. Compute the gradient $\frac{\partial C}{\partial W}$ of the cost function C (equation 4) with respect to W.
>
>   7. $\Delta W \leftarrow m\Delta W - \eta(|S_O - S_{O'}|\frac{\partial C}{\partial W})$.
>
>   8. $W \leftarrow W + \Delta W$.

Figure 3: Summary of LambdaRank learning algorithm.

For LambdaRank, we restrict the number of bins from which documents are considered for swaps to be one-third of the number of bits. For the tasks considered here, swaps only happen between a document in one of the top k bins and a document outside of the top k bins, but still within the restricted set. We can make this even more efficient by considering only a subset of the full training set (L_i in step 2 of figure 3) to populate these bins. How much subsampling can be done without degrading accuracy depends on the dataset, but in our experiments we have seen that good results can be achieved even with $10\times$ subsampling.

Gradient descent: The weight matrix W is updated by averaging gradient estimates given by a set of queries and taking a step along the average gradient. Averaging can reduce the noise in the updates. The average is typically computed over 100 queries. One can easily parallelize this computation by splitting the set of queries across multiple cores and then averaging together the gradients computed at all cores.

We use a fixed step size to update W. The best value depends on the dataset – for the datasets in section 5 we have tried values in the range $[10^{-1}, 10^{-4}]$. On some datasets we have observed significant sensitivity in the accuracy to the step size value. We set the step size to the highest possible value that does not produce large oscillations in the objective function value during optimization.

We have not yet tried any second-order optimization methods like conjugate gradient to improve convergence speed. Such methods may help in RankNet training, but are unlikely to be useful

for LambdaRank since the actual objective function is implicit and cannot be directly evaluated.

As in [11], we maintain an exponentially decaying sum of the previous gradients which is added to the current gradient to compute the weight update. The decay factor is set to 0.8, so the effect of the gradient computed at a particular step persists for several steps afterwards.

5. EXPERIMENTAL EVALUATION

Performance is evaluated on two types of tasks: 1) nearest neighbor classification and 2) retrieval. We present results for four classification and two retrieval datasets.

5.1 Metrics

The evaluation metric for classification is the number of incorrect label predictions on a test set. Given a test case, its bit vector is compared against the bit vectors for all the training cases. Neighbors are selected using a relative threshold k – those training cases that fall within the nearest k *non-empty* Hamming distances to the test case bit vector are returned. The predicted label is then picked by a majority vote among the neighbors.

For retrieval we use the same precision metric as in Weiss et al. [23]. Given a query, its bit vector is compared against the bit vectors for all the documents in the search set. Documents are selected using an absolute threshold – those documents that are less than a pre-specified Hamming distance threshold from the query bit vector are retrieved. We consider binary relevance labels here[7] – a document is either relevant for a query or not. So precision for a single query is computed as follows:

$$\text{Precision} = \frac{\text{\# of relevant documents in the retrieved set}}{\text{\# of documents in the retrieved set}}. \quad (9)$$

This quantity is then averaged over a held-out set of queries to get a single precision value.

5.2 Datasets

Tables 2 and 3 summarize the datasets. The datasets span a range of training set sizes (60K to 1.45 million), input dimensionality (128 to 47K), and in the case of classification, number of classes (7 to 101). They include a number of different types of data – images (MNIST, INRIA SIFT 1M, Tiny Images Subset), text documents (MCAT, RCV), and geospatial measurements (Covertype).

We briefly describe each dataset:

MNIST[8] is a set of handwritten images of the digits 0 to 9. The images are grayscale and of size 28×28.

MCAT contains text documents that belong to a subtree of the Reuters Corpus Volume 1 (RCV1) dataset [14]. A document is represented as a bag-of-words with a vocabulary size of 11429. Only about 0.58% of the word counts are nonzero, so the representation is sparse.

Covertype[9] [3] is a dataset of geospatial measurements that are used to predict the forest covertype at various locations in the US.

RCV1 Subset[10] contains only those documents in the RCV1 dataset that do not have multiple labels associated with them. As in MCAT, a document is represented as a bag-of-words, but with a much

---

[7] The two retrieval datasets we use supply binary relevance labels.

[8] http://yann.lecun.com/exdb/mnist/

[9] http://archive.ics.uci.edu/ml/datasets/Covertype

[10] http://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/ binary.html\#rcv1.binary

Dataset	Train set size	Test set size	Input dim.	Number of classes
MNIST	60000	10000	784	10
MCAT	150344	4362	11429	7
Covertype	522911	58101	54	7
RCV1	531742	15913	47236	101

Table 2: Classification datasets.

Dataset	Train set size	# of test queries	# of test docs to search	Input dim.
INRIA SIFT1M	100000	10000	1000000	128
Tiny Images Subset	1458356	100000	1458356	512

Table 3: Retrieval datasets.

larger vocabulary size of 47236 and 101 classes. The representation is 0.14% sparse.

INRIA SIFT1M[11] is a web image dataset designed for evaluating retrieval algorithms. It consists of 128-dimensional SIFT features [15] computed for images collected from the web. Given a query image's SIFT feature vector, the relevant images to retrieve are defined to be its 50 nearest neighbors, according to Euclidean distance, in the SIFT feature space.

Tiny Images Subset is derived from the Tiny Images dataset [21], which contains 80 million images collected from the web. Various text queries were given to popular image search engines and the results were downloaded. The subset we use here contains only the top-ranked images for the query terms, so it is likely to be less noisy than the full 80 million set. The images are represented using 512 dimensional GIST feature vectors [17]. The retrieval task is defined in the same way as in INRIA SIFT1M: the goal is to retrieve the 50 nearest neighbors, according to Euclidean distance, of a query image's GIST feature vector.

5.3 Baselines

We compare against other methods that compute a bit vector representation. For retrieval, Spectral Hashing is a state-of-the-art method. Binary LSH is a commonly used baseline in the literature, so we compare against it as well. We use the Matlab implementation of Spectral Hashing by the authors of that paper[12].

For classification, we again use Spectral Hashing and LSH as baselines. However these methods were not intended to be used for classification, so they cannot be taken as strong baselines. Therefore we decided to compare also against nearest neighbor methods that do not use bit vectors. The simplest one is kNN classification with L_2 distance. State-of-the-art learning methods are NCA and LMNN. LMNN does not scale to datasets with more than a few tens of thousands of training cases [22], so we cannot run it on Covertype, MCAT, and RCV1 Subset. For MNIST we quote the LMNN classification error from [22]. For NCA, we use the implementation in the Matlab Toolbox for Dimensionality Reduction[13].

Note that one can always apply binary LSH on top of a metric learning method like NCA or LMNN to construct bit vectors. But such a two-stage approach will at best give the same accuracy as the

---

[11] http://corpus-texmex.irisa.fr/

[12] www.cs.huji.ac.il/~yweiss/SpectralHashing/

[13] http://homepage.tudelft.nl/19j49/Matlab_Toolbox_for_Dimensionality_Reduction.html

underlying real-valued metric since binary LSH is only an approximation of it. So we compare against the accuracy of the real-valued metric directly, i.e. the best-case result for the two-stage approach.

On the RCV1 Subset, Spectral Hashing required doing an eigenvalue decomposition of the 47236×47236 data covariance matrix and selecting the top eigenvectors. For 32 and 64 bits, Matlab was not able to perform this computation due to excessive memory use, even with 32GB of RAM. Those results are not given. In NCA, the dimensionality of the output of the linear projection matrix can be made less than the input dimensionality. The dimensionality we chose is 60 for MNIST, 500 for MCAT, 54 for Covertype, and 500 for RCV. On MNIST 60 gave the same accuracy as bigger values. On Covertype the input dimensionality is already small (54), so a lower number was not tried. On MCAT and RCV, we chose 500 to keep the training times reasonble.

5.4 Classification Results

Table 4 shows the test set error rates for the four classification datasets. We use a relative threshold of $k = 3$ for all methods that use bit vectors. LambdaRank achieves the lowest classification error on three out of four datasets. Among bit vector methods, LambdaRank is 48.9% better than others (averaged over four datasets), with RankNet being the closest competitor. LambdaRank is on average 26.8% better than the best RankNet result. Only on the Covertype dataset does RankNet perform comparably to LambdaRank. So task-specific optimization by LambdaRank improves accuracy, and as MNIST and Covertype results show, the improvement can be substantial (+54% and +42%, respectively).

Recall from section 3.5.1 that there can be ties in the Hamming distance ranking. A relative threshold k only guarantees that the number of neighbors used to classify a given test case is at least k. The actual number can be much larger than k if there are many ties. All the four bit vector methods compared in table 4 share this property. The better accuracy of LambdaRank despite this commonality implies that simply having a large number of neighbors to classify a test case is not sufficient for high accuracy, and that learning a good bit vector representation is also crucial.

The benefit of LambdaRank can also be measured in terms of the bit "compression" it gives with respect to the other methods while matching their best error. In many cases LambdaRank needs significantly fewer bits. To get a rough idea of how big the compression factor is, we linearly interpolate between the datapoints in table 4 to determine the number of bits needed by LambdaRank to achieve a particular error rate. On average the ratio of the number of bits needed by LSH, Spectral Hashing and RankNet to achieve their best error rate divided by the number of bits needed by LambdaRank to achieve the same error rate is $7.1\times$, $4.1\times$, and $2.0\times$, respectively.

5.5 Retrieval Results

Table 5 shows the precision results computed on the test sets of INRIA SIFT1M and Tiny Images Subset. Again, LambdaRank gives the best results on both datasets. It gives $1.2\times$ and $2.66\times$ better precision than Spectral hashing on the two datasets.

The improvements over RankNet are again substantial for both datasets: +6.5% and +13.6%. This adds further evidence to the usefulness of task-specific learning.

5.6 Training and Testing Times

The training time of our algorithm scales well to large datasets. On Tiny Images Subset (1.46 million training cases), learning with 8-core parallelization converges in approximately 2 days on an Intel Xeon 2.50GHz machine. Figure 4 shows the convergence be-

No. of bits	LSH	Spectral hashing	Rank Net	Lambda Rank	Non-bit vector methods
MNIST					
8	58.73	33.63	12.64	12.32	
16	52.15	19.67	8.50	7.57	kNN, L_2: 3.09
32	24.61	10.10	5.54	4.85	NCA: 2.45
64	13.20	6.82	4.20	4.02	LMNN: 1.72
128	7.71	5.74	4.01	2.37	
256	4.64	4.63	3.55	**1.63**	
MCAT					
8	67.35	24.97	1.70	1.54	
16	60.39	8.02	1.38	1.47	kNN, L_2: 3.67
32	42.85	5.36	1.42	**1.31**	NCA: 7.66
Covertype					
8	42.07	42.17	29.00	30.36	
16	33.96	34.30	26.54	21.60	kNN, L_2: 6.25
32	20.65	27.29	21.62	14.24	NCA: **4.01**
64	12.33	14.68	18.50	10.88	
128	9.55	9.46	15.40	7.74	
256	7.64	7.44	9.58	5.52	
RCV1					
8	84.63	75.24	45.60	25.21	
16	75.20	59.46	18.56	15.93	kNN, L_2: 29.67
32	63.79	-	16.84	14.41	NCA: 45.79
64	50.95	-	13.34	**12.58**	

Table 4: Classification error (%) on the test sets of MNIST, MCAT, Covertype, and RCV1.

No. of bits	LSH	Spectral hashing	Rank Net	Lambda Rank
INRIA SIFT1M (Precision $\times 10^{-4}$)				
8	6.11	14.26	15.00	17.16
16	44.70	200.04	184.61	266.63
32	476.29	1462.76	1687.21	1805.25
Tiny Image Subset (Precision $\times 10^{-5}$)				
8	6.93	28.73	36.85	42.93
16	34.58	211.39	430.98	578.99
32	410.41	3396.62	7979.02	9065.89

Table 5: Precision at Hamming distance < 2 from the query on the test sets of INRIA SIFT1M and Tiny Images Subset.

haviour. Even after one pass through the data, the precision on the test set has already reached 90% of its final value. The memory needs of the learning algorithm are minimal. The weight matrix, its gradient, and the bit vectors for the training set (computed using the current weight matrix) account for most of the memory use, and all of these together takes up much less memory than the training data itself (e.g. for Tiny Images Subset these variables take up 0.2% of the memory occupied by the training data).

Table 6 shows the CPU times needed for finding the nearest neighbors for the test set of different datasets by the different algorithms. Note that nearest neighbor classification with bit vectors is substantially faster than with real-valued representations, even without using an inverted index. This is because the distance calculation for bit vectors is done using bitwise XOR on chunks of 8 bits, and the 8-bit result is converted into a Hamming distance with a lookup table. The distances from the 8-bit chunks are then added together to get the full distance. So computing the Hamming dis-
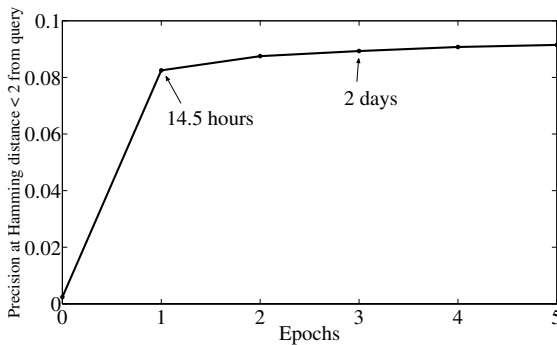
Figure 4: Precision on the test set as a function of the number of passes through the Tiny Images Subset training data. Note that most of the improvement happens in the first pass.

Dataset	Lambda Rank	L_2 kNN	NCA
MNIST	23.99	756.81	121.19
MCAT	8.85	385.13	707.84
Covertype	765.83	1744.83	1785.33
RCV1 Subset	127.78	5457.93	10706.11

Table 6: CPU time (seconds) required to find the nearest neighbors for the entire test set by different methods on the various datasets.

tance between two 256-bit vectors requires only 256 bitwise XORs, 32 accesses into a lookup table and 32 adds.

For INRIA SIFT1M and Tiny Images Subset, the number of bits in the learned representation is small enough to build an inverted index that fits into 4GB RAM. We use subroutines from the Spectral Hashing software (see footnote 11) to build the inverted index and use it for search. The CPU times for finding the nearest neighbors on the test sets of INRIA SIFT1M and Tiny Images Subset with a 32-bit representation and an inverted index are 0.10s and 1.37s, respectively, when run on a single core of an Intel Xeon 2.33GHz machine with 4GB RAM. This translates to an average search time per query of 10 microseconds for INRIA SIFT1M and 13.7 microseconds for Tiny Images Subset. Note that linear search on the same 32-bit representation is much slower than the inverted index. The total CPU times for finding the nearest neighbors on the test sets of INRIA SIFT1M and Tiny Images Subset are 63.09s and 1674.40s, respectively.

6. EXTENSIONS AND FURTHER APPLICATIONS

Learning nonlinear features: Currently the ranking function does not contain any learnable nonlinear features of the input vector. Including such features is easy and can make the function more flexible and accurate. For example, one can use *hidden units* from the neural network literature to learn nonlinear features.

Cascade architecture: A common way to speed up search is to use a multi-stage, cascade architecture where a fast first stage search rules out a large fraction of the search set, followed by increasingly slower stages that only need to search the set retrieved by the previous stage. Such an architecture can also be used in our case – different ranking functions can be cascaded in increasing order of the number of bits, with each stage searching only among the neighbors found by the previous stage. We have tried a two stage

classifier on the MNIST dataset with an 8-bit ranking function first and then a 256-bit classifier. The search set size for the 256-bit classifier reduces by an order of magnitude with almost no loss in accuracy.

Note that with LambdaRank it is possible to modify the *training* of the initial filtering stages such that they are explicitly trained to filter (and not classify). We have not yet explored this option.

Hierarchical classification: If a hierarchy over classes is given in a classification task, then the score function for LambdaRank can be modified to incorporate this information. Instead of binary relevance (1 for a document with the same label as the query, 0 otherwise), now we use *multiple* relevance levels to represent varying degrees of similarity between two classes. The similarity between two classes can be defined in many ways, e.g. with a monotonically decreasing function of the height of the common ancestor of the two classes in the hierarchy. For a query $\mathbf{v}_q$ and L documents $\{\mathbf{v}_{d_1}, \mathbf{v}_{d_2}, ..., \mathbf{v}_{d_L}\}$ retrieved from the top K non-empty Hamming distance bins, we can modify the original classification score function (equation) as follows:

$$S_{Hier} = \sum_{i=1}^{L} F(l_q, l_{d_i}). \qquad (10)$$

where l_q and l_{d_i} are the labels of the query and the ith document, respectively, and $F(a, b)$ is a function that specifies the similarity between classes a and b.

Regression: As mentioned before, a bit vector representation for nearest neighbor regression can be learned using our approach. In the case of RankNet, pairwise ranking of a document pair with respect to a query can be done using the absolute difference between the query target and a document target. The document with the closer target to the query in the pair should get ranked higher.

For LambdaRank we need to define a score function. Consider a query $\mathbf{v}_q$, and L documents $\{\mathbf{v}_{d_1}, \mathbf{v}_{d_2}, ..., \mathbf{v}_{d_L}\}$ retrieved from the top K bins for that query. Let t_q and t_{d_i} be the regression target values for the query and the ith document. The score can be the mean squared error between t_q and the document targets:

$$S_{Regression} = \frac{1}{L} \sum_{i=1}^{L} (t_q - t_{d_i})^2. \qquad (11)$$

As a preliminary experiment, we have trained a 64-bit RankNet model on the SARCOS dataset[14] (see section 2.5 of [18]). It achieves a better standardized mean squared error than a linear regression model. Experiments with LambdaRank is left as future work.

One potential application of nearest neighbor regression is in Collaborative Filtering. Neighborhood based models have been shown to be useful for predicting the rating a user would give to an item [2]. For example, in a user-based neighborhood model, given a user-item pair for which to predict a rating, a bit vector-based representation can be used to retrieve similar users who have rated the same item. The predicted rating is then computed as a weighted average of the ratings of the retrieved users.

7. OTHER RELATED WORK

We discussed several popular methods *closely* related to our work in the introduction, and empirically compared with representative methods. Here, we present other related work.

Hashing Methods Kernel LSH [13] is a recent scheme that generalizes LSH, and is useful when similarity measure is given directly via a kernel function (without explicit knowledge of the underlying transformation), or the underlying transformation is infinitely

---

[14]http://www.gaussianprocess.org/gpml/data/

dimensional (hence, incomputable). He et al. [10] propose a joint optimization method to optimize the codes for both preserving similarity as well as minimizing search time.

The main drawback of these hashing approaches is that they cannot be directly used in applications where we are not given a similarity metric but rather class/relevance labels that indicate which data points are similar or dissimilar to each other.

Metric Learning Methods There are other methods that learn the distance metric by optimizing task specific performance measure. Classification task oriented methods such as NCA and LMNN discussed before fall under this category. Other approaches include Fisher's linear discriminant analysis (FLDA), maximally collapsing metric learning algorithm (MCML) [8], relevance component analysis (RCA) [20], etc. While FLDA and MCML use class label information, RCA assumes that a set of *chunklets* is available, where each chunklet is a set of examples which belong to the same class (but, the class information is unknown; hence, RCA can be seen as a weaker form of supervised learning). Both FLDA and RCA involve matrix inversion (in the dimension of input space), and use projections on eigen vectors for nearest neighbor classification. Therefore, they are computationally expensive for high dimensional data. MCML method tries to collapse all examples belonging to the same class into a single point, and keep examples belonging to other class far away. Like LMNN, MCML uses positive definite matrix constraints during training. All these methods are not scalable, and are limited to classification application.

To summarize, our work can be seen as merging two streams of work, one for learning a task-specific metric from labeled data, and the other for learning bit vector representations for doing fast search. To our knowledge this is the first attempt that combines the two types of learning into a unified framework.

8. CONCLUSIONS

We have presented a unified approach to learning a bit vector representation for nearest neighbor search in different tasks. The key contribution is the ability to customize the learning algorithm to the task at hand by modifying the score function used in LambdaRank training. What makes the unified approach possible is that the differences across the tasks are abstracted out of the core nearest neighbor search problem and pushed into the score function in LambdaRank. As a result the same algorithm can be easily adapted to different tasks to learn accurate representations for each. Experimental results clearly demonstrate that our algorithm 1) outperforms other methods for nearest neighbor classification and retrieval, and 2) scales well to large text and image datasets, making it particularly useful for web applications.

9. REFERENCES

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *In FOCS 2006*, pages 459–468. IEEE Computer Society, 2006.

[2] R. Bell, Y. Koren, and C. Volinsky. Modeling relationships at multiple scales to improve accuracy of large recommender systems. In *SIGKDD*, KDD '07, pages 95–104, New York, NY, USA, 2007. ACM.

[3] J. A. Blackard and D. J. Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and Electronics in Agriculture*, vol.24:131–151, 1999.

[4] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, ICML '05, pages 89–96, New York, NY, USA, 2005. ACM.

[5] C. J. Burges, R. Ragno, and Q. V. Le. Learning to rank with nonsmooth cost functions. In B. Schölkopf, J. Platt, and T. Hoffman, editors, *NIPS 19*, pages 193–200. MIT Press, Cambridge, MA, 2007.

[6] G. Chechik, V. Sharma, U. Shalit, and S. Bengio. Large scale online learning of image similarity through ranking. *J. Mach. Learn. Res.*, 11:1109–1135, March 2010.

[7] P. Donmez, K. M. Svore, and C. J. Burges. On the local optimality of lambdarank. In *SIGIR*, pages 460–467, New York, NY, USA, 2009. ACM.

[8] A. Globerson and S. T. Roweis. Metric learning by collapsing classes. In *NIPS*, pages –1–1, 2005.

[9] J. Goldberger, S. Roweis, G. Hinton, and R. Salakhutdinov. Neighbourhood components analysis. In *NIPS 17*, pages 513–520. MIT Press, 2004.

[10] J. He, R. Radhakrishnan, S.-F. Chang, and C. Bauer. Compact hashing with joint optimization of search accuracy and time. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2011.

[11] G. Hinton and R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504 – 507, 2006.

[12] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. *CVPR*, 0:1–8, 2008.

[13] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, 2009.

[14] D. D. Lewis, Y. Yang, T. G. Rose, F. Li, G. Dietterich, and F. Li. Rcv1: A new benchmark collection for text categorization research. *JMLR*, 5:361–397, 2004.

[15] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vision*, 60:91–110, November 2004.

[16] B. Mcfee and G. Lanckriet. Metric learning to rank. In *ICML*, 2010.

[17] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *IJCV*, 42:145–175, 2001.

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

[19] R. Salakhutdinov and G. Hinton. Semantic Hashing. In *SIGIR workshop on Information Retrieval and applications of Graphical Models*, 2007.

[20] N. Shental, T. Hertz, D. Weinshall, and M. Pavel. Adjustment learning and relevant component analysis. In *ECCV*, ECCV '02, pages 776–792, London, UK, UK, 2002. Springer-Verlag.

[21] A. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *PAMI*, 30:1958–1970, November 2008.

[22] K. Q. Weinberger and L. K. Saul. Distance metric learning for large margin nearest neighbor classification. *JMLR*, 10:207–244, June 2009.

[23] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.