Bregman hyperplane trees for fast approximate nearest neighbor search

November 5, 2012

Abstract

We present a new approximate indexing structure, the Bregman hyperplane tree, for indexing the Bregman divergence, aiming to decreasing the number of distance computations required at query processing time, by sacrificing some accuracy in the result. The experimental results on various high-dimensional data sets demonstrate that the proposed indexing structure performs comparable to the state-of-the-art Bregman ball tree in terms of search performance and result quality. Moreover, our method results in a speedup of well over an order of magnitude for index construction. We also apply our space partitioning principle to the Bregman ball tree and obtain a new indexing structure for exact nearest neighbor search that is faster to build and a slightly slower in query processing than the original one.

1 Introduction

Efficient nearest neighbor processing is important in multimedia applications. In such applications, the real objects (image, audio etc.) are usually represented as vectors in a high-dimensional space. Then the nearest neighbor search is to retrieve all those points in a database that are close to a query. One distance measure that has many applications, but which so far has received little attention in the similarity retrieval community, is the Bregman divergence. Bregman divergences have many real-world applications, and are used to measure dissimilarity in such diverse areas as multimedia, machine learning, computational geometry and speech recognition. For example, the Kullback-Leibler divergence is used to assess the similarity between histograms and has been applied to image classification (Rubner et al., 1999) and image retrieval (Rasiwasia et al., 2007). Another example is the Itakura-Saito divergence (Itakura and Saito, 1970), used in speech recognition as a measure of the difference between signals.

Numerous indexing structures have been proposed for similarity retrieval under vector norms and other metrics (Samet, 2005; Zezula et al., 2006). Bregman divergences, however, are non-metric, and therefore not amenable to indexing

using these techniques. Because the most important base property of these techniques is the triangle inequality which is invalid for the Bregman divergences.

We propose an index structure for Bregman divergences, based on the hyperplane partitioning principle. The proposed index structure is easier to implement than the main competing method, and we show that it achieves a manifold speedup over a linear scan with a low degree of error. This indexing structure was considered in conference version of our paper. In this paper, we extend our previous paper and apply the hyperplane partitioning principle to the Bregman ball tree, and obtain a new version of the ball tree. The main purpose of this tree was to see the practical applicability of the hyperplane partitioning principle in exact nearest neighbor search. The rest of the paper is organized as follows: Some preliminary definitions are given in Section 2. Section 3 is referred to the related work. We describe our indexing structures in Section 4. Experimental results are provided in Section 5. Finally, Section 6 contains some concluding remarks.

2 Preliminaries

Let $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ be our database. Let $f : \mathbb{R}^D \to \mathbb{R}$ be a strictly convex differentiable function. The Bregman divergence (Bregman, 1967) between $x, y \in \mathcal{X}$ based on f is defined as

$$d_f(x, y) \equiv f(x) - f(y) - \langle \nabla f(y), x - y \rangle . \tag{1}$$

Here, $\nabla f(y)$ denotes the gradient of the function f at y and $\langle x, y \rangle$ denotes the dot product between x and y. A Bregman divergence measures the distance between f and its first-order Taylor expansion. The definition is illustrated in Figure 1.

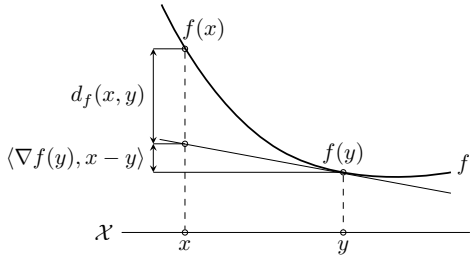

Figure 1: Illustration of the Bregman divergence $d_f(x, y)$.

Some standard Bregman functions and their divergences are listed in Table 1.

Note that, as opposed to most distances used in similarity retrieval, Bregman divergences are not, in general, metric; they are neither symmetric nor triangular.

Name	Domain	Function f	$d_f(x, y)$
Squared function (ℓ_2^2)	$\mathbb{R}^D$	$\frac{1}{2}\|x\|_2^2$	$\frac{1}{2}\|x - y\|_2^2$
Kullback-Leibler div.	$\mathbb{R}_+^D$	$\sum x_i \log x_i$	$\sum x_i \log \frac{x_i}{y_i} - x_i + y_i$
Itakura-Saito	$\mathbb{R}_+^D$	$-\sum \log x_i$	$\sum \left(\frac{x_i}{y_i} - \log \frac{x_i}{y_i} - 1\right)$
Mahalanobis	$\mathbb{R}^D$	$\frac{1}{2}x^\top Q x$	$\frac{1}{2}(x - y)^\top Q(x - y)$
Exponential	$\mathbb{R}^D$	$\sum e^{x_i}$	$\sum e^{x_i} - (x_i - y_i + 1)e^{y_i}$

Table 1: Some standard Bregman functions and their divergences.

3 Related Work

Cayton (2008) proposed the first general index structure for Bregman proximity search, the Bregman ball tree (bbtree). The bbtree is a binary tree where and each node represents a Bregman ball that covers all points in the subtree rooted at that node. The k-means algorithm is employed as the space partitioning method, using $k = 2$, yielding two subsets. The tree is built recursively, in a top-down manner. The coordinates of a node's center is computed as the mean of the coordinates of all points in the subset. Thus data points are located in the leaf nodes, the index will have more than n points. Nearest neighbor search is performed using the depth-first branch-and-bound strategy. In order to check overlap between a Bregman ball and a query ball, a bisection is performed on a line between the ball center and the query to find the lower and upper bounds from the ball to the query. The processing of a query may require more than n distance computations due to the node center selection criteria. Note that this method supports both exact and approximate nearest neighbor search.

Nielsen et al. (2007) studied the geometric properties of Bregman Voronoi diagrams. The Voronoi diagrams are closely related to nearest neighbor search, and form the basis of several metric indexing structures, such as the spatial approximation tree (Navarro, 2002). However, Cayton (2008) claimed that Voronoi diagrams do not lead to an efficient nearest neighbor data structure for Bregman divergences beyond two dimensions.

A space transformation method for Bregman divergence is proposed by Zhang et al. (2009): the points in the original d-dimensional space are transformed into a $d+1$ dimensional space. The authors developed several strategies that can efficiently handle data in the extended space and applied them to R-trees (Guttman, 1984) and VA-files (Weber et al., 1998).

4 Bregman hyperplane trees

First, let us have a look at a space partitioning principle that is used in our tree. We adapt the space partitioning principle proposed by Nielsen et al. (2007). Since Bregman divergences are asymmetric, we define two types of bisectors.

The Bregman bisector of the *first type* is defined for $p, q \in \mathcal{X}$ as

$$BB_f(p,q) = \{x \in \mathcal{X} \mid d_f(x,p) = d_f(x,q)\} \tag{2}$$

Similarly, the Bregman bisector of the *second type* is defined as

$$BB'_f(p,q) = \{x \in \mathcal{X} \mid d_f(p,x) = d_f(q,x)\} \tag{3}$$

The Bregman bisector of the first type $BB_f(p,q)$ can be written as the equation

$$BB_f(p,q,x) = 0\,, \tag{4}$$

where

$$\begin{aligned} BB_f(p,q,x) &= d_f(x,p) - d_f(x,q) \\ &= f(x) - f(p) - \langle \nabla f(p), x-p\rangle - f(x) + f(q) + \\ &\quad \langle \nabla f(q), x-q\rangle \\ &= f(q) - f(p) - \langle \nabla f(p), x\rangle + \langle \nabla f(p), p\rangle + \\ &\quad \langle \nabla f(q), x\rangle - \langle \nabla f(q), q\rangle \\ &= f(q) - f(p) + \langle \nabla f(p), p\rangle - \langle \nabla f(q), q\rangle - \\ &\quad \langle \nabla f(p) - \nabla f(q), x\rangle. \end{aligned}$$

Similarly, the Bregman bisector of the second type $BB'_f(p,q)$ is the hypersurface given by the equation

$$\begin{aligned} BB'_f(p,q,x) &= d_f(p,x) - d_f(q,x) \\ &= f(p) - f(x) - \langle \nabla f(x), p-x\rangle - f(q) + \\ &\quad f(x) + \langle \nabla f(x), q-x\rangle \\ &= f(p) - f(q) - \langle \nabla f(x), p\rangle + \langle \nabla f(x), x\rangle + \\ &\quad \langle \nabla f(x), q\rangle - \langle \nabla f(x), x\rangle \\ &= f(p) - f(q) + \langle \nabla f(x), q-p\rangle. \end{aligned}$$

Based on these equations, we define binary partitioning principles for our tree. For any given points $p, q, x \in \mathcal{X}$, *the first type* Bregman hyperplane partitioning is defined as follows.

$$\texttt{partition}(p,q,x) = \begin{cases} \text{left}, & \text{if } BB_f(p,q,x) < 0 \\ \text{right}, & \text{otherwise.} \end{cases} \tag{5}$$

Similarly, we have *the second type* Bregman hyperplane partitioning.

$$\texttt{partition}(p,q,x) = \begin{cases} \text{left}, & \text{if } BB'_f(p,q,x) < 0 \\ \text{right}, & \text{otherwise.} \end{cases} \tag{6}$$

Then, $\texttt{partition}(p, q, x)$[1] decides which (left or right) partition should contain point x. Note that the condition $d_f(x, p) < d_f(x, q)$ can be used in 5 instead of the evaluation of $BB_f(p, q, x) < 0$. Similarly, for the second type of partitioning we can evaluate $d_f(p, x) < d_f(q, x)$ in 6 instead of the evaluation of $BB'_f(p, q, x) < 0$. However, the latter (i.e., $BB_f(p, q, x)$ and $BB'_f(p, q, x)$) is computationally cheaper than the former. Note that, especially in the second type of partitioning, it is necessary to compute $\nabla f(x)$ only once for a given point x before the construction of tree (for all points in the data set) and query processing (for a query point) because $\nabla f(x)$ is used several times in the construction/traversal. It improves the performance greatly. Even for hyperplane points p and q, those constants that are independent from x (such as $f(p)$ and $\langle \nabla f(p), p \rangle$) are computed once and kept for further use.

The Bregman hyperplane tree (bhtree) is a (possibly unbalanced) binary tree, built in a manner similar to the generalized hyperplane trees (Uhlmann, 1991). In each recursive call of the tree construction, the size of data set $\mathcal{X}$ is compared to a given maximum leaf size and the current node becomes a leaf if the size falls below the threshold. Otherwise, the node is internal, and two representative points p_i and p_j are selected from $\mathcal{X}$ and stored in the node. The construction algorithm then applies the Bregman hyperplane partitioning $\texttt{partition}(p_i, p_j, x)$ to each point x of the data set, resulting two subsets $\mathcal{X}_l$ and $\mathcal{X}_r$, that is $\mathcal{X}_l \leftarrow \{x \mid x \in \mathcal{X} \backslash \{p_i \cup p_j\}, \texttt{partition}(p_i, p_j, x) = \mathit{left}\}$ and $\mathcal{X}_r \leftarrow \{x \mid x \in \mathcal{X} \backslash (\mathcal{X}_l \cup \{p_i \cup p_j\})\}$. The algorithm recursively builds the left and right subtrees for $\mathcal{X}_l$ and $\mathcal{X}_r$. An example of a node partitioning of the bhtree is given in Figure 2.

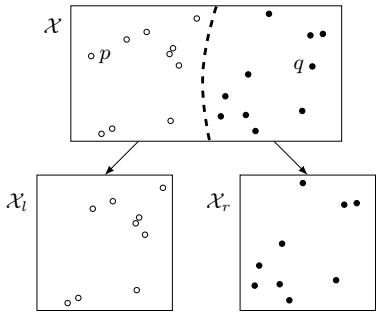

Figure 2: A data set $\mathcal{X}$ is recursively divided into two disjoint subsets $\mathcal{X}_l$ and $\mathcal{X}_r$ by applying the Bregman hyperplane partitioning based on two points $p, q \in \mathcal{X}$. The hyperplane is represented as a dashed line. Note that p and q are kept in the current node and not included in either of $\mathcal{X}_l$ and $\mathcal{X}_r$.

The exact query processing in bhtrees is quite expensive. The Bregman divergence from the query center to the hyperplane (i.e., the closest point)

[1] This journal version of the article has fixed bugs from the conference paper.

can be solved by an optimization problem, with the hyperplane as its domain. If the optimum is lower than or equal to the query sphere radius, the query intersects the hyperplane, and both child trees must be investigated. For a one-dimensional space, this problem can be solved by bisection. The running time of the optimization may grow exponentially with the number of dimensions, however.

To address this problem, we have focused only on *approximate* query processing, which is considerably cheaper. To achieve this, we adapted the early termination principle that is used by Liu et al. (2004) and Cayton (2008). The main idea of this principle is that a point close to the nearest neighbor is found quickly and then a large fraction of the execution is spent on backtracking to prove that the point is the actual nearest neighbor.

A nearest neighbor query q is performed by recursively traversing the tree from the root to leaves. The search algorithm exploits geometrical properties of the tree, exploring the regions that contain and/or are closer to the query (best-first). If the visited node is a leaf, a linear scan is performed on this node. Otherwise (i.e., in an internal node), p_i and p_j are reported if they qualify. Then, the algorithm decides which subtree should visit first by evaluating `partition`(p_i, p_j, q). When backtracking, the other subtree is explored. The search stops after a user-defined node count (m) is reached. The pseudocode is sketched in Algorithm 1.

There are many different ways to choose the hyperplanes used in constructing the tree. A simple strategy that works well in practice is described in the following. Two points are selected from the data set at random and the data set is divided into two disjoint subsets by applying the Bregman hyperplane partitioning for each point of the data set. These actions are repeated several times for different hyperplanes, and we choose the hyperplane that gives us the most even split (i.e., the minimum absolute difference in cardinality between the two subsets).

Another contribution of our work is that we apply the hyperplane partitioning principle in bbtree instead of k-means algorithm. Thus, we call the resulting indexing structure as bregman ball tree with hyperplane partitioning (bbhtree) which supports exact nearest neighbor search. The main principle of construction of bbhtree is almost same as the original bbtree. A detailed pseudocode of our algorithm is given in Algorithm 2. Each node of the tree represents a Bregman ball that covers all points in the subtree rooted at that node and the coordinates of the center of this node is computed as the mean of the coordinates of all points in that subset for each dimension. In addition to that, it also has a hyperplane. The tree is built recursively as follows. First we check the size of data set with a given maximum leaf size and the node becomes a leaf if the size below the threshold. Otherwise, the node becomes an internal and we choose a hyperplane for that node. The hyperplane divides the data set that will result two subsets. Then the algorithm builds the left and right subtrees for the resulting subsets.

Nearest neighbor search is done with two filtering phases. The hyperplanes are used as *navigation tools* to decide which child node should be visited first

Algorithm 1 Bregman hyperplane tree

function *Build*($\mathcal{X}$):
  if $|\mathcal{X}| \leq$ *leafsize*:
    return $\mathcal{X}$ ▷ leaf node
  else:
    Select $p_i, p_j \in \mathcal{X}$ ▷ pivot selection
    $\mathcal{X}_l \leftarrow \{x | \texttt{partition}(p_i, p_j, x) = \textit{left}, x \in \mathcal{X} \setminus \{p_i \cup p_j\}\}$
    $\mathcal{X}_r \leftarrow \{x | x \in \mathcal{X} \setminus (\mathcal{X}_l \cup \{p_i \cup p_j\})\}$
    $l \leftarrow \textit{Build}(\mathcal{X}_l)$ ▷ left subtree
    $r \leftarrow \textit{Build}(\mathcal{X}_r)$ ▷ right subtree
    return (p_i, p_j, l, r)

function *Query*(*ans*,*q*,*m*):
  if $m \leq 0$:
    exit ▷ early termination
  if this node is leaf:
    Search using q in the leaf node and update *ans* with results
    $m = m - 1$
  else:
    Compute the Bregman div. between p_i and q update *ans* with p_i if necessary
    Compute the Bregman div. between p_j and q update *ans* with p_j if necessary
    if $\texttt{partition}(p_i, p_j, q) = \textit{left}$:
      $l.\textit{Query}(\textit{ans},q,m)$ ▷ explore the closest branch first
      $r.\textit{Query}(\textit{ans},q,m)$ ▷ then the other one
    else:
      $r.\textit{Query}(\textit{ans},q,m)$
      $l.\textit{Query}(\textit{ans},q,m)$

Algorithm 2 Bregman ball tree with hyperplane partitioning

```
1: function Build(𝒳):
2:     Compute node's center c over 𝒳 and covering radius
3:     if |𝒳| ≤ leafsize:
4:         return 𝒳                                          ▷ leaf node
5:     else:
6:         Select p_i, p_j ∈ 𝒳                               ▷ pivot selection
7:         𝒳_l ← {x|partition(p_i, p_j, x) = left, x ∈ 𝒳}
8:         𝒳_r ← {x|x ∈ 𝒳 \ 𝒳_l}
9:         l ← Build(𝒳_l)                                    ▷ left subtree
10:        r ← Build(𝒳_r)                                    ▷ right subtree
11:        return (c, p_i, p_j, l, r)

12: function Query(ans,q):
13:     if this node is leaf:
14:         Search using q in the leaf node and
              update ans with results
15:     else:
16:         if partition(p_i, p_j, q) = left:
17:             l.Query(ans,q)
                                          ▷ explore the closest branch first
18:             if query ball ∩ r:
19:                 r.Query(ans,q)                            ▷ then the other one
20:         else:
21:             r.Query(ans,q)
22:             if query ball ∩ l:
23:                 l.Query(ans,q)
```

(while the other node may be visited during backtracking) and then the algorithm follows the bbtree's nearest neigbor search algorithm: first the algorithm visits the chosen node from the hyperplane partitioning condition and then checks overlap between the other child's ball and the query ball and visits only if they intersect. In order to check overlap of balls we use the bisection method which is explained in Section 3.

5 Experiments

In this section, we present an experimental evaluation of our trees (bhtree and bbhtree), comparing them to the bbtree (Cayton, 2008). We compare the time required to build the trees, and measure the performance of the trees compared to linear scan as well as the quality of the results, on several data sets. We conducted our experiments on the same, high-dimensional data sets used by Cayton (2008). For all data sets, we use the Kullback-Leibler divergence. In

fact, little attention has been paid to the efficient proximity search for the Kullback-Leibler divergence whereas some other divergences (such as ℓ_2^2 and Mahalanobis) can be handled by the existing metric indexes.

- **SIFT signatures**: 1111-dimensional representations of 12 360 images from the PASCAL 2007 data set (Everingham et al., 2007). Each point is a histogram of quantized SIFT features (Nowak et al., 2006).
- **Semantic space**: 371-dimensional representations of 5000 images from the Corel stock photo collection. Each image is represented as a distribution over 371 keywords (Rasiwasia et al., 2007).
- **Corel histograms**: a collection of 64-dimensional 66 616 color histograms extracted from the Corel image data set.
- **rcv-8**, **rcv-16**, **rcv-32**, **rcv-64** and **rcv-256**: 8, 16, 32, 64 and 256 dimensional representations of topic histograms for 8, 16, 32, 64 and 256 topics, respectively, by using latent Dirichlet allocation (Blei et al., 2003) on 500 000 documents from rcv1 corpus (Lewis et al., 2004).

We used the following number of points from the data sets as queries: 2360, 500, 6616, 1000, 1000, 1000, 1000 and 1000 points from SIFT Signatures, Semantic space, Corel histograms, rcv-8, rcv-16, rcv-32, rcv-64 and rcv-256 respectively.

As a baseline for comparison, we selected the bbtree and used the original implementation of bbtree (Cayton, 2012) which is written in C. Our program is written in C++.[2] All programs were compiled in gcc 4.7.0 with the option -O3. All experiments are performed on a PC with a 3.3 GHz Intel Core i5-2500 processor and 8 GiB RAM. For both trees, the leaf node size was 10 and the node counts (m) were set to 2, 4, 8, 16, 32, 64, 128 and 256 following the bbtree's experimental settings. In all of our experiments, we report the *left* nearest neighbors (NN) (i.e., $d_f(x, q)$, searching for a nearest neighbor $x \in \mathcal{X}$ for a given query q). Thus, the first type Bregman hyperplane partitioning is used in bhtree. We compared the search performance and result quality of the trees by varying the result size thresholds, using the values 1, 5, 10, 20, 40 and 80. We report only the results with NN and 10 NN because the results with the other result size thresholds were quite similar. We measured the wall clock time that is required to build the trees. The build time and speedup of bhtree over bbtree are shown in Table 2. The high build time of the bbtree is likely due to the fact that the k-means algorithm is employed as a space partitioning method. In addition, the speedup values are noticeably high when the dimensionalities of the data sets are high.

[2] A C++ implementation will, most likely, entail some slight overhead, but this would not bias the results in our favor.

	SIFT signatures	Semantic space	Corel hist.	rcv-8	rcv-256
bhtree	4.5 s	0.6 s	2.5 s	8.7 s	78.8 s
bbtree	131.9 s	19.2 s	54.2 s	80.4 s	2303.4 s
speedup	29.3×	32.0×	21.6×	9.2×	29.2×

Table 2: Build time of trees and speedup of bhtree over bbtree.

For each query, we counted the number of distance computations needed for the approximate search, normalized by the size of the data set, as the performance measure, and measured a slightly modified version[3] of the error on the position (Zezula et al., 2006) as the result quality measure. This modified version of the error on the position yields the average absolute position difference between every point of the exact and approximate results. In general, approximation techniques will produce results that vary with with their parameter settings. In order to make a fair comparison between different techniques we have to compare their speed-up factors with the same error or vice versa. In some cases, it would be difficult to achieve this goal. In order to compare the results properly, they are plotted as a lines with one point for each parameter setting, with the coordinates for each point given by the mean error and mean normalized distance count for all queries. On the y-axis, the value 10^{-2} means that the indexing structure performed 100 times as fast as a linear scan. On the x-axis, the value 10^1 means that the average absolute position difference between exact and approximate result is 10 (for instance, if the result size threshold is 1, then the 11th NN is reported instead of the NN).

In Figure 3, the errors on the position vs. normalized distance counts for NN and 10 NN on SIFT signatures are shown. It can be clearly seen that the result quality is a non-decreasing function of the number of scanned nodes.

Consider the results for NN: when the error on the position for the bhtree is 26 with normalized distance count 0.0083 (i.e., used only 0.83 % of the data set), the error for the bbtree is 35 with normalized distance count 0.297 (i.e., used 29.7 % of the data set). In general, the bhtree is about 4.6–35.6 times as fast as the bbtree for the same level of error. Moreover, on this data set the bbtree takes 29.3 times as long to build as the bhtree.

Figure 4 shows the results on Semantic space, Corel histograms, rcv-8 and rcv-256 data sets. The trees have roughly the same speedups and result qualities on Semantic space and Corel histograms. The bhtree outperforms the bbtree on rcv-8. As the dimensionality of rcv1 data set increases, the bhtree is about 100–1000 times as fast as linear scan with a low degree of error. The bbtree is about 1.5–3 times as fast as the bhtree for the same error, but this still yields

[3]We normalized the error by the exact result's size instead of the approximate result's size, and in addition to this we did not normalize the error by the data set's size.

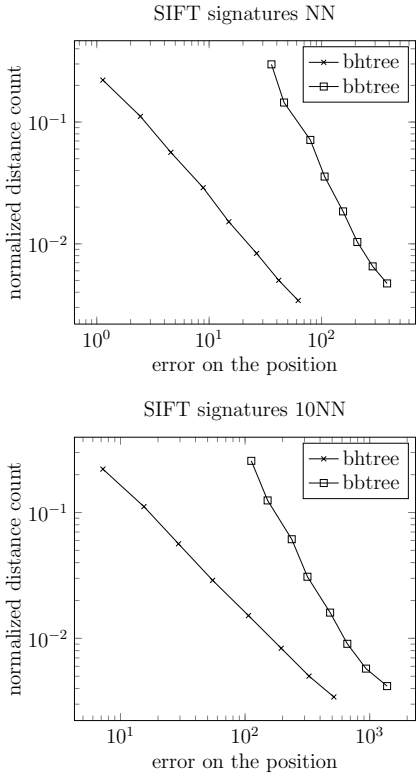

Figure 3: Performance vs. result quality of approximation on SIFT signatures.

a good tradeoff between construction time and search speedup, as the bbtree requires 29.2 times as much build time as the bhtree on this data set.

The bhtree achieved good results with very high-dimensional SIFT signatures. For both trees, we also notice that as the result size threshold increases, the result quality becomes worse on all of our data sets; this is probably due to the fact that their dynamic search radii intersect with many balls/hyperspaces.

So far, we have paid attention to approximate nearest neighbor retrieval. Let us have a look at the exact nearest neighbor search, in particular the performance of bbhtree. The build time and the results for exact NN on various data sets are given in Table 3. The results show that the bbhtrees are relatively cheaper to build than the bbtrees.

For rcv-8 and rcv-16, the bbtree slightly (less than 1 %) outperforms bhtree. However, the bbtree requires at least 7.92 times as much build time as the bbhtree on all of the experiments. As the dimensionality increases, the performance of the bbtree and bbhtree degrades, especially on Corel, rcv-32 and

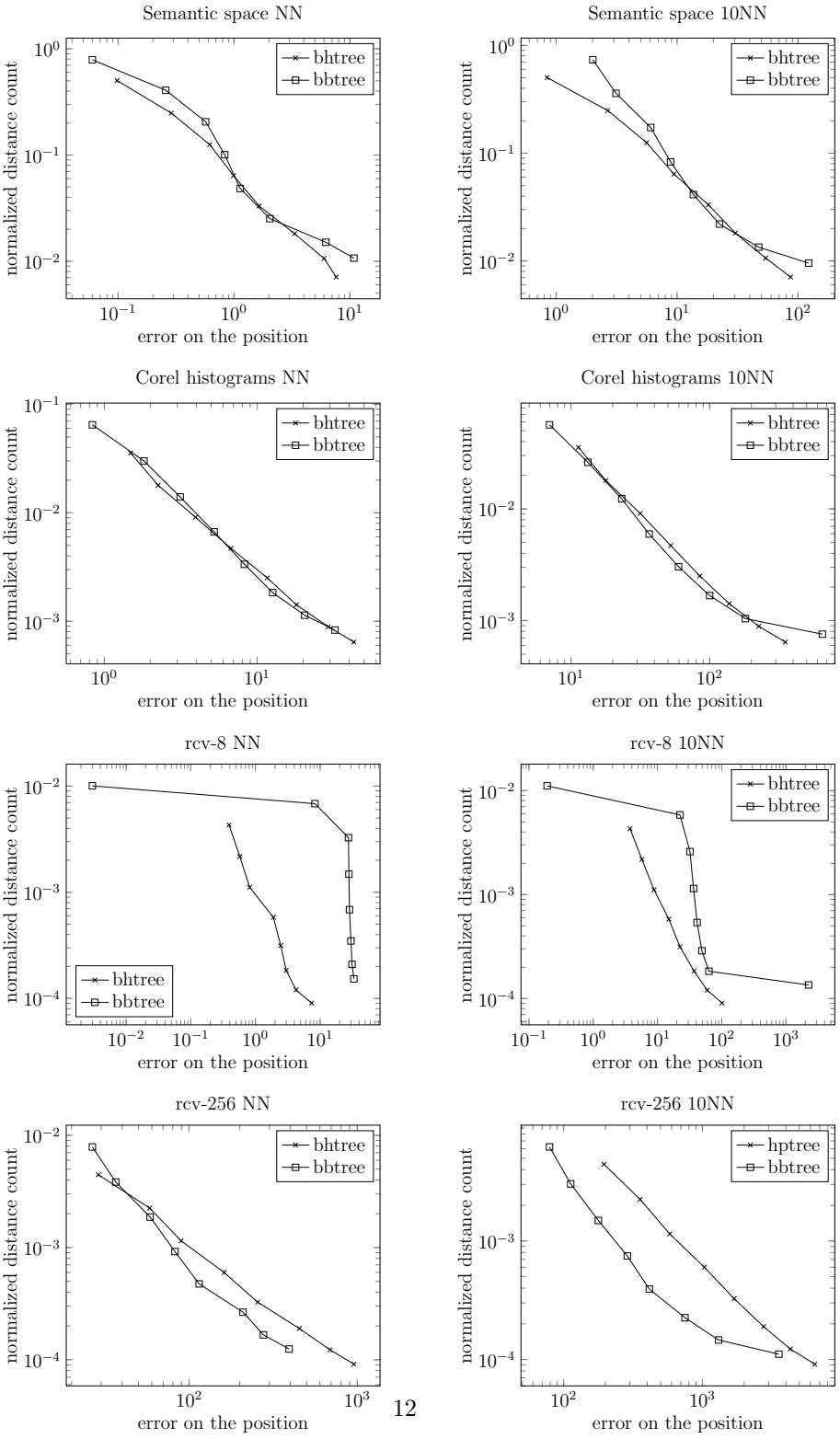

Figure 4: Performance vs. result quality of approximation on Semantic space, Corel histograms, rcv-8 and rcv-256.

	Build time (s)			Data set scanned (%)		
	bbtree	bbhtree	ratio	bbtree	bbhtree	ratio
Corel	54.21	6.30	8.60	34.58	38.47	0.90
rcv-8	84.41	10.66	7.92	1.05	1.17	0.90
rcv-16	155.60	18.51	8.41	3.54	4.48	0.79
rcv-32	285.95	33.04	8.65	9.91	13.47	0.74
rcv-64	557.41	60.48	9.22	27.52	34.82	0.79

Table 3: Build time, percentage of data set scanned of trees and their ratio between bbtree and bbhtree.

rcv-64. It is well-known fact that exact search fails in high-dimensional spaces. The both trees performed worser than a linear scan on rcv-256, SIFT signatures and Semantic space.

6 Conclusion

We have proposed a new indexing structure, the Bregman hyperplane tree, for approximate Bregman proximity search, and a new indexing structure, the Bregman ball tree with hyperplane partitioning, for exact Bregman proximity search. Our experimental results show that our approximate indexing structure is comparable to the Bregman ball tree in terms of result quality and both of our indexing structures require significantly less construction time. One of the important features of the Bregman hyperplane tree is that it is simple to implement.

Acknowledgements

We wish to thank Øystein Torbjørnsen and Svein Erik Bratsberg for helpful discussions and Lawrence Cayton for providing data sets.

References

Blei, D. M., Ng, A. Y., Jordan, M. I., and Lafferty, J. (2003). Latent dirichlet allocation. *Journal of Machine Learning Research*, 3.

Bregman, L. M. (1967). The relaxation method of finding the common points of convex sets and its application to the solution of problems in convex programming. *USSR Computational Mathematics and Mathematical Physics*, 7(3):200–217.

Cayton, L. (2008). Fast nearest neighbor retrieval for bregman divergences. In *Proceedings of the 25th international conference on Machine learning*, ICML '08, pages 112–119.

Cayton, L. (2012). Implementation of bregman ball tree. Downloaded on May 4th, 2012 from `http://lcayton.com/code.html`.

Everingham, M., Van Gool, L., Williams, C. K. I., Winn, J., and Zisserman, A. (2007). The PASCAL Visual Object Classes Challenge 2007 Results.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. In *International Conference on Management of Data*, pages 47–57.

Itakura, F. and Saito, S. (1970). A statistical method for estimation of speech spectral density and formant frequencies. *Electr. and Commun. in Japan*, 53:36–43.

Lewis, D. D., Yang, Y., Rose, T. G., and Li, F. (2004). Rcv1: A new benchmark collection for text categorization research. *Journal of Machine Learning Research*, 5.

Liu, T., Moore, A. W., Gray, E., and Yang, K. (2004). An investigation of practical approximate nearest neighbor algorithms. In *NIPS*.

Navarro, G. (2002). Searching in metric spaces by spatial approximation. *The VLDB Journal*, 11(1):28–46.

Nielsen, F., Boissonnat, J.-D., and Nock, R. (2007). On bregman voronoi diagrams. In *Proceedings of the 18th ACM-SIAM symposium on Discrete algorithms*, SODA '07, pages 746–755.

Nowak, E., Jurie, F., and Triggs, B. (2006). Sampling strategies for bag-of-features image classification. In *European Conference on Computer Vision*.

Rasiwasia, N., Member, S., Moreno, P. J., and Vasconcelos, N. (2007). Bridging the gap: Query by semantic example. *IEEE Trans. Multimedia*, 9.

Rubner, Y., Puzicha, J., Tomasi, C., and Buhmann, J. M. (1999). Empirical evaluation of dissimilarity measures for color and texture. In *ICCV*, pages 1165–1173.

Samet, H. (2005). *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*.

Uhlmann, J. K. (1991). Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179.

Weber, R., Schek, H.-J., and Blott, S. (1998). A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of VLDB'98*, pages 194–205.

Zezula, P., Amato, G., Dohnal, V., and Batko, M. (2006). *Similarity Search : The Metric Space Approach*. Springer.

Zhang, Z., Chin, B., Srinivasan, O., Anthony, P., and Tung, K. H. (2009). Similarity search on bregman divergence: Towards non-metric indexing. In *VLDB*.