LARGE-SCALE MUSIC SIMILARITY SEARCH WITH SPATIAL TREES

Brian McFee
Computer Science and Engineering
University of California, San Diego

Gert Lanckriet
Electrical and Computer Engineering
University of California, San Diego

ABSTRACT

Many music information retrieval tasks require finding the nearest neighbors of a query item in a high-dimensional space. However, the complexity of computing nearest neighbors grows linearly with size of the database, making exact retrieval impractical for large databases. We investigate modern variants of the classical KD-tree algorithm, which efficiently index high-dimensional data by recursive spatial partitioning. Experiments on the Million Song Dataset demonstrate that content-based similarity search can be significantly accelerated by the use of spatial partitioning structures.

1. INTRODUCTION

Nearest neighbor computations lie at the heart of many content-based approaches to music information retrieval problems, such as playlist generation [4, 14], classification and annotation [12, 18] and recommendation [15]. Typically, each item (*e.g.*, song, clip, or artist) is represented as a point in some high-dimensional space, *e.g.*, $\mathbb{R}^d$ equipped with Euclidean distance or Gaussian mixture models equipped with Kullback-Leibler divergence.

For large music databases, nearest neighbor techniques face an obvious limitation: computing the distance from a query point to each element of the database becomes prohibitively expensive. However, for many tasks, *approximate* nearest neighbors may suffice. This observation has motivated the development of general-purpose data structures which exploit metric structure to locate neighbors of a query in sub-linear time [1, 9, 10].

In this work, we investigate the efficiency and accuracy of several modern variants of KD-trees [1] for answering nearest neighbor queries for musical content. As we will demonstrate, these *spatial trees* are simple to construct, and can provide substantial improvements in retrieval time while maintaining satisfactory performance.

2. RELATED WORK

Content-based similarity search has received a considerable amount of attention in recent years, but due to the obvious data collection barriers, relatively little of it has focused on retrieval in large-scale collections.

Cai, et al. [4] developed an efficient query-by-example audio retrieval system by applying locality sensitive hashing (LSH) [9] to a vector space model of audio content. Although LSH provides strong theoretical guarantees on retrieval performance in sub-linear time, realizing those guarantees in practice can be challenging. Several parameters must be carefully tuned — the number of bins in each hash, the number of hashes, the ratio of *near* and *far* distances, and collision probabilities — and the resulting index structure can become quite large due to the multiple hashing of each data point. Cai, et al.'s implementation scales to upwards of 10^5 audio clips, but since their focus was on playlist generation, they did not report the accuracy of nearest neighbor recall.

Schnitzer, et al. developed a filter-and-refine system to quickly approximate the Kullback-Leibler (KL) divergence between timbre models [17]. Each song was summarized by a multivariate Gaussian distribution over MFCC vectors, and then mapped into a low-dimensional Euclidean vector space via the FastMap algorithm [10], so that Euclidean distance approximates the symmetrized KL divergence between song models. To retrieve nearest neighbors for a query song, the approximate distances are computed from the query to each point in the database by a linear scan (the *filter* step). The closest points are then *refined* by computing the full KL divergence to the query. This approach exploits the fact that low-dimensional Euclidean distances are much cheaper to compute than KL-divergence, and depending on the size of the filter set, can produce highly accurate results. However, since the filter step computes distance to the entire database, it requires $O(n)$ work, and performance may degrade if the database is too large to fit in memory.

3. SPATIAL TREES

Spatial trees are a family of data structures which recursively bisect a data set $\mathcal{X} \subset \mathbb{R}^d$ of n points in order to facilitate efficient (approximate) nearest neighbor retrieval [1, 19]. The recursive partitioning of $\mathcal{X}$ results in a binary tree, where

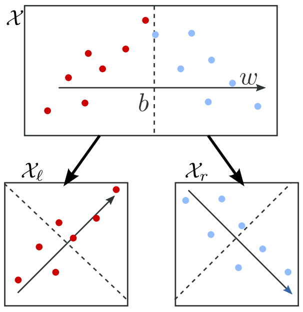

Figure 1. Spatial partition trees recursively split a data set $\mathcal{X} \subset \mathbb{R}^d$ by projecting onto a direction $w \in \mathbb{R}^d$ and splitting at the median b (dashed line), forming two disjoint subsets $\mathcal{X}_\ell$ and $\mathcal{X}_r$.

Algorithm 1 Spatial partition tree

```
Input: data $\mathcal{X} \subset \mathbb{R}^d$, maximum tree depth $\delta$
Output: balanced binary tree $t$ over $\mathcal{X}$
    PARTITION($\mathcal{X}, \delta$)
 1: if $\delta = 0$ then
 2:     return $\mathcal{X}$ (leaf set)
 3: else
 4:     $w_t \leftarrow \mathrm{split}(\mathcal{X})$ {find a split direction}
 5:     $b_t \leftarrow \mathrm{median}\left(\left\{w_t^\top x \mid x \in \mathcal{X}\right\}\right)$
 6:     $\mathcal{X}_\ell \leftarrow \left\{x \mid w_t^\top x \leq b_t,\ x \in \mathcal{X}\right\}$
 7:     $\mathcal{X}_r \leftarrow \left\{x \mid w_t^\top x > b_t,\ x \in \mathcal{X}\right\}$
 8:     $t_\ell \leftarrow$ PARTITION($\mathcal{X}_\ell, \delta - 1$)
 9:     $t_r \leftarrow$ PARTITION($\mathcal{X}_r, \delta - 1$)
10:     return $t = (w_t, b_t, t_\ell, t_r)$
```

each node t corresponds to a subset of the data $\mathcal{X}_t \subseteq \mathcal{X}$ (Figure 1). At the root of the tree lies the entire set $\mathcal{X}$, and each node t defines a subset of its parent.

A generic algorithm to construct partition trees is listed as Algorithm 1. The set $\mathcal{X} \subset \mathbb{R}^d$ is projected onto a direction $w_t \in \mathbb{R}^d$, and split at the median b_t into subsets $\mathcal{X}_\ell$ and $\mathcal{X}_r$: splitting at the median ensures that the tree remains balanced. This process is then repeated recursively on each subset, until a specified tree depth δ is reached.

Spatial trees offer several appealing properties. They are simple to implement, and require minimal parameter-tuning: specifically, only the maximum tree depth δ, and the rule for generating split directions. Moreover, they are efficient to construct and use for retrieval. While originally developed for use in metric spaces, the framework has been recently extended to support general Bregman divergences (including, *e.g.*, KL-divergence) [5]. However, for the remainder of this article, we will focus on building trees for vector space models ($\mathbb{R}^d$ with Euclidean distance).

In order for Algorithm 1 to be fully specified, we must provide a function $\mathrm{split}(\mathcal{X})$ which determines the split direction w. Several splitting rules have been proposed in the literature, and our experiments will cover the four described by Verma, et al. [20]: maximum variance KD, principal direction (PCA), 2-means, and random projection.

3.1 Maximum variance KD-tree

The standard KD-tree (k-dimensional tree) chooses w by cycling through the standard basis vectors $\mathbf{e}_i$ ($i \in \{1, 2, \ldots, d\}$), so that at level j in the tree, the split direction is $w = \mathbf{e}_{i+1}$ with $i = j \mod d$ [1]. The standard KD-tree can be effective for low-dimensional data, but it is known to perform poorly in high dimensions [16, 20]. Note also that if $n < 2^d$, there will not be enough data to split along every coordinate, so some (possibly informative) features may never be used by the data structure.

A common fix to this problem is to choose w as the coordinate which maximally spreads the data [20]:

$$\mathrm{split}_{\mathrm{KD}}(\mathcal{X}) = \operatorname*{argmax}_{\mathbf{e}_i} \sum_{x \in \mathcal{X}} \left(\mathbf{e}_i^\top (x - \mu)\right)^2, \qquad (1)$$

where μ is the sample mean vector of $\mathcal{X}$. Intuitively, this split rule picks the coordinate which provides the greatest reduction in variance (increase in concentration).

The maximum variance coordinate can be computed with a single pass over $\mathcal{X}$ by maintaining a running estimate of the mean vector and coordinate-wise variance, so the complexity of computing $\mathrm{split}_{\mathrm{KD}}(\mathcal{X})$ is $O(dn)$.

3.2 PCA-tree

The KD split rule (Eqn. (1)) is limited to axis-parallel directions w. The principal direction (or principal components analysis, PCA) rule generalizes this to choose the direction $w \in \mathbb{R}^d$ which maximizes the variance, *i.e.*, the leading eigenvector v of the sample covariance matrix $\widehat{\Sigma}$:

$$\mathrm{split}_{\mathrm{PCA}}(\mathcal{X}) = \operatorname*{argmax}_{v} v^\top \widehat{\Sigma} v \quad \text{s. t. } \|v\|_2 = 1. \qquad (2)$$

By using the full covariance matrix to choose the split direction, the PCA rule may be more effective than KD-tree at reducing the variance at each split in the tree.

$\widehat{\Sigma}$ can be estimated from a single pass over $\mathcal{X}$, so (assuming $n > d$) the time complexity of $\mathrm{split}_{\mathrm{PCA}}$ is $O(d^2 n)$.

3.3 2-means

Unlike the KD and PCA rules, which try to maximally reduce variance with each split, the 2-means rule produces splits which attempt preserve cluster structure. This is accomplished by running the k-means algorithm on $\mathcal{X}$ with $k = 2$, and defining w to be the direction spanned by the cluster centroids $c_1, c_2 \in \mathbb{R}^d$:

$$\mathrm{split}_{\mathrm{2M}}(\mathcal{X}) = c_1 - c_2. \qquad (3)$$

While this general strategy performs well in practice [13], it can be costly to compute a full k-means solution. In our experiments, we instead use an online k-means variant which runs in $O(dn)$ time [3].

3.4 Random projection

The final splitting rule we will consider is to simply take a direction uniformly at random from the unit sphere $\mathbb{S}^{d-1}$:

$$\text{split}_{\text{RP}}(\mathcal{X}) \sim_U \mathbb{S}^{d-1}, \tag{4}$$

which can equivalently be computed by normalizing a sample from the multivariate Gaussian distribution $\mathcal{N}(0, I_d)$. The random projection rule is simple to compute and adapts to the intrinsic dimensionality of the data $\mathcal{X}$ [8].

In practice, the performance of random projection trees can be improved by independently sampling m directions $w_i \sim \mathcal{S}^{d-1}$, and returning the w_i which maximizes the decrease in data diameter after splitting [20]. Since a full diameter computation would take $O(dn^2)$ time, we instead return the direction which maximizes the *projected diameter*:

$$\underset{w_i}{\text{argmax}} \max_{x_1, x_2 \in \mathcal{X}} w_i^\mathsf{T} x_1 - w_i^\mathsf{T} x_2. \tag{5}$$

This can be computed in a single pass over $\mathcal{X}$ by tracking the maximum and minimum of $w_i^\mathsf{T} x$ in parallel for all w_i, so the time complexity of split_{RP} is $O(mdn)$. Typically, $m \leq d$, so split_{RP} is comparable in complexity to $\text{split}_{\text{PCA}}$.

3.5 Spill trees

The main drawback of partition trees is that points near the decision boundary become isolated from their neighbors across the partition. Because data concentrates near the mean after (random) projection [8], hard partitioning can have detrimental effects on nearest neighbor recall for a large percentage of queries.

Spill trees remedy this problem by allowing overlap between the left and right subtrees [13]. If a point lies close to the median, then it will be added to both subtrees, thus reducing the chance that it becomes isolated from its neighbors (Figure 2). This is accomplished by maintaining two decision boundaries b_t^ℓ and b_t^r. If $w_t^\mathsf{T} x > b_t^r$, then x is added to the right tree, and if $w_t^\mathsf{T} x \leq b_t^\ell$, it is added to the left. The gap between b_t^ℓ and b_t^r controls the amount of data which *spills* across the split.

The algorithm to construct a spill tree is listed as Algorithm 2. The algorithm requires a spill threshold $\tau \in [0, 1/2)$: rather than splitting at the median (so that a set of n items is split into subsets of size roughly $n/2$), the data is split at the $(1/2 + \tau)$-quantile, so that each subset has size roughly $n(1/2 + \tau)$. Note that when $\tau = 0$, the thresholds coincide ($b_t^\ell = b_t^r$), and the algorithm simplifies to Algorithm 1. Partition trees, therefore, correspond to the special case of $\tau = 0$.

3.6 Retrieval algorithm and analysis

Once a spill tree has been constructed, approximate nearest neighbors can be recovered efficiently by the *defeatist search* method [13], which restricts the search to only the leaf sets which contain the query. For a novel query $q \in \mathbb{R}^d$ (*i.e.*, a previously unseen point), these sets can be found by Algorithm 3.

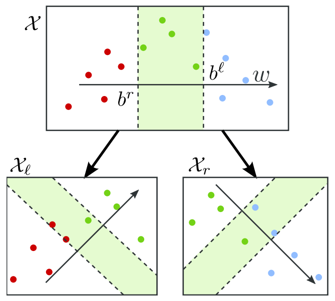

Figure 2. Spill trees recursively split data like partition trees, but the subsets are allowed to overlap. Points in the shaded region are propagated to both subtrees.

Algorithm 2 Spill tree

```
Input: data $\mathcal{X} \subset \mathbb{R}^d$, depth $\delta$, threshold $\tau \in [0, 1/2)$
Output: $\tau$-spill tree $t$ over $\mathcal{X}$
    SPILL($\mathcal{X}, \delta, \tau$)
 1: if $\delta = 0$ then
 2:     return $\mathcal{X}$ (leaf set)
 3: else
 4:     $w_t \leftarrow \text{split}(\mathcal{X})$
 5:     $b_t^\ell \leftarrow \text{quantile}\left(1/2 + \tau, \{w_t^\mathsf{T} x \mid x \in \mathcal{X}\}\right)$
 6:     $b_t^r \leftarrow \text{quantile}\left(1/2 - \tau, \{w_t^\mathsf{T} x \mid x \in \mathcal{X}\}\right)$
 7:     $\mathcal{X}_\ell \leftarrow \{x \mid w_t^\mathsf{T} x \leq b_t^\ell, x \in \mathcal{X}\}$
 8:     $\mathcal{X}_r \leftarrow \{x \mid w_t^\mathsf{T} x > b_t^r, x \in \mathcal{X}\}$
 9:     $t_\ell \leftarrow$ SPILL($\mathcal{X}_\ell, \delta - 1, \tau$)
10:     $t_r \leftarrow$ SPILL($\mathcal{X}_r, \delta - 1, \tau$)
11:     return $t = (w_t, b_t^\ell, b_t^r, t_\ell, t_r)$
```

The total time required to retrieve k neighbors for a novel query q can be computed as follows. First, note that for a spill tree with threshold τ, each split reduces the size of the set by a factor of $(1/2 + \tau)$, so the leaf sets of a depth-δ tree are exponentially small in δ: $n(1/2 + \tau)^\delta$. Note that $\delta \leq \log n$, and is typically chosen so that the leaf set size lies in some reasonable range (*e.g.*, between 100 and 1000 items).

Next, observe that in general, Algorithm 3 may map the query q to some h distinct leaves, so the total size of the retrieval set is at most $n' = hn(1/2 + \tau)^\delta$ (although it may be considerably smaller if the sets overlap). For h leaves, there are at most h paths of length δ to the root of the tree, and each step requires $O(d)$ work to compute $w_t^\mathsf{T} q$, so the total time taken by Algorithm 3 is

$$\mathcal{T}_{\text{RETRIEVE}} \in O\left(h\left(d\delta + n(1/2 + \tau)^\delta\right)\right).$$

Finally, once the retrieval set has been constructed, the k closest points can be found in time $O(dn' \log k)$ by using a k-bounded priority queue [7]. The total time to retrieve k

Algorithm 3 Spill tree retrieval

```
Input: query $q$, tree $t$
Output: Retrieval set $\mathcal{X}_q$
  RETRIEVE$(q, t)$
1: if $t$ is a leaf then
2:   return $\mathcal{X}_t$ {all items contained in the leaf}
3: else
4:   $\mathcal{X}_q \leftarrow \emptyset$
5:   if $w_t^\mathsf{T} q \leq b_t^\ell$ then
6:     $\mathcal{X}_q \leftarrow \mathcal{X}_q \cup$ RETRIEVE$(q, t_\ell)$
7:   if $w_t^\mathsf{T} q > b_t^r$ then
8:     $\mathcal{X}_q \leftarrow \mathcal{X}_q \cup$ RETRIEVE$(q, t_r)$
9:   return $\mathcal{X}_q$
```

approximate nearest neighbors for the query q is therefore

$$\mathcal{T}_{k\text{NN}} \in O\big(hd\left(\delta + n(1/2 + \tau)^\delta \log k\right)\big) .$$

Intuitively, for larger values of τ, more data is spread throughout the tree, so the leaf sets become larger and retrieval becomes slower. Similarly, larger values of τ will result in larger values of h as queries will map to more leaves. However, as we will show experimentally, this effect is generally mild even for relatively large values of τ.

In the special case of partition trees ($\tau = 0$), each query maps to exactly $h = 1$ leaf, so the retrieval time simplifies to $O(d(\delta + n/2^\delta \log k))$.

4. EXPERIMENTS

Our song data was taken from the Million Song Dataset (MSD) [2]. Before describing the tree evaluation experiments, we will briefly summarize the process of constructing the underlying acoustic feature representation.

4.1 Audio representation

The audio content representation was developed on the 1% Million Song Subset (MSS), and is similar to the model proposed in [15]. From each MSS song, we extracted the time series of Echo Nest timbre descriptors (ENTs). This results in a sample of approximately 8.5 million 12-dimensional ENTs, which were normalized by z-scoring according to the estimated mean and variance of the sample, randomly permuted, and then clustered by online k-means to yield 512 acoustic codewords. Each song was summarized by quantizing each of its (normalized) ENTs and counting the frequency of each codeword, resulting in a 512-dimensional histogram vector. Each codeword histogram was mapped into a probability product kernel (PPK) space [11] by square-rooting its entries, which has been demonstrated to be effective on similar audio representations [15]. Finally, we appended the song's tempo, loudness, and key confidence, resulting in a vector $v_i \in \mathbb{R}^{515}$ for each song x_i.

Next, we trained an optimized similarity metric over audio descriptors. First, we computed target similarity for each pair of MSS artists by the Jaccard index between their user sets in a sample of Last.fm [1] collaborative filter data [6, chapter 3]. Tracks by artists with fewer than 30 listeners were discarded. Next, all remaining artists were partitioned into a training (80%) and validation (20%) set, and for each artist, we computed its top 10 most similar training artists.

Having constructed a training and validation set, the distance metric was optimized by applying the metric learning to rank (MLR) algorithm on the training set of 4455 songs, and tuning parameters $C \in \{10^5, 10^6, \ldots, 10^9\}$ and $\Delta \in$ {AUC, MRR, MAP, Prec@10} to maximize AUC score on the validation set of 1110 songs. Finally, the resulting metric W was factored by PCA (retaining 95% spectral mass) to yield a linear projection $L \in \mathbb{R}^{222\times515}$.

The projection matrix L was then applied to each v_i in MSD. As a result, each MSD song was mapped into $\mathbb{R}^{222}$ such that Euclidean distance is optimized by MLR to retrieve songs by similar artists.

4.2 Representation evaluation

To verify that the optimized vector quantization (VQ) song representation carries musically relevant information, we performed a small-scale experiment to evaluate its predictive power for semantic annotation. We randomly selected one song from each of 4643 distinct artists. (Artists were restricted to be disjoint from MSS to avoid contamination.) Each song was represented by the optimized 222-dimensional VQ representation, and as ground truth annotations, we applied the corresponding artist's terms from the *top-300 terms* provided with MSD, so that each song x_i has a binary annotation vector $y_i \in \{0, 1\}^{300}$. For a baseline comparison, we adapt the representation used by Schnitzer, et al. [17], and for each song, we fit a full-covariance Gaussian distribution over its ENT features.

The set was then randomly split 10 times into 80%-training and 20%-test sets. Following the procedure described by Kim, et al. [12], each test song was annotated by thresholding the average annotation vector of its k nearest training neighbors as determined by Euclidean distance on VQ representations, and by KL-divergence on Gaussians. Varying the decision threshold yields a trade-off between precision and recall. In our experiments, the threshold was varied between 0.1 and 0.9.

Figure 3 displays the precision-recall curves averaged across all 300 terms and training/test splits for several values of k. At small values of k, the VQ representation achieves significantly higher performance than the Gaussian representation. We note that this evaluation is by no means conclusive, and is merely meant to demonstrate that the underlying space is musically relevant.

[1] `http://last.fm`

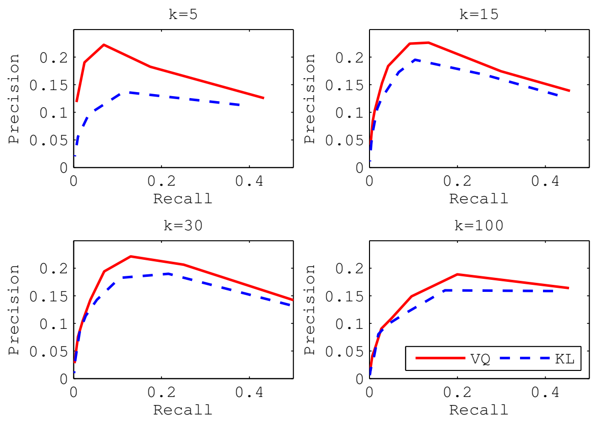

Figure 3. Mean precision-recall for k-nearest neighbor annotation with VQ and Gaussian (KL) representations.

4.3 Tree evaluation

To test the accuracy of the different spatial tree algorithms, we partitioned the MSD data into 890205 training songs $\mathcal{X}$ and 109795 test songs $\mathcal{X}'$. Using the optimized VQ representations on $\mathcal{X}$, we constructed trees with each of the four splitting rules (PCA, KD, 2-means, and random projection), varying both the maximum depth $\delta \in \{5, 6, \ldots, 13\}$ and spill threshold $\tau \in \{0, 0.01, 0.05, 0.10\}$. At $\delta = 13$, this results in leaf sets of size 109 with $\tau = 0$, and 1163 for $\tau = 0.10$. For random projection trees, we sample $m = 64$ dimensions at each call to split_{RP}.

For each test song $q \in \mathcal{X}'$, and tree t, we compute the retrieval set with Algorithm 3. The *recall* for q is the fraction of the true nearest-neighbors $k\text{NN}(q)$ contained in the retrieval set:

$$R(q, t) = |\text{RETRIEVE}(q, t) \cap k\text{NN}(q)| / k. \qquad (6)$$

Note that since true nearest neighbors are always closer than any other points, they are always ranked first, so precision and recall are equivalent here.

To evaluate the system, $k = 100$ exact nearest neighbors $k\text{NN}(q)$ were found from $\mathcal{X}$ for each query $q \in \mathcal{X}'$ by a full linear search over $\mathcal{X}$.

4.4 Retrieval results

Figure 4 lists the nearest-neighbor recall performance for all tree configurations. As should be expected, for all splitting rules and spill thresholds, recall performance degrades as the maximum depth of the tree increases.

Across all spill thresholds τ and tree depths δ, the relative ordering of performance of the different split rules is essentially constant: $\text{split}_{\text{PCA}}$ performs slightly better than split_{KD}, and both dramatically outperform split_{RP} and split_{2M}. This indicates that for the feature representation under consideration here (optimized codeword histograms), variance reduction seems to be the most effective strategy for preserving nearest neighbors in spatial trees.

For small values of τ, recall performance is generally poor for all split rules. However, as τ increases, recall performance increases across the board. The improvements are most dramatic for $\text{split}_{\text{PCA}}$. With $\tau = 0$, and $\delta = 7$, the PCA partition tree has leaf sets of size 6955 (0.8% of $\mathcal{X}$), and achieves median recall of 0.24. With $\tau = 0.10$ and $\delta = 13$, the PCA spill tree achieves median recall of 0.53 with a comparable median retrieval set size of 6819 (0.7% of $\mathcal{X}$): in short, recall is nearly doubled with no appreciable computational overhead. So, by looking at less than 1% of the database, the PCA spill tree is able to recover more than half of the 100 true nearest neighbors for novel test songs. This contrasts with the filter-and-refine approach [17], which requires a full scan of the entire database.

4.5 Timing results

Finally, we evaluated the retrieval time necessary to answer k-nearest neighbor queries with spill trees. We assume that all songs have already been inserted into the tree, since this is the typical case for long-term usage. As a result, the retrieval algorithm can be accelerated by maintaining indices mapping songs to leaf sets (and vice versa).

We evaluated the retrieval time for PCA spill trees of depth $\delta = 13$ and threshold $\tau \in \{0.05, 0.10\}$, since they exhibit practically useful retrieval accuracy. We randomly selected 1000 test songs and inserted them into the tree prior to evaluation. For each test song, we compute the time necessary to retrieve the k nearest training neighbors from the spill tree (ignoring test songs), for $k \in \{10, 50, 100\}$. Finally, for comparison purposes, we measured the time to compute the true k nearest neighbors by a linear search over the entire training set.

Our implementation is written in Python/NumPy, [2] and loads the entire data set into memory. The test machine has two 1.6GHz Intel Xeon CPUs and 4GB of RAM. Timing results were collected through the *cProfile* utility.

Figure 5 lists the average retrieval time for each algorithm. The times are relatively constant with respect to k: a full linear scan typically takes approximately 2.4 seconds, while the $\tau = 0.10$ spill tree takes less than 0.14 seconds, and the $\tau = 0.05$ tree takes less than 0.02 seconds. In relative terms, setting $\tau = 0.10$ yields a speedup factor of 17.8, and $\tau = 0.05$ yields a speedup of 119.5 over the full scan. The difference in speedup from $\tau = 0.10$ to $\tau = 0.05$ can be explained by the fact that smaller overlapping regions result in smaller (and fewer) leaf sets for each query. In practice, this speed-accuracy trade-off can be optimized for the particular task at hand: applications requiring only a few neighbors which may be consumed rapidly (*e.g.*, sequential playlist generation) may benefit from small values of τ, whereas

[2] `http://numpy.scipy.org`

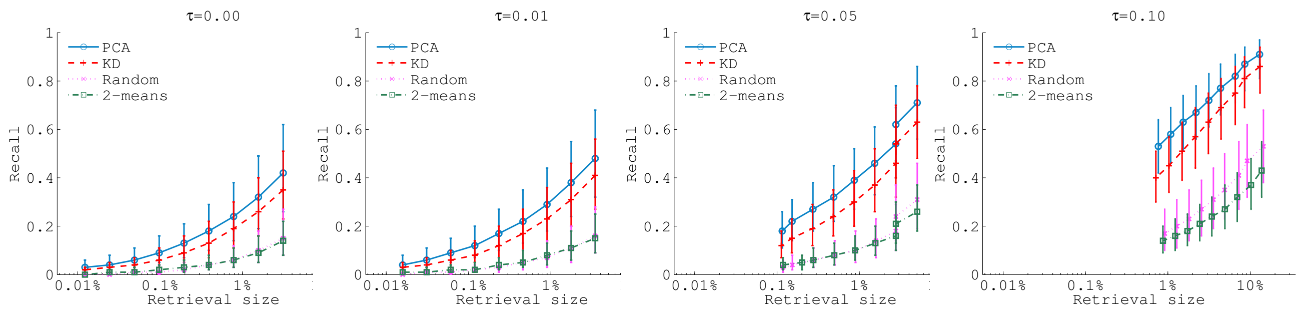

Figure 4. Median 100-nearest-neighbor recall for each splitting rule (PCA, KD, 2-means, and random projection), spill threshold $\tau \in \{0, 0.01, 0.05, 0.10\}$, and tree depth $\delta \in \{5, 6, \ldots, 13\}$. Each point along a curve corresponds to a different tree depth δ, with larger retrieval size indicating smaller δ. For each δ, the corresponding recall point is plotted at the median size of the retrieval set (as a fraction of the entire database). Error bars correspond to 25th and 75th percentiles of recall for all test queries.

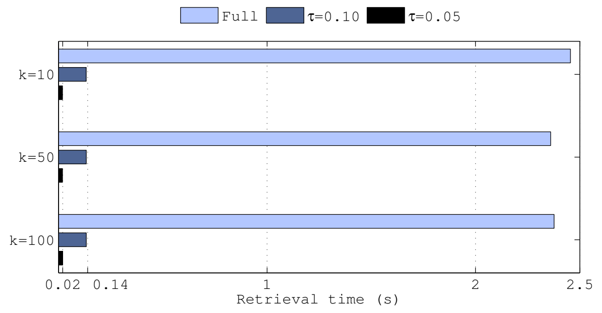

Figure 5. Average time to retrieve k (approximate) nearest neighbors with a full scan versus PCA spill trees.

applications requiring more neighbors (*e.g.*, browsing recommendations for discovery) may benefit from larger τ.

5. CONCLUSION

We have demonstrated that spatial trees can effectively accelerate approximate nearest neighbor retrieval. In particular, for VQ audio representations, the combination of spill trees with and PCA splits yields a favorable trade-off between accuracy and complexity of k-nearest neighbor retrieval.

6. ACKNOWLEDGMENTS

The authors thank Jason Samarin, and acknowledge support from Qualcomm, Inc., Yahoo! Inc., the Hellman Fellowship Program, and NSF Grants CCF-0830535 and IIS-1054960.

7. REFERENCES

[1] J.L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18:509–517, Sep. 1975.

[2] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, and Paul Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[3] Léon Bottou and Yoshua Bengio. Convergence properties of the kmeans algorithm. In *Advances in Neural Information Processing Systems*, volume 7. MIT Press, Denver, 1995.

[4] Rui Cai, Chao Zhang, Lei Zhang, and Wei-Ying Ma. Scalable music recommendation by search. In *International Conference on Multimedia*, pages 1065–1074, 2007.

[5] Lawrence Cayton. Fast nearest neighbor retrieval for bregman divergences. In *International Conference on Machine Learning*, pages 112–119, 2008.

[6] O. Celma. *Music Recommendation and Discovery in the Long Tail*. Springer, 2010.

[7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.

[8] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *ACM Symposium on Theory of Computing*, pages 537–546, 2008.

[9] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pages 253–262, New York, NY, USA, 2004. ACM.

[10] Christos Faloutsos and King-Ip Lin. Fastmap: a fast algorithm for indexing, data-mining and visualization. In *Proceedings of ACM SIGMOD*, pages 163–174, 1995.

[11] Tony Jebara, Risi Kondor, and Andrew Howard. Probability product kernels. *JMLR*, 5:819–844, December 2004.

[12] J.H. Kim, B. Tomasik, and D. Turnbull. Using artist similarity to propagate semantic information. In *ISMIR*, 2009.

[13] Ting Liu, Andrew W. Moore, Alexander Gray, and Ke Yang. An investigation of practical approximate nearest neighbor algorithms. In *NIPS*, pages 825–832. 2005.

[14] B. Logan. Music recommendation from song sets. In *International Symposium on Music Information Retrieval*, 2004.

[15] B. McFee, L. Barrington, and G.R.G. Lanckriet. Learning content similarity for music recommendation, 2011. http://arxiv.org/1105.2344.

[16] J. Reiss, J.J. Aucouturier, and M. Sandler. Efficient multidimensional searching routines for music information retrieval. In *ISMIR*, 2001.

[17] Dominik Schnitzer, Arthur Flexer, and Gerhard Widmer. A filter-and-refine indexing method for fast similarity search in millions of music tracks. In *ISMIR*, 2009.

[18] M. Slaney, K. Weinberger, and W. White. Learning a metric for music similarity. In *ISMIR*, pages 313–318, September 2008.

[19] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. 40(4):175–179, 1991.

[20] Nakul Verma, Samory Kpotufe, and Sanjoy Dasgupta. Which spatial partition trees are adaptive to intrinsic dimension? In *Uncertainty in Artificial Intelligence*, pages 565–574, 2009.