LEARNING BINARY CODES FOR EFFICIENT LARGE-SCALE MUSIC SIMILARITY SEARCH

Jan Schlüter
Austrian Research Institute for Artificial Intelligence, Vienna
jan.schlueter@ofai.at

ABSTRACT

Content-based music similarity estimation provides a way to find songs in the unpopular "long tail" of commercial catalogs. However, state-of-the-art music similarity measures are too slow to apply to large databases, as they are based on finding nearest neighbors among very high-dimensional or non-vector song representations that are difficult to index.

In this work, we adopt recent machine learning methods to map such song representations to binary codes. A linear scan over the codes quickly finds a small set of likely neighbors for a query to be refined with the original expensive similarity measure. Although search costs grow linearly with the collection size, we show that for commercial-scale databases and two state-of-the-art similarity measures, this outperforms five previous attempts at approximate nearest neighbor search. When required to return 90% of true nearest neighbors, our method is expected to answer 4.2 1-NN queries or 1.3 50-NN queries per second on a collection of 30 million songs using a single CPU core; an up to 260 fold speedup over a full scan of 90% of the database.

1. INTRODUCTION

Content-based music similarity measures allow to scan a collection for songs that *sound* similar to a query, and could provide new ways to discover music in the steadily growing catalogs of online distributors. However, an exhaustive scan over a large database is too slow with state-of-the-art similarity measures. A possible solution are *Filter-and-Refine* indexing methods: To find the k nearest neighbors (k-NN) to a query, a prefilter returns a small subset of the collection, which is then refined to the k best items therein.

Here, we consider the following scenario: (1) We have a commercial-scale music collection, (2) we want to return on average at least a fraction Q of the items an exhaustive scan would find, and (3) we cannot afford costly computations when a song enters or leaves the collection (ruling out nonparametric methods, or precomputing all answers). We then search for the fastest indexing method under these constraints.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page.

© 2013 International Society for Music Information Retrieval.

Compared to existing work on approximate k-NN search, what makes this quest special is the nature of state-of-the-art music similarity measures, and a low upper bound on database sizes: The largest online music store only offers 26 million songs as of February 2013, while web-scale image or document retrieval needs to handle billions of items.

Among the first approaches to fast k-NN search were space partitioning trees [1]. McFee et al. [12] use an extension of k-d trees on 890,000 songs, reporting a 120 fold speedup over a full scan when missing 80% of true neighbors. No comparison to other methods is given.

Hash-based methods promise cheap lookup costs. Cai et al. [2] apply Locality-Sensitive Hashing (LSH) [6] to 114,000 songs, but do not evaluate k-NN recall. Torralba et al. [23] learn binary codes with Restricted Boltzmann Machines (RBMs) for 12.9 mio. images, achieving 80% 50-NN recall looking at 0.2% of the database and decisively outperforming LSH. Using similar techniques, other researchers learn codes for documents [16] and images [8, 13], but, to the best of our knowledge, never for songs.

Pivot-based methods map items to a vector space only using distances to landmark items. Rafailidis et al. [15] apply L-Isomap to 9,000 songs. Schnitzer et al. [19] apply FastMap to 100,000 songs, achieving 80% 10-NN recall looking at 1% of the collection. Chávez et al. [3] map items to byte vectors with good results, but on our data, their approach needs 50% of a collection to find 80% of 1-NN.

In what follows, we will adapt the most promising methods for application on two music similarity measures and evaluate their performance under our scenario.

2. FILTER-REFINE COST MODEL

The cost of answering a query using a filter-and-refine approach can be decomposed into the cost for running the prefilter and the cost for refining the selection to k items:

$$\mathrm{cost}_{\mathrm{filtref}}(n,k) = \mathrm{cost}_{\mathrm{filt}}(n,k) + \mathrm{cost}_{\mathrm{ref}}(\mathrm{n}_{\mathrm{filt}}(n,k),k)$$

We assume that the refine step is a linear scan over the candidates returned by the prefilter, picking the k best:

$$\mathrm{cost}_{\mathrm{ref}}(n_f,k) = n_f \cdot (R + \log(k)\cdot S), \qquad (1)$$

where R is the cost for computing the distance of a query to a candidate, and $\log(k)\cdot S$ is the cost for updating a k-element min-heap of the best candidates found so far. [1]

[1] This model is not entirely correct, as the heap is generally not updated for each item. However, for $k \leq 100 \ll n_f$, we found the sorting cost to be indeed linear in n_f, which is all we need for our argument.

As a baseline method meeting our requirement of finding on average a fraction Q of the true neighbors, we adopt a zero-cost prefilter returning a fraction Q of the dataset:

$$\mathrm{cost}_{\mathrm{baseline}}(n,k) = \mathrm{cost}_{\mathrm{ref}}(Q \cdot n, k)$$

Under these assumptions, using a prefilter gives the following speedup factor over the baseline:

$$\begin{aligned}
\mathrm{spu}(Q,n,k) &= \frac{\mathrm{cost}_{\mathrm{ref}}(Q \cdot n, k)}{\mathrm{cost}_{\mathrm{filtref}}(n,k)} \\
&= \left(\frac{\mathrm{cost}_{\mathrm{filt}}(n,k) + \mathrm{cost}_{\mathrm{ref}}(\mathrm{n}_{\mathrm{filt}}(n,k),k)}{Q \cdot \mathrm{cost}_{\mathrm{ref}}(n,k)} \right)^{-1} \\
&= Q \cdot \left(\frac{\mathrm{cost}_{\mathrm{filt}}(n,k)}{\mathrm{cost}_{\mathrm{ref}}(n,k)} + \frac{\mathrm{n}_{\mathrm{filt}}(n,k)}{n} \right)^{-1} \\
&= Q \cdot \big(\rho_{\mathrm{t}}(n,k) + \rho_{\mathrm{s}}(n,k) \big)^{-1} \qquad (2)
\end{aligned}$$

We see that making the prefilter fast compared to a full scan (small ρ_{t}) is just as important as making the filter selective (small ρ_{s}). More specifically, we need to minimize the sum of the two ratios to maximize the speedup factor.

As we will see in our experiments, some existing methods put too much emphasis on a fast prefilter, resulting in $\rho_{\mathrm{t}}(n,k)$ being orders of magnitude smaller than $\rho_{\mathrm{s}}(n,k)$ especially for large databases. In this work we will balance the two ratios better to maximize the performance.

3. MUSIC SIMILARITY MEASURES

For our experiments, we choose two very different similarity measures: One based on high-dimensional vectors, and another based on Gaussian distributions.

3.1 Vector-Based Measure

Seyerlehner et al. [20] propose a set of six *Block-Level Features* to represent different aspects of a song's audio content, totalling in 9448 dimensions. These features work well for genre classification and tag prediction [21], and similarity measures based on them ranked among the top three algorithms in the MIREX Audio Music Similarity (AMS) tasks 2010–2012. For the similarity measure, the six feature vectors are individually compared by Manhattan distance, and the resulting feature-wise distances are combined to form the final similarity estimation.

To combine the feature-wise distances, they must be brought to similar scale. Instead of finding six appropriate scaling factors on an arbitrary dataset, Seyerlehner et al. normalize the feature-wise distance matrices for the handled collection: This *Distance Space Normalization (DSN)* processes each distance matrix entry by subtracting the mean and dividing by the standard deviation of its row and column. [2] The six normalized matrices are added up and normalized once again to form the final similarities.

While the normalizations seem unnecessarily complex, Flexer et al. [5] recently showed that they remove *hubs* – items appearing as neighbors of undesirably many other items – and are vital to achieve state-of-the-art results.

[2] When it is infeasible to compute full distance matrices, the song-wise distance statistics can be approximated from a random subset of the collection and stored with each feature vector.

3.2 Gaussian-Based Measure

As a second method, we use the timbre model proposed by Mandel and Ellis [11]: Each song is represented by the mean vector and covariance matrix of its frame-wise Mel-Frequency Cepstral Coefficients (MFCCs). [3] Song distances are computed as the symmetrized Kullback-Leibler divergence between these multivariate Gaussian distributions [17, p. 24], and normalized with DSN.

This measure does not reach state-of-the-art performance on its own, but forms the main component of [14], which ranked among the top two algorithms in the MIREX AMS tasks 2009–2012. Furthermore, it is easy to reproduce and allows direct comparison to Schnitzer et al. [18, 19].

4. INDEXING METHODS

We will evaluate seven different methods for fast k-NN search: One oblivious to the indexed dataset, four based on song models and two based on song similarities.

4.1 Locality-Sensitive Hashing (LSH)

For vectors in an Euclidean space, the family of projections onto a random line, followed by binary thresholding or fixed-width quantization, is *locality-sensitive*: For such projections, two close items are more probable to be mapped to the same value than two items far apart [6].

LSH uses $L \cdot K$ projections to map each item x_i to L discrete K-dimensional vectors $h_l(x_i)$. Using L conventional hash-table lookups, it can quickly find all items x_j matching a query q in at least one vector, $\exists_{l \le L} h_l(q) = h_l(x_j)$.

Here, this serves as a prefilter for finding neighbor candidates. Increasing K makes it more likely for candidates to be true nearest neighbors, but strongly reduces the candidate set size. Increasing L counters this, but increases query and storage costs. As a complementary way to increase the number of candidates, Multi-probe LSH [10] considers items with a *close* match in one of their vectors.

4.2 Principal Component Analysis (PCA)

PCA finds a linear transformation $\boldsymbol{y} = \boldsymbol{W'x}$ of Euclidean vectors $\boldsymbol{x_i} \in \mathcal{X}$ to a lower-dimensional space minimizing the squared reconstruction error $\sum_i \|\boldsymbol{x_i} - \boldsymbol{WW'x_i}\|_2^2$.

Nearest neighbors in the low-dimensional space are good candidates for neighbors in the original space, so a linear scan over items in the low-dimensional space serves as a natural prefilter. The candidate set size can be tuned at will to achieve a target k-NN recall. Increasing the dimensionality of the space allows to reduce the candidate set size, but increases prefilter costs.

4.3 Iterative Quantization (ITQ)

ITQ [7] finds a rotation of the PCA transformation minimizing squared reconstruction error after bit quantization of the low-dimensional space: $\sum_i \|\boldsymbol{x_i} - \boldsymbol{W}\,\mathrm{b}(\boldsymbol{W'x_i})\|_2^2$, where $\mathrm{b}_i(\boldsymbol{z})$ is 1 for positive z_i and 0 otherwise.

[3] Specifically, we use frames of 46 ms with 50% overlap, 37 Mel bands from 0 Hz to 11025 Hz and retain the first 25 MFCCs.

It can serve as a prefilter just like PCA, but using bit vectors reduces computational costs for the linear scan. For compact bit codes, neighbors within a small hamming distance of a query can alternatively be found with a constant number of conventional hash table lookups.

4.4 PCA Spill Trees

K-d Trees [1] are binary trees recursively partitioning a vector space: Each node splits the space at a hyperplane, assigning the resulting half-spaces to its two child nodes. Spill Trees [9] allow the half-spaces to overlap, making it less likely for close items to be separated. McFee et al. [12] additionally propose to choose hyperplanes perpendicular to a dataset's principal components, and to strongly restrict the depth of the tree. In the resulting *PCA Spill Tree*, each item ends up in one or more leaves, with similar items often sharing at least one leaf. Locating the leaves for an item is linear in the database size [12, Sec. 3.6], but can be avoided by precomputing all leaf sets.

As in [12], we regard all items in the leaf sets of a query to be candidate neighbors. The candidate set size can be increased by decreasing the tree depth or by increasing the overlap at each node.

4.5 Auto-Encoder (AE)

An AE finds a nonlinear transformation of inputs to a low-dimensional or binary code space and back to the input space, minimizing the difference between inputs and reconstructions (e.g., ℓ_2 distance for Euclidean input vectors). Similar to PCA and ITQ, candidate neighbors to a query can quickly be found in the code space.

The transformation is realized as an artificial neural network and can be optimized with backpropagation. For deep networks, it is helpful to initialize the network weights using Restricted Boltzmann Machines (RBMs). Salakhutdinov et al. [16] were the first to use a deep AE for approximate nearest neighbor search, under the term *Semantic Hashing*, and describe the method in detail.

4.6 Hamming Distance Metric Learning (HDML)

HDML [13] finds a nonlinear transformation to a binary code space optimized to preserve neighborhood relations of the input space. Specifically, for any triplet (x, x^+, x^-) of items for which x is closer to x^+ than to x^- in the input space, it aims to have x closer to x^+ than to x^- in the code space. Again, the transformation is realized as an artificial neural network, optimized with backpropagation, and HDML can be used as a prefilter just like ITQ or AE.

4.7 FastMap

FastMap [4] maps items to a d-dimensional Euclidean space based on their (metric) distances to d previously chosen pivot pairs in the input space. Schnitzer et al. [19] show how to apply FastMap to Gaussian-based models and propose an improved pivot selection strategy we will adopt.

FastMap serves as a prefilter like PCA, but supports non-vector models as it is purely distance-based.

5. EXPERIMENTS

We will now compare the seven indexing methods empirically, conducting a range of retrieval experiments.

5.1 Dataset and Methodology

From a collection of 2.5 million 30-second song excerpts used in [18, 19], we randomly select 120k albums of 120k different artists. We use 10k albums (124,013 songs) for training, 20k albums (246,117 songs) for validation and the remaining 90k albums (1,101,737 songs) for testing. In addition, we use 20k albums (253,347 songs) of the latter as a smaller test set.

For each applicable combination of similarity measure and indexing method, we will train different parameterizations of the method on the training set and determine the speedup over the baseline (Eq. 2) for retrieving on average 90% of the 1 or 50 nearest neighbors on the validation set. We will then evaluate the best parameterizations on the small test set to ensure we did not overfit on the validation set, and use the large test set to assess the methods' scalability.

5.2 Vector-based Measure

To be able to compute the speedup, we first determine the costs of the similarity measure. [4] Computing 1 million 9,448-dimensional Manhattan distances takes 2.361 s, finding the (indices of) the smallest 100 distances takes 1.17 ms, and both costs scale linearly with the collection size, as assumed in Eq. 1. Costs for the approximate DSN are negligible (see Sect. 3.1). For prefilters based on a linear scan, computing 1 million 80-dimensional ℓ_2 distances takes 22 ms, and computing 1 million 1024-bit hamming distances takes 9.8 ms. The costs of finding the best candidates in a linear scan depend on the candidate set size; we will use separate measurements for each case.

PCA: We start by evaluating PCA as a prefilter, as it proved useful as a preprocessing step for most other filters as well. To mimic how the similarity measure is combined from six features, we first apply PCA to each feature separately, compressing to about 10% of its size, then rescale each feature to unit mean standard deviation (this brings the distances to comparable ranges, and forms good inputs for the AE later) and stack the compressed features to form an 815-dimensional vector. Finally, we apply another PCA to compress these vectors to a size suitable for prefiltering.

In Table 1, we see that this cuts down query costs: For retrieving 90% of the true nearest neighbors, prefiltering with a linear scan over 40-dimensional PCA vectors takes $\rho_\mathrm{t} = 0.56\%$ the time of a full scan and only needs to examine $\rho_\mathrm{s} = 0.26\%$ of the database afterwards, resulting in a 110 fold speedup over the baseline $(0.9/(0.0052+0.0026)$, Eq. 2). For retrieving 50-NN, it needs a larger candidate set, increasing the prefilter costs (higher sorting costs to find the candidates), but still achieving a 47 fold speedup.

[4] All timings are reported on an Intel Core i7-2600 3.4 GHz CPU with DDR3 RAM, use a single core, and leverage AVX/POPCNT instructions. Implementations are in C, carefully optimized to maximize throughput.

Method		1-NN			50-NN		
		$\rho_t(\%)$	$\rho_s(\%)$	spu	$\rho_t(\%)$	$\rho_s(\%)$	spu
PCA	20 dim	0.38	0.59	93x	0.58	1.85	37x
	40 dim	0.56	0.26	110x	0.71	1.20	47x
	80 dim	1.01	0.18	76x	1.16	1.12	40x
LSH	8 bit	0.00	17.23	5x	0.00	23.82	4x
	16 bit	0.00	6.66	14x	0.00	11.00	8x
	20 bit	0.00	4.68	19x	0.00	8.42	11x
mp-LSH	128x16 bit	0.83	11.12	8x	0.83	34.18	3x
	64x32 bit	0.83	6.03	13x	0.83	12.23	7x
	32x64 bit	0.83	3.65	20x	0.83	7.59	11x
	16x128 bit	0.83	3.58	20x	0.83	7.11	11x
	1x256 bit	0.10	3.85	23x	0.10	7.98	11x
	8x256 bit	0.83	2.80	25x	0.83	6.22	13x
ITQ	64 bit	0.03	5.43	16x	0.03	9.94	9x
	128 bit	0.05	4.74	19x	0.05	8.27	11x
	Spill Tree	0.00	10.25	9x	0.00	21.27	4x
AE	64 bit	0.03	2.14	41x	0.03	4.40	20x
	128 bit	0.05	0.57	144x	0.05	2.47	36x
	256 bit	0.10	0.24	**265x**	0.10	1.28	65x
	512 bit	0.21	0.14	258x	0.21	0.93	79x
	1024 bit	0.42	0.09	177x	0.42	0.70	**81x**
FastMap	40 dim	0.77	1.56	39x	1.11	3.72	19x
	80 dim	1.20	1.36	35x	1.53	3.43	18x
	128 dim	1.73	1.20	31x	2.03	3.07	18x

Table 1. Results for the vector-based music similarity measure on the validation set of 246,117 songs: Ratio of prefilter time to full scan (ρ_t), ratio of candidate set to dataset size (ρ_s) and resulting speedup over baseline (spu) for retrieving 90% of 1 and 50 true nearest neighbors.

Varying the vector dimensionality changes the tradeoff between ρ_t and ρ_s, but does not improve the speedup.

LSH: We apply different versions of LSH to the 815-dimensional intermediate PCA representation. [5] First, we follow Slaney et al. [22] to compute optimal quantization width, dimensionality and table count for 90% 1-NN recall under the assumption that all projections are independent (it suggests 92.192, 25 and 430, respectively). To reach our target 1-NN recall, we need a 3-fold increase in table count and obtain $\rho_s = 16.52\%$, which is not competitive. Turning to binary LSH, we fix the dimensionality K to 8, 16 or 20 bit and increase L until we reach 90% recall (for 20 bits and 50-NN, we need 8353 hash tables). Even assuming zero prefilter costs, speedup is far below PCA. As a third alternative, we use a simple version of multi-probe LSH: We fix L and K, but consider all buckets within a hamming distance of r to the query in any of the tables. We increase r to reach the target k-NN recall, still achieving moderate speedups of up to 25x only.

ITQ directly builds on the PCA transform above, but maps items to bit vectors. Instead of directly tuning the candidate set size, we consider all items in a hamming ball of radius r around the query (in code space), and tune r. This avoids the sorting costs for finding the candidates. ITQ has small ρ_t, but large ρ_s, resulting in low speedups.

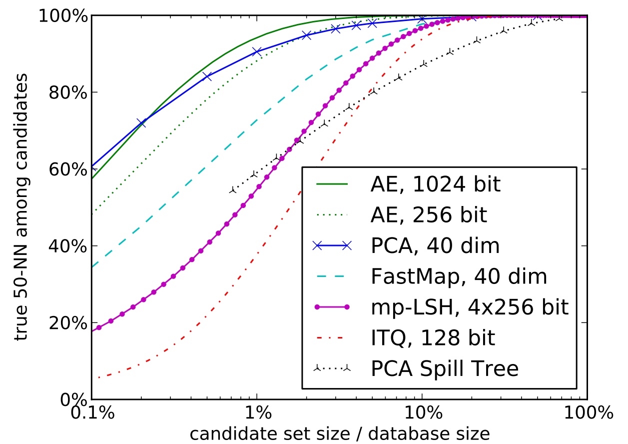

Figure 1. 50-NN recall versus candidate set size for the vector-based music similarity measure on the test set of 253,347 songs, averaged over all 253,347 possible queries.

PCA Spill Tree: We build a tree with spill factor 0.1 (the best performing in [12]) and adjust the depth to reach our target recall. Assuming zero prefilter costs, it achieves poor speedups as it needs very large candidate sets.

AE: We train a deep AE on the 815-dimensional intermediate PCA representation, pretrained with stacked RBMs as in [8]. We use an encoder architecture of 1024-256-128-64 layers for the shorter codes, 1024-512 and 2048-1024 for the two longer codes. [6] We encourage binary codes by adding noise in the forward pass as in [16]; especially for 128 bits and more, this worked better than thresholding as in [8]. For 256 bits and less, it also helped to encourage zero mean code unit activations as in [13, Eq. 12].

We use the learned codes as in ITQ. We obtain a prefilter which is both faster than PCA and more selective, achieving a 265 fold speedup for 1-NN and 81 fold speedup for 50-NN queries. Note how the accuracy of longer codes pays off for 50-NN, while shorter codes win for 1-NN.

HDML did not yield any improvement over AE.

FastMap is about twice as fast as LSH or ITQ, but falls behind AE and PCA.

We evaluate the best-performing instantiations of each method on the small test set and find results to be very similar to Table 1. As the relative prefilter costs ρ_t stay the same anyway, we only show how the candidate set size ρ_s and 50-NN recall interact (Fig. 1). We can see that the 1024-bit AE again only needs about 0.7% of the dataset to find 90% of 50-NN, and we see that AE and PCA perform best over a wide range of target recall values. Besides, comparison with [12, Fig. 4] shows that our PCA Spill Tree performs similar to its first publication.

[5] PCA is a useful stepping stone as the DSN (Sect. 3.1) invalidates any theoretical guarantees of LSH finding the nearest neighbors in the original space. Directly working on the 9448-dimensional vectors, rescaling the six components to comparable range, consistently gave worse results.

[6] Results are robust to the exact architecture as long as there is at least one layer before the code layer, and the first layer is wide enough.

Method		1-NN			50-NN		
		ρ_t(%)	ρ_s(%)	spu	ρ_t(%)	ρ_s(%)	spu
PCA	20 dim	6.18	16.03	4x	10.96	29.80	2x
	40 dim	6.00	14.04	4x	11.11	28.79	2x
AE	64 bit	0.06	11.89	8x	0.06	19.36	5x
	128 bit	0.11	6.95	13x	0.11	15.77	6x
	1024 bit	0.91	4.76	16x	0.91	13.13	6x
HDML	128 bit	0.11	1.46	57x	0.11	3.73	23x
	256 bit	0.23	1.37	56x	0.23	3.93	22x
	2x128 bit	0.23	1.15	**65x**	0.23	3.12	**27x**
	4x128 bit	0.45	1.09	58x	0.45	3.02	26x
	1024 bit	0.91	1.20	43x	0.91	4.65	16x
FastMap	40 dim	2.03	2.62	19x	3.37	6.48	9x
	80 dim	2.78	1.85	19x	3.85	4.94	10x
	128 dim	3.98	1.81	16x	5.03	4.85	9x

Table 2. Results for the Gaussian-based music similarity measure on the validation set of 246,117 songs.

5.3 Gaussian-based Measure

Again, we first determine the costs of the similarity measure: Computing 1 million symmetric Kullback-Leibler (sKL) divergences between 25-dimensional full-covariance Gaussian models takes 1.085 s, using precomputed inverse covariance matrices as in [17, Ch. 4.2]. Note that most indexing methods evaluated above are vector-based and not applicable to Gaussian models, so we expect the most from HDML and FastMap, but still try AE and PCA to be sure.

AE: In order for learned codes to be useful, they must reflect the input space. For the input space at hands, it seems natural to learn codes by minimizing the sKL divergence between inputs and reconstructions. For this to work, the AE must be forced to output valid covariance matrices Σ, otherwise it quickly learns to produce reconstructions that push the sKL divergence unboundedly below zero. We solve this by representing models in terms of the mean vector and Cholesky decomposition of Σ (multiplying the reconstructed Cholesky decomposition by itself transposed always gives a positive-semidefinite Σ), but our sKL-optimizing AEs only learn to reconstruct the centroid of all training data. Interestingly, however, ordinary ℓ_2-optimizing AEs benefit from the modified input representation. Using the same architectures as in Sect. 5.2 and a similar preprocessing (we separately compress mean vectors and Cholesky decompositions with PCA to 99.9% variance, then scale to unit mean standard deviation), we obtain moderate speedups of up to 16x.

PCA on the same representation performs poorly.

HDML learns codes from triplets of items (x, x^+, x^-), see Sect. 4.6. We select x^+ among the k^+ nearest neighbors of x, and x^- outside the 500 nearest neighbors. During training, we gradually increase k^+ from 10 to 200. Instead of training a randomly initialized network as in [13], we fine-tune the existing AEs. We obtain good results with 128-bit codes, but longer codes do not improve the speedup. To close the gap between ρ_t and ρ_s, we instead employ multiple 128-bit codes handled as in mp-LSH, obtaining an up to 65 fold speedup over the baseline.

FastMap is faster than AE, but slower than HDML. Results fall a bit behind [19] because unlike Schnitzer et al., we evaluate against nearest neighbors found with DSN.

Again, Fig. 2 demonstrates that our conclusions also hold for the test set and a wide range of target recall values.

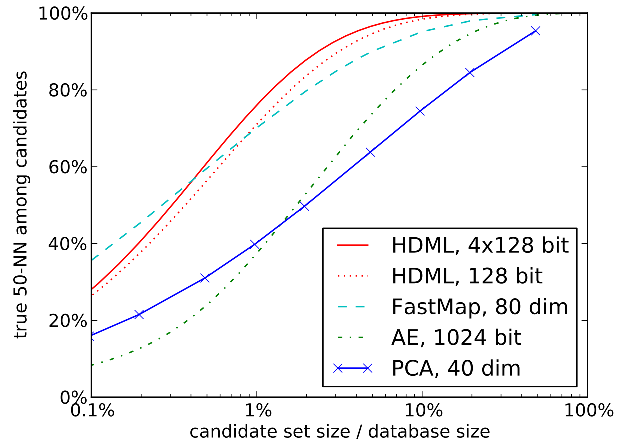

Figure 2. 50-NN recall versus candidate set size for the Gaussian-based music similarity measure on the test set of 253,347 songs.

Method		1-NN			50-NN		
		ρ_t(%)	ρ_s(%)	spu	ρ_t(%)	ρ_s(%)	spu
HDML	128 bit	0.11	1.19	69x	0.11	3.14	28x
	2x128 bit	0.23	0.93	**78x**	0.23	2.61	32x
	4x128 bit	0.45	0.88	67x	0.45	2.08	**36x**
FastMap	40 dim	1.56	0.94	36x	2.17	2.27	20x
	80 dim	2.53	0.85	27x	3.14	2.19	17x
	128 dim	3.67	0.69	21x	4.20	1.86	15x

Table 3. Results for the Gaussian-based music similarity measure on the test set of 1.1 million songs.

5.4 Scalability

Finally, we evaluate how the best-performing approaches scale with the collection size. For the large test set of 1.1 million songs, it is computationally infeasible to compute the exact DSN, and we do not want to evaluate an approximate retrieval algorithm against approximate ground truth. Thus, we will limit ourselves to the Gaussian-based measure, omitting the DSN altogether (as in [19]).

From Table 3, we find that the results scale better than linearly, because all methods need smaller candidate sets. For FastMap, the improvement is partly explained by evaluating against non-DSN neighbors: On the validation set, this alone improves the speedup by about 60% (it does not improve HDML, which seemingly learned the DSN well).

Still extrapolating linearly from the validation set to 30 million songs, the best methods are expected to answer 4.2 1-NN queries or 1.3 50-NN queries per second on the vector-based measure, and 2.2 1-NN queries or 0.9 50-NN

queries per second on the Gaussian-based one, using a single CPU core, with 90% true nearest neighbor recall. [7]

6. DISCUSSION

We have shown how to learn binary codes for song representations of two state-of-the-art music similarity measures that are useful for fast k-NN retrieval. Furthermore, we have demonstrated that for collection sizes encountered in MIR, a k-NN index based on a linear scan can outperform sublinear-time methods when we require a particular accuracy – even more so as scan-based methods are *embarrassingly parallel* and can be easily distributed or performed on a GPU. Note that our experiments explicitly targetted commercial-scale collections and song-level search. For user collections, PCA or FastMap will be preferable as they can quickly adapt to any dataset; training AE and HDML took 20 and 180 minutes, respectively. For similarity search on a finer scale (e.g., 10-second snippets), collections could grow to require sublinear-time methods.

For future work, it may be worthwhile to evaluate the binary representation in different scenarios: Do we obtain *qualitatively* good results using the codes alone, omitting the refine step? Do the codes prove useful for classification or clustering?

7. ACKNOWLEDGEMENTS

The author would like to thank Mohammad Norouzi for publishing his HDML implementation, and Maarten Grachten for fruitful discussions.

This research is supported by the Austrian Science Fund (FWF): TRP 307-N23. The Austrian Research Institute for Artificial Intelligence is supported by the Austrian Federal Ministry for Transport, Innovation, and Technology.

8. REFERENCES

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[2] R. Cai, C. Zhang, L. Zhang, and W.-Y. Ma. Scalable music recommendation by search. In *Proc. of the 15th Int. Conf. on Multimedia (ACM-MM)*, 2007.

[3] E. Chávez, K. Figueroa, and G. Navarro. Proximity searching in high dimensional spaces with a proximity preserving order. In *Proc. of the 4th Mexican Int. Conf. on Artificial Intelligence (MICAI)*, 2005.

[4] C. Faloutsos and K.I. Lin. Fastmap: A fast algorithm for indexing, data-mining and visualization of traditional and multimedia datasets. In *Proc. of the ACM SIGMOD Int. Conf. on Management of Data*, 1995.

[5] A. Flexer, D. Schnitzer, and J. Schlüter. A mirex meta-analysis of hubness in audio music similarity. In *Proc. of the 13th Int. Soc. for Music Information Retrieval Conf. (ISMIR)*, Porto, Portugal, 2012.

[7] This assumes all models are held in main memory, which can be practically achieved by distributing queries over a cluster. Preliminary experiments also indicate that the vector-based models can be compressed to 10% of their size with a minor impact on accuracy (also cf. [21]), removing a possible memory bandwidth bottleneck for the refine step.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. of the 25th Int. Conf. on Very Large Data Bases*, 1999.

[7] Yunchao Gong and Svetlana Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *Proc. of the IEEE Int. Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2011.

[8] A. Krizhevsky and G. Hinton. Using very deep autoencoders for content-based image retrieval. In *Proc. of the 19th Europ. Symp. on Artificial Neural Networks (ESANN)*, 2011.

[9] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Neural Information Processing Systems 17 (NIPS)*. 2005.

[10] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In *Proc. of the 33rd Int. Conf. on Very Large Data Bases*, 2007.

[11] M. Mandel and D. Ellis. Song-level features and support vector machines for music classification. In *Proc. of the 6th Int. Soc. for Music Information Retrieval Conf. (ISMIR)*, pages 594–599, 2005.

[12] B. McFee and G. Lanckriet. Large-scale music similarity search with spatial trees. In *Proc. of the 12th Int. Soc. for Music Information Retrieval Conf. (ISMIR)*, 2011.

[13] M. Norouzi, D. Fleet, and R. Salakhutdinov. Hamming distance metric learning. In *Neural Information Processing Systems 24 (NIPS)*. 2012.

[14] T. Pohle, D. Schnitzer, M. Schedl, P. Knees, and G. Widmer. On rhythm and general music similarity. In *Proc. of the 10th Int. Soc. for Music Information Retrieval Conf. (ISMIR)*, 2009.

[15] D. Rafailidis, A. Nanopoulos, and Y. Manolopoulos. Nonlinear dimensionality reduction for efficient and effective audio similarity searching. *Multimedia Tools Appl.*, 51(3):881–895, 2011.

[16] R. Salakhutdinov and G. Hinton. Semantic hashing. *Int. Journal of Approximative Reasoning*, 50(7), 2009.

[17] D. Schnitzer. Mirage – high-performance music similarity computation and automatic playlist generation. Master's thesis, Vienna Univ. of Technology, 2007.

[18] D. Schnitzer. *Indexing Content-Based Music Similarity Models for Fast Retrieval in Massive Databases*. PhD thesis, Johannes Kepler Univ. Linz, Austria, 2011.

[19] D. Schnitzer, A. Flexer, and G. Widmer. A filter-and-refine indexing method for fast similarity search in millions of music tracks. In *Proc. of the 10th Int. Soc. of Music Information Retrieval Conf. (ISMIR)*, 2009.

[20] K. Seyerlehner, G. Widmer, and T. Pohle. Fusing block-level features for music similarity estimation. In *Proc. of the 13th Int. Conf. on Digital Audio Effects (DAFx)*, 2010.

[21] K. Seyerlehner, G. Widmer, M. Schedl, and P. Knees. Automatic music tag classification based on block-level features. In *Proc. of the 7th Sound and Music Computing Conf. (SMC)*, Barcelona, Spain, 2010.

[22] M. Slaney, Y. Lifshits, and J. He. Optimal parameters for locality-sensitive hashing. *Proceedings of the IEEE*, 100(9), 2012.

[23] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2008.