TerraPattern: A Nearest Neighbor Search Service

Manzil Zaheer [1] **Guru Guruganesh** [1] **Golan Levin** [2] **Alexander Smola** [3]

Abstract

Creative inquiries through similarity search have historically offered novel perspectives in many domains (Broder, 1997; Singh et al., 2017; Kiefer & Laub, 2013; Ngai et al., 2011; Shin & Kim, 2014). A real-time online nearest neighbor (NN) search allows artists to expressively use the power of computation to explore vast datasets. We develop, Terrapattern, a prototype for visual query-by-example in satellite imagery. Terrapattern is an interface for finding "more like this, please" images in satellite photos: when a user selects an interesting tile on Terrapattern's map, the system finds other locations that look similar. This tool has been used by artists, citizen scientists, and hobbyists alike to spot and explore emerging trends, and has been well received by the media and public at large.

To facilitate a fast online NN search, we develop a tree data structure–Stable Greedy Tree (SG Tree)–which allows for almost linear time construction and logarithmic query time under real world assumptions. SG Tree is amenable to parallelization and distribution which greatly enhances its appeal over existing approaches such as NNet trees and Cover Trees and is essential for the scale required on large datasets. Moreover, the data structure adapts to the intrinsic dimension of the data, which is particularly beneficial when the feature vectors come from neural networks that have high euclidean dimensions but often lie on a (significantly) smaller dimensional manifold. The proposed method significantly outperforms existing NN search tools in construction time on many real world data-sets and is well suited for interactive queries.

1. Introduction

There has never been a more exciting time to observe humanity and understand it's impact on the world. Recognition of such patterns through similarity search has helped us in various domains: wildlife tracking and identifying conflicts (Suju & Jose, 2017), discovering survivors in battlefield dynamics (Han et al., 2015), detecting fraudulent online behavior (Ngai et al., 2011), diagnosis using medical histories (Korn et al., 1996), etc. We aim to help people discover such patterns and repeat this success story using satellite data.

In light of this, we present Terrapattern, a prototype to demonstrate a workflow by which lay users - such as journalists, citizen scientists, humanitarian agencies, and others - can easily search for visually consistent "patterns of interest". Our goal is to provide a geospatial software tool that makes it easy for people, who may lack expertise in machine vision, 1) to specify an image that they are interested in and automatically find similar examples , and 2) to provide the locations of those instances in a common data format that easily allows for further examination. After its launch, Terrapattern was well received by the media and the public at large (see (Coldeway, 2016; Ryan, 2016; Robinson, 2016)), with over 9.79M+ human queries till now.

Terrapattern consists of two components: a feature extraction service based on Deep Nets (see Section 4) and the main contribution of this paper, a nearest neighbor (NN) search. To be popular in the aforementioned task, such a NN search service would have to satisfy the following requirements:

- **Online**: Be responsive for interactive queries by artists, i.e. be able to handle online (not batch mode) queries over large high dimensional satellite imagery datasets in real-time. This rules out many efficient batch-mode NN search service running on GPUs (Johnson et al., 2017).
- **High Recall**: Must have high recall as humans are the direct consumers of the results of the NN search. Furthermore, the number of neighbors needed would be small. The super-fast approximate NN search methods work best when they must output a high number of neighbors at moderate recall (e.g. as an intermediary step of an automated ML pipeline), and thus are not applicable.
- **Fast Construction**: Allows for fast construction/indexing. For example, artists can load various datasets and start playing with it immediately.
- **Distributed**: For large datasets that do not fit on a single machine, the NN search service should be distributable over multiple machines, without high cost of synchronization in case of updates.

[1]Google Research, Mountain View CA [2]Carnegie Mellon University, Pittsburgh PA [3]Amazon Web Services, Palo Alto CA. Correspondence to: Manzil Zaheer <manzil@zaheer.ml>.

- **Malleable**: Possible to modify the search index (insert/delete points). This is useful, for example, when a higher resolution satellite imagery becomes available for certain region.

Since none of the existing NN search service meets all of our requirements, we develop a tree data structure Stable Greedy Trees (SG Tree). The proposed tree data structure allows for almost linear time construction and fast query time for exact NN search under real world assumptions (Section 2.3). Perhaps the most surprising features, are that SG Tree is extremely simple and as a result, our construction time is orders of magnitude faster than any comparable algorithm.

At a high level, the SG Tree tries to create a hierarchical tree where each node performs a "coarse" clustering. The centers of these "clusters" become the children and subsequent insertions are recursively performed on these children. When performing the NN query, we prune out solutions based on a subset of the dimensions that are being queried. This is particularly useful when trying to find the nearest neighbor in highly clustered subset of the data, e.g. when the data comes from a recursive mixture of Gaussians or more generally Time-Marginalized Coalescent. The effect of these two optimizations is that our data structure is extremely simple, highly parallelizable and is comparable in performance to existing NN implementations on many data-sets.

Another appealing aspect of our data structure is its simplicity. As a result, it is amenable to theoretical guarantees. Existing data structures with theoretical guarantees such as Navigating Net (Krauthgamer & Lee, 2004), Cover Trees (Beygelzimer et al., 2006), HNSW (Malkov & Yashunin, 2016), P-DCI, (Li & Malik, 2017), RP-tree (Dasgupta & Sinha, 2013), are either sequential in nature or hide large constants. We are able to show that SG Tree inherits some of the nice properties of Cover-Trees and adapts to the intrinsic dimension of the data. This is particularly beneficial when the feature vectors come from neural networks that have high euclidean dimensions but often lie on a (significantly) smaller dimensional manifold.

2. Stable Greedy Trees

2.1. Definition

Stable Greedy Trees (SG Tree) are defined with respect to a hyper-parameter $\gamma > 1$ over a dataset S in a metric space with distance $d(\cdot, \cdot)$. In SG Tree, the data points are hierarchically arranged where each node corresponds to a data point p and an associated maximum distance parameter τ , which will decrease exponentially with the depth of the node. The rate of decrease is controlled by the hyper-parameter γ. The sub-tree rooted at any node only contains data-points which are within τ from the data point p. All nodes that share a parent (i.e. siblings) are seperated by their

Algorithm 1 Insert a new point into SG Tree

```
function INSERT(Node n, Point p)
    # Start with n ← root
    Find child c of n closest in distance to p
    if d(c, p) < γ^{c.level} then
        if c.max < d(c, p) then
            c.max ← d(c, p)
        end if
        Insert(c, p)
    else
        Assign p as child of n
    end if
    return
end function
```

τ. Nodes at a certain height are said to be in the same level. An overview of SG Tree is provided in Figure 1. The tree structure will be more evident from the construction process, which sequentially inserts the data points from S with first point being the root. In more detail, each node is contains:

- `point`: The data point vector
- `level`: The level of the node in the tree as integer
- `max`: The upper bound to distance of furthest node in the current subtree.

Insertion: To insert a node at a given subtree rooted at r at level l , we find the closest child p. If $d(p, q) \leq \gamma^{l-1}$, then we recursively insert q into the subtree defined at p. On the other hand, if $d(p, q) > \gamma^{l-1}$ then we insert it a sibling in the current level and connect it to the parent at the last level. This is outlined in Algorithm 1.

Query: To find the closest neighbor to a query point, we traverse down the levels of the SG Tree starting from the root and maintain the current best node seen. At each level, we only maintain nodes that could still contain the nearest neighbor to a given query point p. In particular, a node q can be eliminated if none of its descendants can be closer than the current best. A lower bound on the distance between p and closest descendent of q can be worked out to be $d(p, q) - q_{\text{max}}$, by using triangle inequality and the maximum distance parameter. Thus, for eliminating p it suffices to check if this lower bound is worse than current best distance. We make this formal in Algorithm 2.

2.2. Connections to Net Trees and Cover Trees

SG Tree is inspired by and closely related to Cover Trees introduced by Beygelzimer et al. (2006). Cover trees form a hierarchical data structure that allows fast retrieval in logarithmic time when the metric has a small expansion constant (defined below). In particular, it allows for $O(n \log n)$ construction time, $O(\log n)$ retrieval, and it only depends

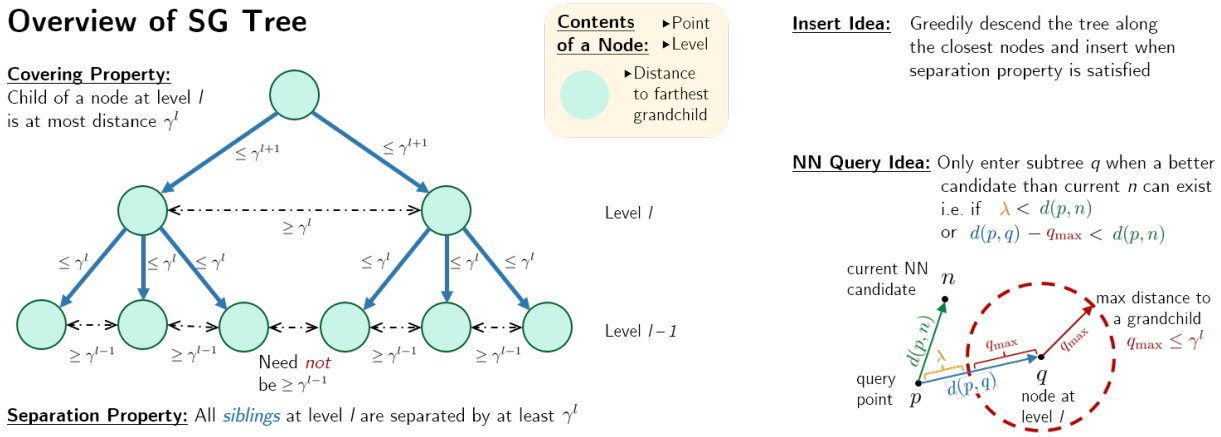


Figure 1. Overview of proposed hierarchical data structure that allows fast retrieval in log time.

Algorithm 2 Find nearest neighbor in SG Tree

```
function NN(Node r, Query point p, Candidate NN n)
    # Start with r ← root and n ← root
    if d(p, r) < d(p, n) then
        n ← r
    end if
    for each child q of r do
        if d(p, q) − q.max < d(p, n) then
            n ← NN(q, p, n)
        end if
    end for
    return n
end function
```

polynomialy on the expansion rate. Moreover, the degree of all internal nodes is well controlled.

Cover trees are defined as an infinite succession of levels S_l with $l \in \mathbb{Z}$. Each level l contains (a nested subset of) the data with the following properties:

- Nesting property: $S_{l-1} \subseteq S_l$.
- Separation property: All $p, q \in S_l$ satisfy $d(p, q) \geq \gamma^l$.
- All $q \in S_{l-1}$ have a parent in $p \in S_l$, possibly with $p = q$, with $d(p, q) \leq \gamma^l$.

The cover tree data structure compresses each node so each p is represented only once: it is only stored in the largest level l for which $p \in S_l$. This data structure has a number of highly desirable properties, as proved in Beygelzimer et al. (2006). SG Tree also satisfies all of those properties except for the separation property: it satisfies the separation property locally (i.e. only for siblings) instead of globally (i.e. for all nodes in S_l). In that sense, SG Tree changes this property from being a global one to a *local* one. This slight modification will have huge repercussions in performance, as it will allow us to insert in parallel and distribute nearest neighbor search.

Another very related data structure is Net Trees introduced by Krauthgamer & Lee (2004). Their data structure is also a hierarchical tree that has stronger properties. In fact, (Jahanseir & Sheehy, 2016) showed that Net Trees capture Cover Trees for an appropriate choice of parameters. They get stronger guarantees in part by maintaining epsilon nets at each level which are constructed greedily, and maintain lists of relatives at each point that maintain close neighbors. SG Tree also builds an epsilon net, however only on the points that are in a ball around some node. Once again, we substitute a global property for a local one. These changes lose many of the theoretical properties but we gain by having a simple implementation that can be optimized for modern architectures.

2.3. Structural Properties

In this section, we show that the SG Tree shares some of the properties of Cover Trees in terms of tree structure and construction time.

Definition 1. *Let* $\delta_{\max} := \max_{p,q \in S} ||p - q||$ *denote the maximum distance between any two points in* S *and* $\delta_{\min} := \min_{p \neq q \in S} ||p - q||$ *denote the minimum distance between any two points. We define the* ***aspect ratio*** *of the metric to be* $\Delta := \delta_{\max}/\delta_{\min}$.

Theorem 1. *The height of the SG Tree is at most* $O(\log(\Delta))$ *where* Δ *is the aspect ratio of the metric.*

Proof. Observe that the deepest level is at least $\log_\gamma(\delta_{\min}/2)$ as all balls of radius $\delta_{\min}/2$ are disjoint. The top level is at most $\log_\gamma(2 \cdot \delta_{\max})$ as all points are at most $\delta_{\max}$ apart. The total height of the tree is at most the difference in these levels: $O(\log(\frac{\delta_{\max}}{\delta_{\min}}))$. □

We use the definition of expansion constant α and the following lemmas from Beygelzimer et al. (2006).

Definition 2. *The expansion constant α is defined as smallest $\alpha \geq 2$ such that $|\mathcal{B}(p, 2r)| \leq \alpha \cdot |\mathcal{B}(p, r)|$ for all $p \in S$ and $r \geq 0$ where $\mathcal{B}(p, r)$ denotes a ball of radius r around the point p.*

Lemma 2. *The maximum degree of any node in this tree is at most α^3.*

Proof. Let r be a node at level l and let $c_1, \ldots, c_t$ be its children. Each c_i creates a disjoint ball of radius $\gamma^{l-1}/2$ as we know that $d(c_i, c_j) \geq \gamma^{l-1}$. WLOG let c_1 be the ball that contains the least number of points in a ball of radius $\mathcal{B}(c_1, \gamma^{l-1}/2)$. Observe that $\mathcal{B}(c_1, \gamma^{l-1}/2) \subseteq \mathcal{B}(r, \gamma^l) \subseteq \mathcal{B}(p, 2 \cdot \gamma^l)$. Since the larger ball is at most $4 \cdot \gamma$ times bigger than the smaller ball, and we choose $\gamma < 2$, we can bound the number of total points by the expansion constant. In particular, $|\mathcal{B}(p, 2 \cdot \gamma^l)| \leq \alpha^3 \cdot |\mathcal{B}(p, \gamma^{l-1}/2)|$. Since we chose p_1 so that $B(p_1, \gamma^{l-1}/2)$ to have the least number of points, we know that there can be at most α^3 such nodes. □

Corollary 3. *The time to insert a new point p is $\alpha^3 \log(\Delta)$.*

Proof. This is an easy consequence of Lemma 2 and Theorem 1. Upon insertion, we query all the children of a node which is at most α^3 and proceed to the next level. Since there are at most $\log(\Delta)$ levels and each node has at most α^3 children, this gives us the required bound. □

2.4. Analysis and Extensions

As we do not have a global separation property, this severely limits our ability to analyze the tree under a *worst case* model. Therefore we analyse SG Tree under a generative model, namely Time-Marginalized Coalescent (TMC) Process. TMC is a popular generative process for modelling hierarchical data (Kingman, 1982b;a; Boyles & Welling, 2012; Teh et al., 2008; Vikram et al., 2019). TMC defines a distribution over binary trees. In this model, we have binary tree together with time labels associated with each node, i.e. formally we have a triplet (V, E, T) where V is the set of nodes, E is the set of edges, and the time labels is given by a function $\tau : V \to [0,\ 1]$ where we denote $t_v = \tau(v)$. In particular, we will deal with a special case of the TMC model which we call recursive Gaussian Mixture Model (rGMM). We formally state this model in the Appendix D. which is a special case of Time-Marginalized Coalescent process (Kingman, 1982b; Boyles & Welling, 2012). Informally, it creates a tree structure whose leafs denote the cluster centers. To generate a new point, one simply randomly walks from the root to the leaf, and then outputs a point according to the Gaussian whose parameters are determined by the leaf. We give a formal description in Appendix D.

Lemma 4. *Given n points from a $rGMM(c, \log n)$, and query a new points q drawn from the same rGMM distribution, SG Tree will take $O(c \log n)$ time to find the closest neighbor.*

To make the analysis more tractable, we make a slight assumption of the insertion procedure. In particular, we assume that the nodes are inserted in decreasing order of level. It is easy to ensure the above if the insertions are done in batch mode and we find that it only incurs a small increase in construction time.

By ensuring that the nodes are inserted in decreasing order of level, we guarantee that the children of a particular node, partition the remaining points according to the Voronoi Partition. I.e. all the grandchildren are connected to the child that they are closest to. Thus, they form an actually clustering based on the children as the centers. We defer the proof of Theorem 4 to Appendix D.

Extension to some improper distances Apart from finding NN according to L2 distance in Euclidean space, there are other popular naturally occurring criteria that are not proper metrics such as: angular distance, maximum cosine similarity or maximum inner product search. These criteria do not obey triangle inequality, so SG Tree cannot be directly used. Following Ram & Gray (2012) and Bachrach et al. (2014), we can show that SG Tree can be used for even such improper distances by performing suitable and cheap transformations of the data points.

- Maximum Cosine Similarity Search (MCSS): Instead of inserting data points x_i in SG Tree, insert normalized version $\tilde{x}_i = x_i / \|x_i\|$. Performing NN search in ℓ_2-distance is equivalent to MCSS as:

$$\begin{aligned}
\text{NN}(q|\tilde{x}) &= \operatorname*{argmin}_i \|\tilde{x}_i - q\| = \operatorname*{argmin}_i \|\tilde{x}_i - q\|^2 \\
&= \operatorname*{argmin}_i \|\tilde{x}_i\|^2 + \|q\|^2 - 2\langle \tilde{x}_i, q \rangle \\
&= \operatorname*{argmax}_i \langle \tilde{x}_i, q \rangle = \operatorname*{argmax}_i \frac{\langle x_i, q \rangle}{\|x_i\| \|q\|} \\
&= \text{MCSS}(q|x)
\end{aligned}$$

- Maximum Inner Product Search (MIPS): As before, we will insert transformed points in SG Tree. Assume, we can have an upper bound u for the L2 norm, i.e. $u \leq \|x\|,\ \forall x$. Instead of inserting original data points x_i, we insert the augmented data points $\tilde{x}_i = [x_i; \sqrt{u^2 - \|x_i\|^2}]$ in SG Tree. The query vector q is also modified as $\tilde{q} = [q; 0]$. Now performing NN search in L2 distance with modified query is equivalent to MIPS for original query as:

$$\begin{aligned}\text{NN}(\tilde{q}|\tilde{x}) &= \underset{i}{\text{argmin}}\, \|\tilde{x}_i - \tilde{q}\| = \underset{i}{\text{argmin}}\, \|\tilde{x}_i - \tilde{q}\|^2 \\ &= \underset{i}{\text{argmin}}\, u^2 + \|q\|^2 - 2\langle x_i, q\rangle \\ &= \underset{i}{\text{argmax}}\langle x_i, q\rangle = \text{MIPS}(q|x)\end{aligned}$$

2.5. Deployment

We perform a lot of empirical experimentation and find our simple data structure SG Tree performs competitively. We outline several important heuristics that improve performance significantly for SG Tree as well as other baseline data-structures like Cover Tree. We state these heuristics below and provide some theoretical justification for them.

Initialization Heuristic We mention a heuristic that makes SG Tree (and cover trees as well) more balanced and improves their performance: Start with a sample of the data points representative of the data distribution. Find the point closest to the (empirical) mean in this estimate. Use this point as the root of the tree. Next, insert remaining points from the sample in descending order of the distance. Despite slight additional work at the beginning, this heuristic reduces both the overall construction time and query search time.

Empirically, we believe that this heuristic allows top layers of the tree to quickly spread out, effectively performing a coarse clustering of the the data and allowing more effective pruning during search.

Faster Rejection Heuristics While performing query search, one of the most expensive operations is the actual distance computation. Most theoretical papers treat this as an atomic operation which is not a valid assumption in practice. We introduce the following simple heuristic that helps to speed up the nearest neighbor search considerably. In Line 7 of Algorithm 2, we first check if $d_{1/2}(p, q) - q.max < d(p, n)$ where $d_{1/2}(p, q)$ evaluates the distance between p and q restricted to the (first) half of the n coordinates.

Parallelized Construction The main advantage of SG Tree over the theoretically guaranteed, cover trees come from absence of a global separation constraint, which makes parallelization easy. To optimally use a modern heavily multi-cored CPU, we should insert a batch of points in parallel using the threadpool. For example,in a cover tree, if any worker thread inserts a point, every other worker thread has to ensure that the inserted point does not violate the global separation property for their point. In SG Tree, the worker does not need to worry about modifications to the tree happening at places other than current node because separation property needs to be only maintained locally among siblings. Even if current node is being modified, the worker does not need to discard any work, but only has to account for the newly inserted point. Thus, an efficient construction for SG Tree can be implemented in a work-stealing Fork/Join framework.

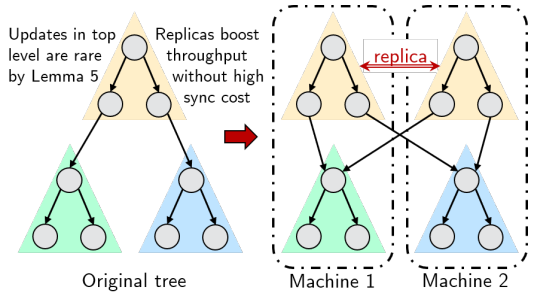


Figure 2. A natural distributed implementation of SG Tree. Multiple replicas can be maintained cheaply, i.e. with low synchronization cost as modifications to top layers are rare by Lemma 5.

In other methods, like HNSW, RP-Trees, etc. bulk of the work can also be parallelized, but the total work needed is often much higher, as seen from empirical study (App. Table 1). For example, building small world graphs in worst case is $O(n^2)$ (Fu et al., 2017), and unknown for others.

Distributed high throughput implementation For large scale data, such as satellite imagery data of whole earth (¿ 8TB), that do not fit in memory of a single computer, one resorts to distribution across multiple machines. In a natural setup for a distributed tree search, as first proposed by Patwary et al. (2016) using kd-trees, first the data would be spatially partitioned into roughly equal size portions and distributed among the nodes. Then a global kd-tree containing representation data points of these spatial partitions would be constructed and would be used to direct queries/inserts to appropriate nodes. Results from the polled nodes would be aggregated and returned. However, with kd-tree such spatial partitioning of can be very expensive to construct. Moreover with new points coming in the spliting points/directions in the global kd-tree might have to be updated often to maintain fast look-ups.

SG Tree has favorable properties for distributed systems. First, no two-step construction is needed. The tree can be grown normally till memory is exhausted, at which point sub-trees starting from level k can be distributed among other nodes. The data can be continued to be added after distribution. Unlike kd-trees, in SG Tree there is no axis aligned splits that have to be recomputed with new points coming in. Second, the top-k levels of SG Tree can be replicated across all nodes, without incurring much synchronization penalty due to Lemma 5 (proof sketch in the App. D). It says modifications in top-k levels of SG Tree are rare, and replica would hardly needed to be synchronized. Thus, every node can process incoming queries and direct it to appropriate nodes, thereby increasing the throughput as illustrated in Figure 2.

Lemma 5. *Let us construct SG Tree with $n \geq c^T$ points sampled from distribution $rGMM(c, T)$ and insert a new point, then with high probability, the new point will not create a new node in the first $T = o(\log n)$ levels.*

3. Experiments

In this section, we present empirical studies comparing SG Tree with other data structures. These show that SG Tree has fast construction speed and competitive query time meeting the needs of TerraPattern service. We have conducted an extensive comparison with existing software and benchmarks and include all of the settings (hyperparameters etc) in the supplementary material for the interested reader.

Datasets To test the above claims, we evaluate on a number of datasets from UCI repository[1] and from the popular ANN-Benchmark[2], as listed in Figure 3 along with their size (N) and dimensionality (D). These are multivariate datasets from a varied set of sources meant to provide a broad picture of performance across different domains. They also include a mix of Euclidean and Angular distance (1-cosine similarity) as the natural metric for NN query among the selected datasets. The datasets come with a train and test split; we use the train set to build the NN index and test set for query.

Method We perform our experiments with respect to three metrics: *experience time, construction time* and *query time*. The most import metric in our application is the "experience time", i.e. the total time taken for constructing the index and performing 1k queries to find the 10-NN in the index for each query vector. This metric better reflects the experience of an user who interactively wants to perform some NN search. The traditional metrics of construction time and time taken per query are provided in Appendix C. We compare our multi-threaded implementation of SG Tree in C++14 against following methods, each covering a broad class of NN search strategy[3]:

- Cover Tree (Beygelzimer et al., 2006): We use an in-house C++14 implementation of Cover Tree as no open source multi-threaded code was available that exactly implemented cover tree.[4] Although construction is not parallelizable, we tried to parallelize distance computations where possible. Also to be fair we added the heuristics for speed-up. It is an exact search method.
- Random Projection (RP) Tree: We use Annoy[5], an highly optimized implementation of forest of RP trees in C++11. It is an approximate search method based on space partitioning trees.
- Hierarchical Navigation Small World (HNSW) Graphs (Malkov & Yashunin, 2016): We use the excellent implementation of HNSW from NMSLIB[6] in C++11. It builds a proximity graph at multiple resolutions for the points in index. It is also an approximate search method in the category of neighborhood based methods.
- Prioritized Dynamic Continuous Indexing (P-DCI) (Li & Malik, 2017): We use the multi-threaded C code[7] released by the authors. The method is a clever application of Johnson-Lindenstrauss lemma to query in lower dimension at a much lower cost. Theoretically, it is an exact search method with high probability, but in practice can be considered to be an approximate search method.

For the approximate search methods, we select hyperparameters such that Recall@10 ≥ 0.99, i.e. we require recall to be high as needed by TerraPattern. The final hyperparameter used by each method is reported in Appendix B. Also note that HNSW does not allow modification to the index, which is undesirable for our application, but we still include them in comparison.

Setup & Hardware We follow the experimental setup of the popular ANN-Benchmark . Similar to their protocol, the queries are not batched as we are interested in online queries. Instead each query is processed individually, but in parallel using a thread pool that saturates all the CPU cores. We run our experiments on an Amazon EC2 c5.18xlarge nodes having 72 virtual threads per node and 144GB of memory. In the experiments, all data and calculations are carried out at single floating-point precision.

Hyperparameter Selection: We tune each hyperparameter according to the guidelines specified by the packages to maximize performance. We do so in as scientific a manner as possible, while ensuring recall@10 ¿ 0.99, as stated in Line 267. The chosen hyper-parameters are are reported in Appendix A. We elaborate further below:

- RP-Tree/Annoy has two hyper-parameters: `n_tree` and `search_k`. Larger values of either will yield more accurate results, but at the expense of higher build & search time. We do a line search on `n_tree` with default setup of `search_k` until we hit Recall@10 ¿ 0.99. Then we

[1] https://archive.ics.uci.edu/ml

[2] https://github.com/erikbern/ann-benchmarks

[3] We provide a rationale for selection of the algorithms for comparison in Appendix A, e.g. not using Faster Cover Trees (Izbicki & Shelton, 2015).

[4] MLPack claims to implement Cover Tree, but they ignore the global separation property https://github.com/mlpack/mlpack/blob/master/src/mlpack/core/tree/cover_tree/cover_tree.hpp#L39

[5] https://github.com/spotify/annoy

[6] https://github.com/nmslib/nmslib/

[7] https://github.com/UOMXiaoShuaiShuai/PDCI

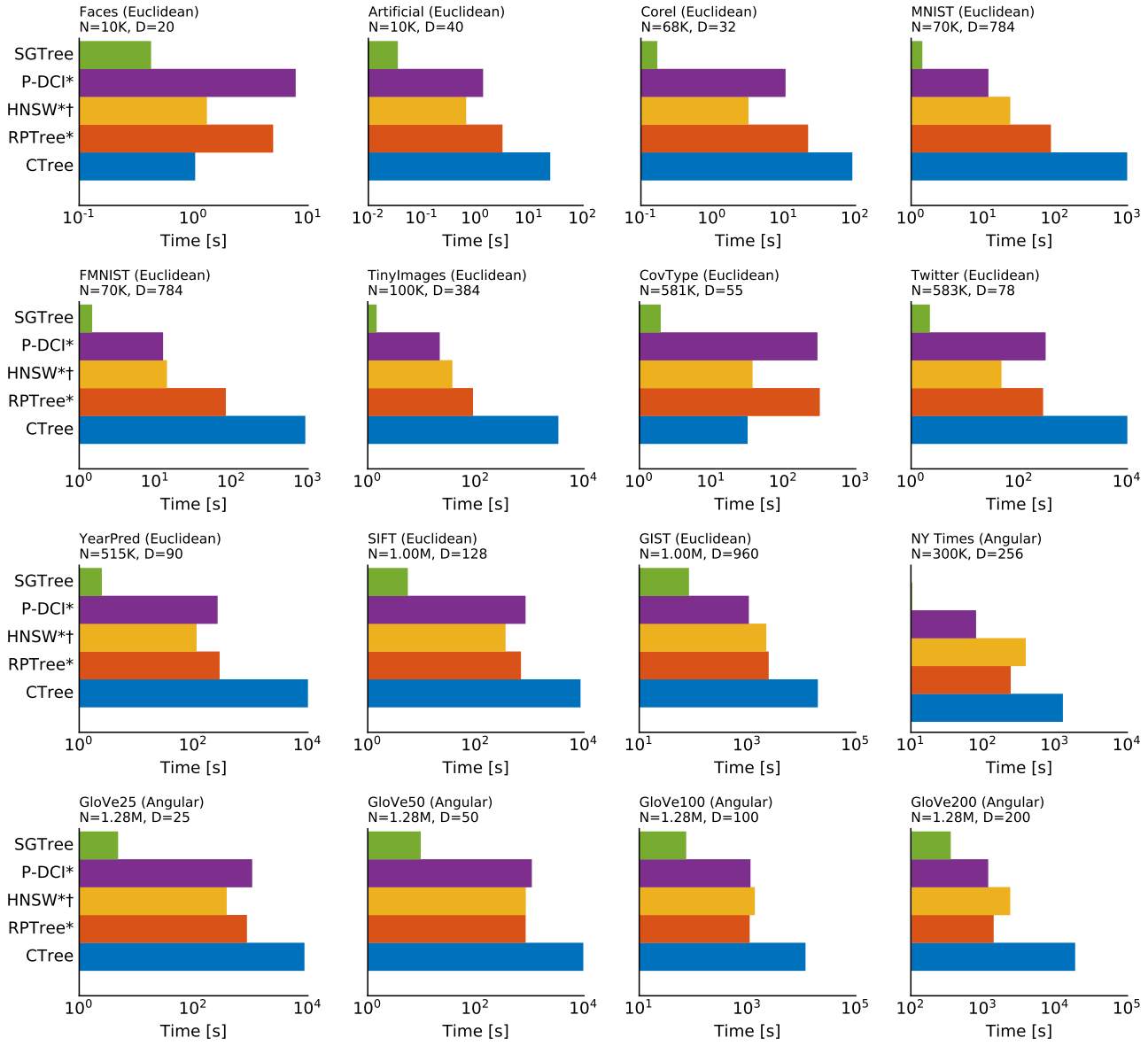


Figure 3. Comparison of SG Tree on numerous benchmark datasets in terms of "experience time", i.e. total time taken to build the index and perform 1K queries. Approximate NN methods are marked with * and for such methods the hyper-parameters are chosen to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

reduce `search_k` to improve runtime till recall does not degrade.

- HNSW/NMSLIB has multiple hyper-parameters, too many to search in a systematic fashion. We followed the guide provided by developers (Page 32, section 4.5.2). The important parameters are `M`, `efCons`, and `efSearch`. Following their recommendation "One way to check if the selection of `efCons` was ok is to measure a recall for `M` nearest neighbor search when `efSearch=efCons`: if the recall is lower than 0.9, then there is room for improvement.", we tied `efCons=efSearch` and kept increasing them until we hit Recall@10 > 0.99.
- P-DCI has multiple hyper-parameters, but without any guidance for selecting them. Thus starting from default, we tried our best to select parameters for fastest performance by making perturbations in each coordinate until they decreased performance. Finally obtained hyper-parameters are reported in Appendix B.

Observations Overall, Figure 3 is in line with our claim that SG Tree can enhance user experience for interactive NN search compared to other methods because it takes

lowest time for build plus search across multiple datasets of varying sizes, dimenionality, and distance. Even in terms of time taken per query (or equivalently queries per second), SG Tree is quite competitive with other state of the art methods like HNSW which are not modifiable as can be seen from Table 2 (in supplementary materials). Another salient observation is regarding the GloVe datasets, which are word embeddings of various ambient dimensions 25, 50, 100, 200. Ideally all of these vector collections should contain similar semantic information and thus posses similar intrinsic dimension.

Separate Indexing & Query Time: Although, the mixed indexing and query time are the primary use case for our needs, we report indexing & querying separately in Table 2 and Table 3, fig. 5 and fig. 6 of Appendix. As can be seen, SG Tree significantly outperforms existing NN search tools in construction time and is competitive on query time, thus well suited for interactive queries. All algorithms are multi-threaded in both Indexing and query time and use the same number of threads.

4. TerraPattern Service

Terrapattern provides an open-ended interface for visual query-by-example, a test bed for our SG Tree nearest neighbor search algorithm. Simply click an interesting spot on Terrapattern's map, and it will find other locations that look similar. Our tool is ideal for locating specialized 'nonbuilding structures' and other forms of soft infrastructure that aren't usually indicated on maps. Terrapattern is a "panoptic perceptron" that allows a user to perform arbitrary queries-by-example in satellite imagery. A guest clicks on a "feature of interest" in a satellite image; the Terrapattern system presents a batch of the most similar-looking places nearby; and the guest can then download a list of these locations in GeoJSON format. An example query is shown in Figure 4 for school bus parking.

We emphasize that Terrapattern is a limited prototype. As of May 2019, it allows users to search in the greater metropolitan regions of a few major cities: New York City, San Francisco, Pittsburgh, Berlin etc. Altogether more than $5,200$ square miles are fully searchable. Allowing high-resolution searches in size of the United States (e.g. 3.8M mi^2) is financially beyond the scope of project. Website: `anonymized`.

TerraPattern uses a deep convolutional neural network (DCNN), based on the ResNet architecture (He et al., 2016). We trained a 34-layer DCNN using hundreds of thousands of satellite images labeled in OpenStreetMap, teaching the neural network to predict the category of a place from a satellite photo. Terrapattern was only possible due to the astonishing crowdsourced mapping effort of the OpenStreetMap project, which has generously categorized large parts of the

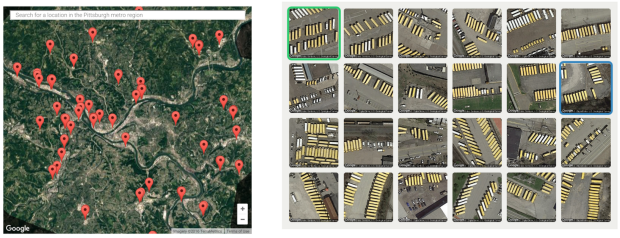


Figure 4. Using Terrapattern we can identify some of Pittsburgh's finest school bus depots.

world with its Nominatim taxonomy. We trained our DCNN using 466 of the Nominatim categories (such as "airport", "marsh", "gas station", "prison", "monument", "church", etc.), with approximately 1000 satellite images per category. Our resulting model, which took 5 days to compute on an nVidia Titan Xp GPU, has a top-5 error rate of 25.4%. In the process, our network learned which high-level visual features (and combinations of those features) are important for the classification of satellite imagery.

After training the model, we removed the final classification layer of the network and extracted the next-to-last layer of the DCNN. Using this layer of proto-features (a technique called "transfer learning"), we computed descriptors for millions more satellite photos that cover a few metropolitan regions. When we want to discover places that look similar to your query, we just have to find places whose descriptors are similar to those of the tile you selected. To perform this search in near real time, we use the proposed SG Tree algorithm for kNN. Some interesting patterns found are illustrated in Appendix E.

5. Conclusion

We have introduced a new service for fast nearest neighbor search on satellite images. To do this, we construct a natural data-structure: SG Tree which has nice theoretical properties in bounding its depth and construction time. Furthermore, it provides state of the art results for metric of experience time (construction + query time) which is very useful for the purposes of nearest neighbor search. In particular, SG Tree significantly outperforms existing NN search tools in construction time, while being competitive in query time on many real world data-sets and thus is well suited for interactive queries. We hope that this data-structure along with its distributed implementation will find wider use in the community.

TerraPattern has been positively endorsed by a variety of communities to fulfil a number of different roles, with total human queries just shy of 10M. We feel we have only scratched the surface of what is possible both from systems and algorithms point of view, to a practical service that can help unlock the potential of satellite images to the world.

References

Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I., and Schmidt, L. Practical and optimal lsh for angular distance. In *Advances in Neural Information Processing Systems*, pp. 1225–1233, 2015.

Bachrach, Y., Finkelstein, Y., Gilad-Bachrach, R., Katzir, L., Koenigstein, N., Nice, N., and Paquet, U. Speeding up the xbox recommender system using a euclidean transformation for inner-product spaces. In *Proceedings of the 8th ACM Conference on Recommender systems*, pp. 257–264. ACM, 2014.

Beygelzimer, A., Kakade, S., and Langford, J. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pp. 97–104. ACM, 2006.

Boyles, L. and Welling, M. The time-marginalized coalescent prior for hierarchical clustering. In *Advances in Neural Information Processing Systems*, pp. 2969–2977, 2012.

Broder, A. Z. On the resemblance and containment of documents. In *Compression and complexity of sequences 1997. proceedings*, pp. 21–29. IEEE, 1997.

Clarkson, K. L. Nearest neighbor queries in metric spaces. *Discrete & Computational Geometry*, 22(1):63–93, 1999.

Coldeway, D. Terrapattern is reverse image search for maps, powered by a neural network". *Techcrunch*, May 2016. URL https://techcrunch.com/2016/05/25/terrapattern-is-a-neural-net-powered-reverse-image-search-for-maps/

Dasgupta, S. and Sinha, K. Randomized partition trees for exact nearest neighbor search. In *Conference on Learning Theory*, pp. 317–337, 2013.

Fu, C., Xiang, C., Wang, C., and Cai, D. Fast approximate nearest neighbor search with the navigating spreading-out graph. *arXiv preprint arXiv:1707.00143*, 2017.

Han, Y., Park, K., Hong, J., Ulamin, N., and Lee, Y.-K. Distance-constraint k-nearest neighbor searching in mobile sensor networks. *Sensors*, 15(8):18209–18228, 2015.

Har-Peled, S. and Mendel, M. Fast construction of nets in low-dimensional metrics and their applications. *SIAM Journal on Computing*, 35(5):1148–1184, 2006.

He, K., Zhang, X., Ren, S., and Sun, J. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 770–778, 2016.

Huang, Q., Feng, J., Zhang, Y., Fang, Q., and Ng, W. Query-aware locality-sensitive hashing for approximate nearest neighbor search. *Proceedings of the VLDB Endowment*, 9(1):1–12, 2015.

Izbicki, M. and Shelton, C. Faster cover trees. In *International Conference on Machine Learning*, pp. 1162–1170, 2015.

Jahanseir, M. and Sheehy, D. Transforming hierarchical trees on metric spaces. In *CCCG*, pp. 107–113, 2016.

Johnson, J., Douze, M., and Jégou, H. Billion-scale similarity search with gpus. *arXiv preprint arXiv:1702.08734*, 2017.

Karger, D. R. and Ruhl, M. Finding nearest neighbors in growth-restricted metrics. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pp. 741–750. ACM, 2002.

Kiefer, C. and Laub, J. Google Faces. https://onformative.com/work/google-faces, 2013. Accessed: 2018-08-15.

Kingman, J. F. On the genealogy of large populations. *Journal of applied probability*, 19(A):27–43, 1982a.

Kingman, J. F. C. The coalescent. *Stochastic processes and their applications*, 13(3):235–248, 1982b.

Korn, F., Sidiropoulos, N., Faloutsos, C., Siegel, E., and Protopapas, Z. Fast nearest neighbor search in medical image databases. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pp. 215–226. Morgan Kaufmann Publishers Inc., 1996.

Krauthgamer, R. and Lee, J. R. Navigating nets: simple algorithms for proximity search. In *Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms*, pp. 798–807. Society for Industrial and Applied Mathematics, 2004.

Li, K. and Malik, J. Fast k-nearest neighbour search via prioritized dci. In *International Conference on Machine Learning*, pp. 2081–2090, 2017.

Li, W., Zhang, Y., Sun, Y., Wang, W., Zhang, W., and Lin, X. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement (v1. 0). *arXiv preprint arXiv:1610.02455*, 2016.

Malkov, Y. A. and Yashunin, D. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320*, 2016.

Muja, M. and Lowe, D. G. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pp. 331–340, 2009.

Ngai, E. W., Hu, Y., Wong, Y., Chen, Y., and Sun, X. The application of data mining techniques in financial fraud detection: A classification framework and an academic review of literature. *Decision Support Systems*, 50(3): 559–569, 2011.

Patwary, M. M. A., Satish, N. R., Sundaram, N., Liu, J., Sadowski, P., Racah, E., Byna, S., Tull, C., Bhimji, W., Dubey, P., et al. Panda: Extreme scale parallel k-nearest neighbor on distributed architectures. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pp. 494–503. IEEE, 2016.

Ram, P. and Gray, A. G. Maximum inner-product search using cone trees. In *Proceedings of the 18th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 931–939. ACM, 2012.

Robinson. The thrill of terrapattern, a new way to search satellite imagery. *The Atlantic*, May 2016. URL https://www.theatlantic.com/technology/archive/2016/05/the-promise-of-terrapattern-the-visual-search-engine-for-satellite-imagery/484610/.

Ryan. This website lets you find the hidden similarities in big cities. *Popular Science*, May 2016. URL https://www.popsci.com/you-can-now-search-google-maps-by-matching-similar-landscapes.

Shin, S. B. and Kim, Y. H. Cloud Face. http://ssbkyh.com/works/cloud_face, 2014. Accessed: 2018-08-15.

Singh, S., Ram, F., and De Graef, M. Application of forward models to crystal orientation refinement. *Journal of Applied Crystallography*, 50(6):1664–1676, 2017.

Suju, D. A. and Jose, H. Flann: Fast approximate nearest neighbour search algorithm for elucidating human-wildlife conflicts in forest areas. In *Signal Processing, Communication and Networking (ICSCN), 2017 Fourth International Conference on*, pp. 1–6. IEEE, 2017.

Teh, Y. W., Daume III, H., and Roy, D. M. Bayesian agglomerative clustering with coalescents. In *Advances in Neural Information Processing Systems*, pp. 1473–1480, 2008.

Vikram, S., Hoffman, M. D., and Johnson, M. J. The loracs prior for vaes: Letting the trees speak for the data. In *AISTATS*, volume 89 of *Proceedings of Machine Learning Research*, pp. 3292–3301. PMLR, 16–18 Apr 2019.

A. Algorithms for Comparison

Existing algorithms for approximate NN search can be categorized into following classes and we tried to include best algorithms from each class:

- *Locality sensitive hashing (LSH):* There are theoretical guarantees on query result quality, efficiency, and memory requirements. Software packages like FALCONN and QALSH implement variant of this idea. Typically they are modifiable and fast, but have relatively lower recall. We didn't find any competitive open source implementation with high recall. Ann-benchmark remarks that "FALCONN ... [is] the only library I've seen that gets decent results using locality sensitive hashing. Other than that, I haven't been very impressed by LSH. Graph-based algorithms seem to be the state of the art, in particular HNSW". [8]. We tried FALCONN but could not achieve very good recall, similar to result by ANN-Benchmark.[9]. Alternatively instead of hashing, Li & Malik (2017) cleverly applied Johnson-Lindenstrauss lemma to project down to multiple lower dimension "buckets" and then query in the bucket in the lower dimensional space at a much lower cost. This method is called Prioritized Dynamic Continuous Indexing (P-DCI) and we compare to them.
- *Space partitioning:* Popular frameworks such as Annoy belong to this class and are used in a commercial recommendation system. These methods are usually not modifiable once the index is constructed, and have a favorable speed-vs-accuracy tradeoff, but construction can be slow. More theoretically grounded approaches such as Cover Trees also belong to this class. We use Annoy and original Cover Trees (Beygelzimer et al., 2006) as representatives from this class for comparison.
- *Neighborhood based methods:* Methods in this category seldom have theoretical analysis, nevertheless, they are typically very fast with decent recall empirically. Although, construction can be very costly and the index is often not modifiable once constructed. The most efficient approximate NN search algorithm according to many open source benchmarks, HNSW (Malkov & Yashunin, 2016), belongs to this category and we select to compare against it.

Note: We do not compare to "Faster Cover Trees" (Izbicki & Shelton, 2015) as unfortunately, we are unable to substantiate several of the results claimed in the paper and can provide counterexamples to some important claims. These claims were "stated without proof" and without them it is not clear why this algorithm will perform well. Furthermore, the experimental results in the paper are poor and is not comparable to simple algorithms (even simple brute-force) on standard datasets such as MNIST (30min vs 2.5min). This paper and its approach is not used in any NN benchmarks and hence, we also choose not to compare to it.

```
import numpy as np
from keras.datasets import mnist
from sklearn.neighbors import NearestNeighbors
(xx, _), (_, _) = mnist.load_data()
xx = xx.reshape(60000, -1)
nbrs = NearestNeighbors(n_neighbors=2,
algorithm='brute', n_jobs=1).fit(xx)
distances, indices = nbrs.kneighbors(xx)
```

Code 1: Simple brute force search on MNIST. Runtime 2.5 min on laptop vs 30 min reported by Faster Cover Trees (Izbicki & Shelton, 2015).

B. Hyper-Parameters Used

We tune each hyperparameter according to the guidelines specified by the packages to maximize performance. We do so in as scientific a manner as possible, while ensuring recall@10 ¿ 0.99, as stated in Line 267. The chosen hyper-parameters are are reported in Appendix A. We elaborate further below:

- RP-Tree/Annoy has two hyper-parameters: `n_tree` and `search_k`. Larger values of either will yield more accurate results, but at the expense of higher build & search time. We do a line search on `n_tree` with default setup of `search_k` until we hit Recall@10 ¿ 0.99. Then we reduce `search_k` to improve runtime till recall does not degrade.
- HNSW/NMSLIB has multiple hyper-parameters, too many to search in a systematic fashion. We followed the guide provided by developers (Page 32, section 4.5.2). The important parameters are `M`, `efCons`, and `efSearch`. Following their recommendation "One way to check if the selection of `efCons` was ok is to measure a recall for `M` nearest

[8] https://erikbern.com/2018/02/15/new-benchmarks-for-approximate-nearest-neighbors.html
[9] https://github.com/erikbern/ann-benchmarks/blob/87f9c99/results/glove-100-angular.png

neighbor search when `efSearch=efCons`: if the recall is lower than 0.9, then there is room for improvement.", we tied `efCons=efSearch` and kept increasing them until we hit Recall@10 ¿ 0.99.

- P-DCI has multiple hyper-parameters, but without any guidance for selecting them. Thus starting from default, we tried our best to select parameters for fastest performance and perturbing each coordinate until they decreased performance.

Final Hyper-Parameters used:

- Cover Tree: It has only one hyper-parameters. Scale was chosen to 1.3 as suggest by Beygelzimer et al. (2006). We use transformation presented in (2.4) to support angular criterion.
- RPTree/Annoy: It has only two hyper-parameters. We had to select `n_tree`=400 and `search_k`=400k. It inherently supports angular criterion.
- HNSW/NMSLIB: It has multiple hyper-parameters. Following the guide provided by developers, we had to select `M`=100, `efCons`=2000, `efSearch`=2000, and all others were default value. It inherently supports angular criterion.
- P-DCI: It has multiple hyper-parameters as well. Following the hints provided by authors, we had to select `num_comp_indices`=10, `num_simp_indices`=20, `num_levels`=3, `construction_field_of_view`=200, `construction_prop_to_retrieve`=1, `query_field_of_view`=400, and `query_prop_to_retrieve`=1. We use transformation presented in (2.4) to support angular criterion.
- SG Tree: It has only one hyper-parameter: scale. Similar to cover tree, we chose it to be 1.3.

C. Detailed Comparison of SG Tree

For comparing various algorithms, we report three metrics:

- **Construction time:** A traditional metric measuring the time taken to build the nearest neighbor search index. Reported in Table 1 and plotted in Figure 6.
- **Average time taken per query:** A traditional metric measuring the average time taken to perform 10,000 queries using unseen points from same dataset to retrieve the 10 nearest neighbors in the index. Reported in Table 2 and plotted in Figure 5.
- **Experience time:** The total time taken for constructing the index and performing m queries to find the 10-NN in the index for each query vector. The results for $m = 1,000$ are reported in Table 3 and plotted in Figure 3. The results for $m = 10,000$ are reported in Table 4.

We would like to emphasize that in terms of construction time, SG Tree is an order of magnitude faster than any other method on all datasets. Even in terms of average query time, SG Tree is competitive to other methods and much faster than brute force search. *This is important for our use case, because artists would want to load their datasets and start exploring and playing with it immediately, rather than waiting for hours.* Also it is much faster than using brute-force search which has no construction time delay. Thus we report "experience time", i.e. the total time taken for constructing the index and performing 1k queries to find the 10-NN in the index for each query vector. This metric better reflects the experience of an user who interactively wants to perform some NN search.

SG Tree is not designed to construct the index once and perform huge number of queries, rather its designed for more exploratory purpose. Moreover, SG Tree allows for insertion and deletion of points. This makes it possible to quickly modify (insert/delete points) the search index. It is useful, for example, when a higher resolution satellite imagery becomes available for certain region. Most of the baselines do not provide this functionality and would require the expensive reconstruction of the index from scratch.

Table 1. Comparison of SG Tree on numerous benchmark datasets in terms of construction time on the task of building NN graph. Approximate NN methods have been marked with * and for such methods the hyper-parameters are chosen so as to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

Dataset		N	D	Construction Time [s]				
				CoverTree	RPTree*	HNSW*†	P-DCI*	SG Tree
Euclidean	Artificial	10K	40	2.38E+01	2.47E+00	5.82E-01	5.34E-01	2.01E-02
	Faces	10K	20	1.01E+00	4.27E+00	1.25E+00	7.00E+00	4.14E-01
	Corel	68k	32	8.84E+01	2.07E+01	3.08E+00	8.67E+00	1.65E-01
	MNIST	70K	784	9.71E+02	8.41E+01	2.28E+01	9.56E+00	8.41E-01
	FMNIST	70K	784	9.01E+02	8.12E+01	1.36E+01	1.03E+01	8.75E-01
	TinyImages	100K	384	3.28E+03	8.55E+01	3.52E+01	1.75E+01	9.78E-01
	CovType	581K	55	3.10E+01	3.10E+02	3.63E+01	2.84E+02	1.90E+00
	Twitter	583K	78	9.74E+03	2.68E+02	4.55E+01	2.95E+02	2.09E+00
	YearPred	515K	90	9.75E+03	2.77E+02	1.10E+02	2.51E+02	1.69E+00
	SIFT	1.00M	128	8.41E+03	6.64E+02	3.44E+02	8.02E+02	2.56E+00
	GIST	1.00M	960	1.94E+04	2.39E+03	2.16E+03	1.01E+03	1.96E+01
Angular	NYTimes	300K	256	1.25E+03	2.35E+02	3.80E+02	7.54E+01	6.29E+00
	GloVe25	1.28M	25	8.47E+03	8.35E+02	3.69E+02	1.03E+03	2.44E+00
	GloVe50	1.28M	50	9.46E+03	8.11E+02	8.16E+02	1.05E+03	3.60E+00
	GloVe100	1.28M	100	1.15E+04	1.06E+03	1.33E+03	1.10E+03	6.15E+01
	GloVe200	1.28M	200	1.85E+04	1.37E+03	2.33E+03	1.14E+03	3.32E+02

Table 2. Comparison of SG Tree on numerous benchmark datasets in terms of average time per query of 10-NN search. Approximate NN methods have been marked with * and for such methods the hyper-parameters are chosen so as to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

Dataset		N	D	Avg Time per Query [s]				
				CoverTree	RPTree*	HNSW*†	P-DCI*	SG Tree
Euclidean	Artificial	10K	40	3.75E-05	6.00E-04	6.30E-05	8.00E-04	1.43E-05
	Faces	10K	20	7.05E-06	6.15E-04	3.80E-05	6.80E-04	5.19E-06
	Corel	68k	32	1.07E-05	6.10E-04	3.83E-05	1.63E-03	1.34E-06
	MNIST	70K	784	1.52E-03	9.38E-04	3.98E-04	1.98E-03	5.56E-04
	FMNIST	70K	784	8.02E-04	6.17E-04	2.20E-04	2.03E-03	5.72E-04
	TinyImages	100K	384	1.58E-03	7.56E-04	5.49E-04	3.36E-03	4.37E-04
	CovType	581K	55	9.54E-06	5.46E-04	5.10E-05	3.89E-03	3.11E-06
	Twitter	583K	78	2.29E-04	5.81E-04	7.97E-05	4.45E-03	7.05E-05
	YearPred	515K	90	1.06E-03	7.01E-04	2.94E-04	5.21E-03	7.30E-04
	SIFT	1.00M	128	3.79E-03	1.22E-03	5.03E-04	6.87E-03	2.80E-03
	GIST	1.00M	960	6.66E-02	6.04E-03	3.07E-03	1.60E-02	6.10E-02
Angular	NYTimes	300K	256	3.92E-03	1.74E-03	2.62E-03	2.92E-03	1.89E-03
	GloVe25	1.28M	25	2.24E-03	3.79E-04	3.79E-04	6.36E-03	2.22E-03
	GloVe50	1.28M	50	5.79E-03	1.56E-03	8.57E-04	7.21E-03	5.69E-03
	GloVe100	1.28M	100	1.05E-02	2.37E-03	1.35E-03	8.66E-03	1.02E-02
	GloVe200	1.28M	200	1.77E-02	3.63E-03	2.44E-03	1.08E-02	1.69E-02

Table 3. Comparison of SG Tree on numerous benchmark datasets in terms of "experience time", i.e. total time taken to build the index and perform 1K queries. Approximate NN methods have been marked with * and for such methods the hyper-parameters are chosen so as to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

Dataset		N	D	Experience Time [s]				
				CoverTree	RPTree*	HNSW*†	P-DCI*	SG Tree
Euclidean	Artificial	10K	40	2.38E+01	3.07E+00	6.45E-01	1.33E-00	3.44E-02
	Faces	10K	20	1.02E+00	4.88E+00	1.29E+00	7.68E+00	4.19E-01
	Corel	68k	32	8.84E+01	2.13E+01	3.12E+00	1.03E+01	1.66E-01
	MNIST	70K	784	9.73E+02	8.50E+01	2.32E+01	1.15E+01	1.40E+00
	FMNIST	70K	784	9.02E+02	8.19E+01	1.39E+01	1.23E+01	1.45E+00
	TinyImages	100K	384	3.28E+03	8.63E+01	3.58E+01	2.09E+01	1.42E+00
	CovType	581K	55	3.10E+01	3.11E+02	3.63E+01	2.88E+02	1.93E+00
	Twitter	583K	78	9.74E+03	2.69E+02	4.56E+01	3.00E+02	2.16E+00
	YearPred	515K	90	9.76E+03	2.78E+02	1.10E+02	2.56E+02	2.42E+00
	SIFT	1.00M	128	8.41E+03	6.66E+02	3.45E+02	8.09E+02	5.36E+00
	GIST	1.00M	960	1.94E+04	2.40E+03	2.16E+03	1.03E+03	8.06E+01
Angular	NYTimes	300K	256	8.47E+03	2.37E+02	3.82E+02	7.83E+01	8.18E+00
	GloVe25	1.28M	25	8.47E+03	8.36E+02	3.70E+02	1.03E+03	4.66E+00
	GloVe50	1.28M	50	9.46E+03	8.13E+02	8.17E+02	1.06E+03	9.29E+00
	GloVe100	1.28M	100	1.15E+04	1.07E+03	1.33E+03	1.10E+03	7.17E+01
	GloVe200	1.28M	200	1.85E+04	1.37E+03	2.33E+03	1.15E+03	3.49E+02

Table 4. Comparison of SG Tree on numerous benchmark datasets in terms of "experience time", i.e. total time taken to build the index and perform 10K queries. Approximate NN methods have been marked with * and for such methods the hyper-parameters are chosen so as to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

Dataset		N	D	Experience Time [s]				
				CoverTree	RPTree*	HNSW*†	P-DCI*	SG Tree
Euclidean	Artificial	10K	40	2.42E+01	8.47E+00	1.21E+00	8.54E+00	1.63E-01
	Faces	10K	20	1.08E+00	1.04E+01	1.63E+00	1.38E+01	4.66E-01
	Corel	68k	32	8.85E+01	2.68E+01	3.47E+00	2.49E+01	1.78E-01
	MNIST	70K	784	9.86E+02	9.35E+01	2.68E+01	2.94E+01	6.40E+00
	FMNIST	70K	784	9.09E+02	8.74E+01	1.58E+01	3.06E+01	6.60E+00
	TinyImages	100K	384	3.29E+03	9.31E+01	4.07E+01	5.12E+01	5.35E+00
	CovType	581K	55	3.11E+01	3.16E+02	3.68E+01	3.23E+02	2.17E+00
	Twitter	583K	78	9.74E+03	2.74E+02	4.63E+01	3.40E+02	2.80E+00
	YearPred	515K	90	9.77E+03	2.84E+02	1.13E+02	3.03E+02	8.99E+00
	SIFT	1.00M	128	8.45E+03	6.77E+02	3.49E+02	8.70E+02	3.06E+01
	GIST	1.00M	960	2.00E+04	2.45E+03	2.19E+03	1.17E+03	6.30E+02
Angular	NYTimes	300K	256	1.29E+03	2.52E+02	4.06E+02	1.05E+02	2.52E+01
	GloVe25	1.28M	25	8.49E+03	8.48E+02	3.73E+02	1.09E+03	2.46E+01
	GloVe50	1.28M	50	9.46E+03	8.27E+02	8.24E+02	1.12E+03	6.05E+01
	GloVe100	1.28M	100	1.15E+04	1.09E+03	1.34E+03	1.18E+03	1.64E+02
	GloVe200	1.28M	200	1.85E+04	1.41E+03	2.35E+03	1.25E+03	5.01E+02

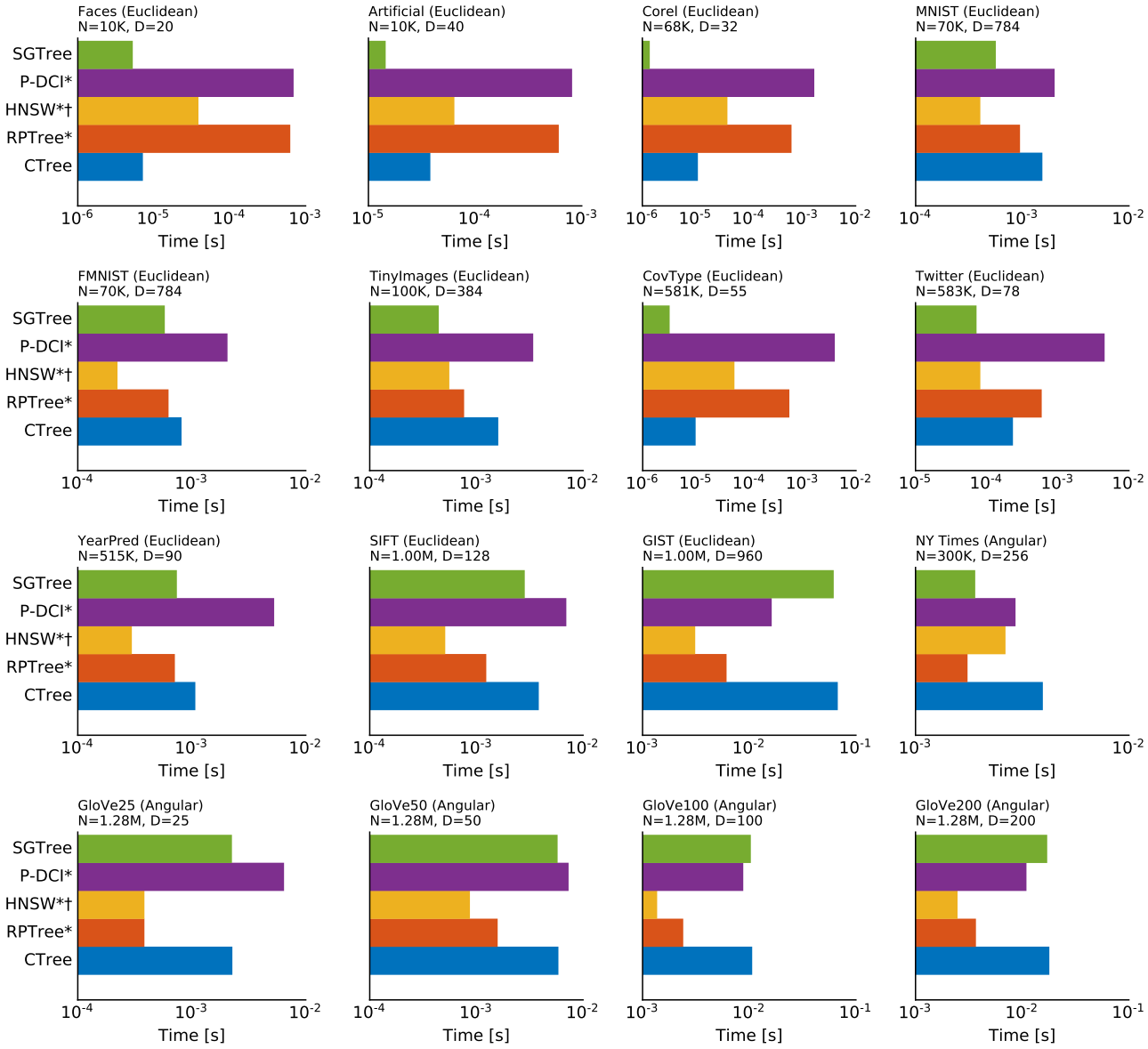


Figure 5. Comparison of SG Tree on the benchmark datasets in terms of "Query Time", i.e. average time to perform 10-NN queries. Approximate NN methods are marked with * and for such methods the hyper-parameters are chosen to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

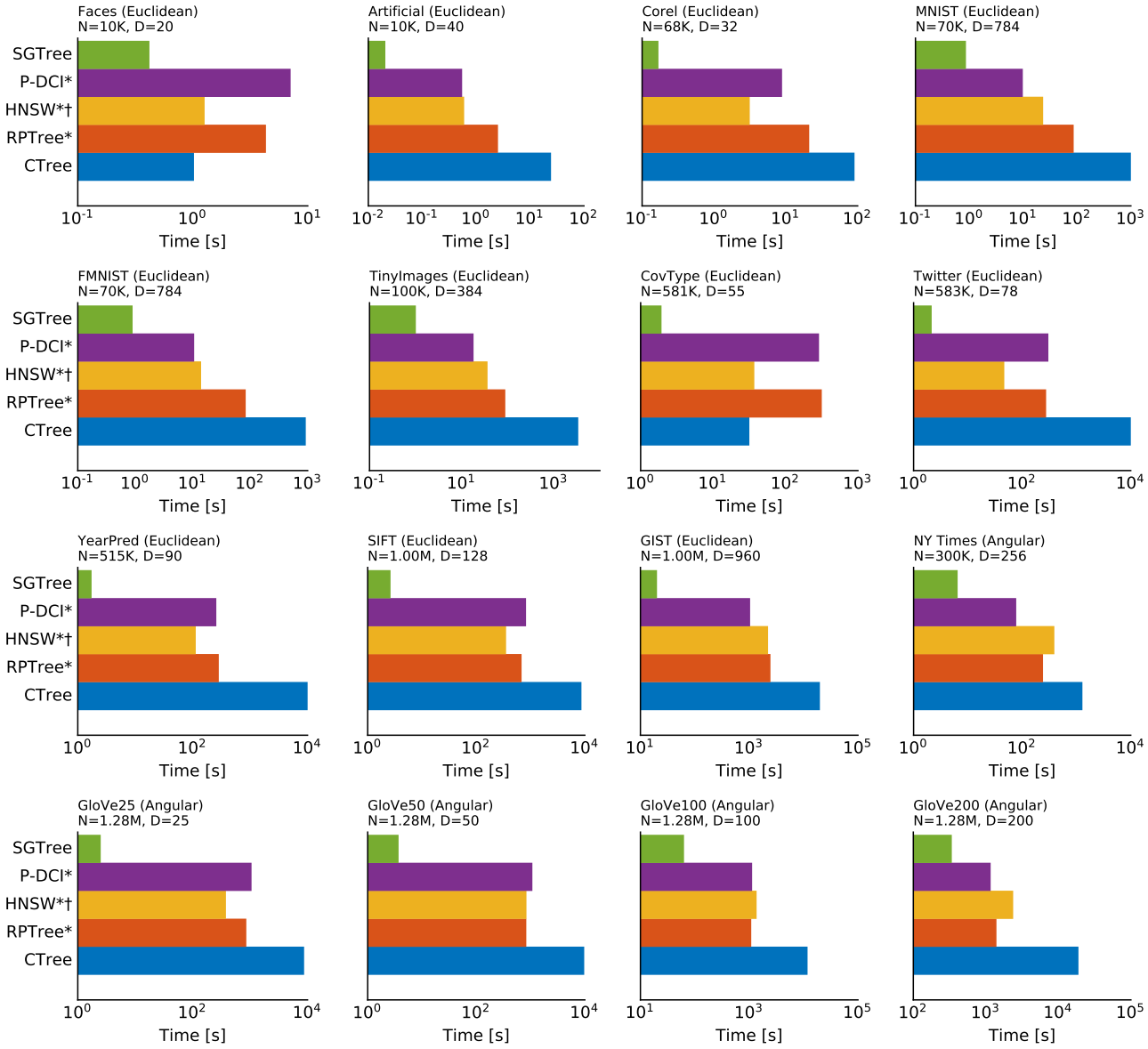


Figure 6. Comparison of SG Tree on all benchmark datasets in terms of "Build Time", i.e. construction time on the task of building NN graph. Approximate NN methods are marked with * and for such methods the hyper-parameters are chosen to produce Recall@10 ≥ 0.99 on all the datasets. Non-modifiable data structures are marked with †.

D. More Details from Section 2.5

In this section we provide the more details about our assumptions and proofs for our theoretical results presented in Section 2.5.

D.1. Time-Marginalized Coalescent (TMC) Process

TMC is a popular generative process for modelling hierarchical data (Kingman, 1982b;a; Boyles & Welling, 2012; Teh et al., 2008; Vikram et al., 2019). TMC defines a distribution over binary trees. In this model we have binary tree together with time labels associated with each node, i.e. formally we have a triplet (V, E, T) where V is the set of nodes, E is the set of edges, and the time labels is given by a function $\tau : V \to [0,\, 1]$ where we denote $t_v = \tau(v)$.

Sampling from Time-Marginalized Coalescent occurs in three steps:

1. Sample tree structure: Start with k leaf nodes and add them to V. Pick a pair of nodes uniformly at random, merge them to create a new node. Add the new node to V and the edges to E. Repeat the process until only one vertex is left, which is the root of the tree.

2. Sample time labels (using stick-breaking process): Start with a stick of unit length at the root of the binary tree. Set the time label for root as 0, i.e. $t_{\text{root}} = 0$. At each internal node v, duplicate the current stick into two sticks, assigning one to each child. Then, sample a Beta random variable $\beta_{\text{left-child}}, \beta_{\text{right-child}} \sim \mathsf{Beta}(\gamma, 1)$ for each of the two sticks. This will be the proportion of the remaining stick attributed to that branch of the tree. Repeat this process until th leaf nodes are hit. The time label for leaf nodes are set to be 1. To summarise the time labels are sampled as follows:

$$t_v = \begin{cases} 0 & v = v_{\text{root}}, \\ 1 & v \in V_{\text{leaf}}, \\ t_{\pi(v)} + \beta_v(1 - t_{\pi(v)}) & v \in V_{\text{int}} \setminus \{v_{\text{root}}\}, \end{cases} \tag{1}$$

   where $\beta_v \sim \mathsf{Beta}(\gamma, 1)$ for $v \in V$. These time labels encode a branch length $t_v - t_{\pi(v)}$ for each edge $e = (\pi(v), v) \in E$.

3. Sample points for nodes: We start with root at time $t = 0$ from a place randomly sample from a Normal distribution $z_{v_{\text{root}}} \sim \mathcal{N}(0, I)$. At each internal node, we split into two independent Wiener processes, which independently evolves for times $t_{\text{child}} - t_{\text{parent}}$. After this time, we say the child node is reached and the process is repeated with a new independent Wiener processes being instantiated from current position until all processes reach the leaves (i.e. $t = 1$). To summarize, for each vertex $v \in V$ a point z_v is sampled according to a Normal distribution centered at its parent's location with variance proportional to the branch length,

$$z_v | z_{\pi(v)} \sim \mathcal{N}(z_{\pi(v)}, (t_v - t_{\pi(v)})I), \quad v \in V \setminus \{v_{\text{root}}\} \tag{2}$$

4. Sample data points: Data points can be generated in iid fashion. To generate a new point, one simply walks randomly from the root to the leaf, and then outputs a point according to the Normal distribution whose parameters are determined by the leaf. In other words, all k leaf nodes can be considered as cluster centers and points being sampled from it.

$$\begin{aligned} \ell &\sim \text{RandomDescent}(V, E) \\ x | z, \ell &\sim \mathcal{N}(z_\ell, (t_\ell - t_{\pi(\ell)})I) \end{aligned} \tag{3}$$

Note that if $\gamma > 1$, then variance (or stick length $s_v = t_v - t_{\pi(v)}$) as we go down the tree keeps decreasing exponentially fast with high probability. One can quickly check this behaviour for a node v at level l in expectation as follows:

$$\begin{aligned} \mathbb{E}[s_v] &= \mathbb{E}\left[\beta_v \prod_{u \in \mathcal{P}_v} (1 - \beta_u)\right] \\ &= \mathbb{E}[\beta_v] \prod_{u \in \mathcal{P}_v} (1 - \mathbb{E}[\beta_u]) \quad \because \text{all } \beta_i \text{ are independent} \\ &= \frac{\gamma}{(1+\gamma)^l} \end{aligned} \tag{4}$$

Here $\mathcal{P}_v$ denotes the set of vertices in the path from root to v (excluding v).

D.2. Recursive GMM

We consider a variant of TMC process called recursive GMM (rGMM), where the tree is not limited to be binary. Also instead of assigning variance to the nodes of the tree in a stick-breaking, we assume it decreases by a constant discount factor at every level. Note that stick-breaking process also exhibits very similar behaviour with high probability. This simplifying assumption makes the analysis of SG Tree tractable, without much change in richness of the hierarchies being represented by the model. The rGMM generative process is outlined in Algorithm 3.

We first create a tree structure whose leafs denote the cluster centers. To generate a new point, one simply walks randomly from the root to the leaf, and then outputs a point according to the Gaussian whose parameters are determined by the leaf.

Algorithm 3 Recursive GMM (γ, c, T)

function RGMM(Depth t, Mean μ, Variance σ^2)
  $S \leftarrow \{\}$
  if $t = 0$ **then**
    $S \leftarrow (\mu, \sigma^2)$
  else
    for $i = 1 \rightarrow c$ **do**
      $\mu_i \sim \mathcal{N}(\mu, \sigma^2)$
      $S \leftarrow S \cup \{\text{RGMM}(c, t-1, \mu_i, \sigma^2/(1+\gamma))\}$
    end for
  end if
  return S
end function
function INITIALIZE
  PARAMS $\leftarrow$ RGMM$(T, 0, I)$
  save PARAMS
end function
function SAMPLE
  $A \leftarrow$ PARAMS
  for $t = 1 \rightarrow T$ **do**
    Pick uniformly at random child $a \sim A$
    $A \leftarrow a$
  end for
  $\mu, \sigma^2 \leftarrow A$
  $x \sim \mathcal{N}(\mu, \sigma^2)$
  return x
end function

D.3. Proofs of Lemma in Seciton 2.5

Below we provide proof sketches for Lemma 4 and Lemma 5.

Proof Sketch of Lemma 4. The proof contains two parts: The first part is that the SG Tree will contain subtrees that are highly correlated with each cluster generated by the rGMM bounds. The second part will show that any new query point will first search the closest pair quickly and will only traverse $O(c)$ nodes at each level before pruning them out.

To show that the SG Tree creates subtrees who are highly correlated, we proceed by induction on the level i. At each i, we will argue that the sets will be highly correlated with the clusters in rGMM model, if the previous levels were highly correlated. Let $x_1, \ldots, x_c$ be the centers generated at level i, who have a common parent p. Observe that with high probability, for all pair of points $\max_{i,j \in [n]} ||x_i - x_j|| \geq c\sqrt{\gamma_i n}$. Furthermore all subsequent subchildren will lie within each $\sqrt{\gamma_i n}$ of x_i due to the decrease in the variance in rGMM. We also know that each point in these clusters will contain at least one point, since we sample $n \geq c^T$ points to insert into the tree, where $T = O(\log n)$.

Consider any level l satisfying $\gamma^l \geq 2 \cdot \sqrt{(1+\gamma)^i n}$. Observe that all the nodes belong to the same cluster at this level.

We now consider the highest level l where $\gamma^l \leq T\sqrt{d}$. Let $c_1, \ldots, c_t$ be the set of children to the (single) root at this level. By the local separation property, we have that $||c_i - c_j|| \geq \gamma^l \geq T\sqrt{d}$.

When $T \geq \sqrt{\log n}$ we know that with probability $(1 - 1/n)$, all points lie within $T\sqrt{d}$ from the cluster center due to standard Chernoff bounds. Since all the children at each level are chosen to be at least γ^l apart from each other, we know that there is at least one element from each cluster in the set of children.

Due to our assumption on the input order, we know that each of these children will partition the space according to their Voronoi Partition. Since $T \geq \Omega(\sqrt{\log n})$, each node in a cluster is closer to every point in its own cluster and farther from all the other cluster centers. Therefore, each node will be belong to a child from its own cluster.

Now we show the second point. Let the query q be sampled from one of the centers generated by the rGMM distribution. WLOG, let us say that μ_T, σ_T. Futhremore, let (μ_i, σ_i) denote the mean and variance of the i^{th} node on the root to leaf path.

We will show that the query algorithm will find the right cluster with high probability at each level. Once it reaches the bottom node, then it gets it correct. To process each level takes $O(c)$ time. In particular, we will bound

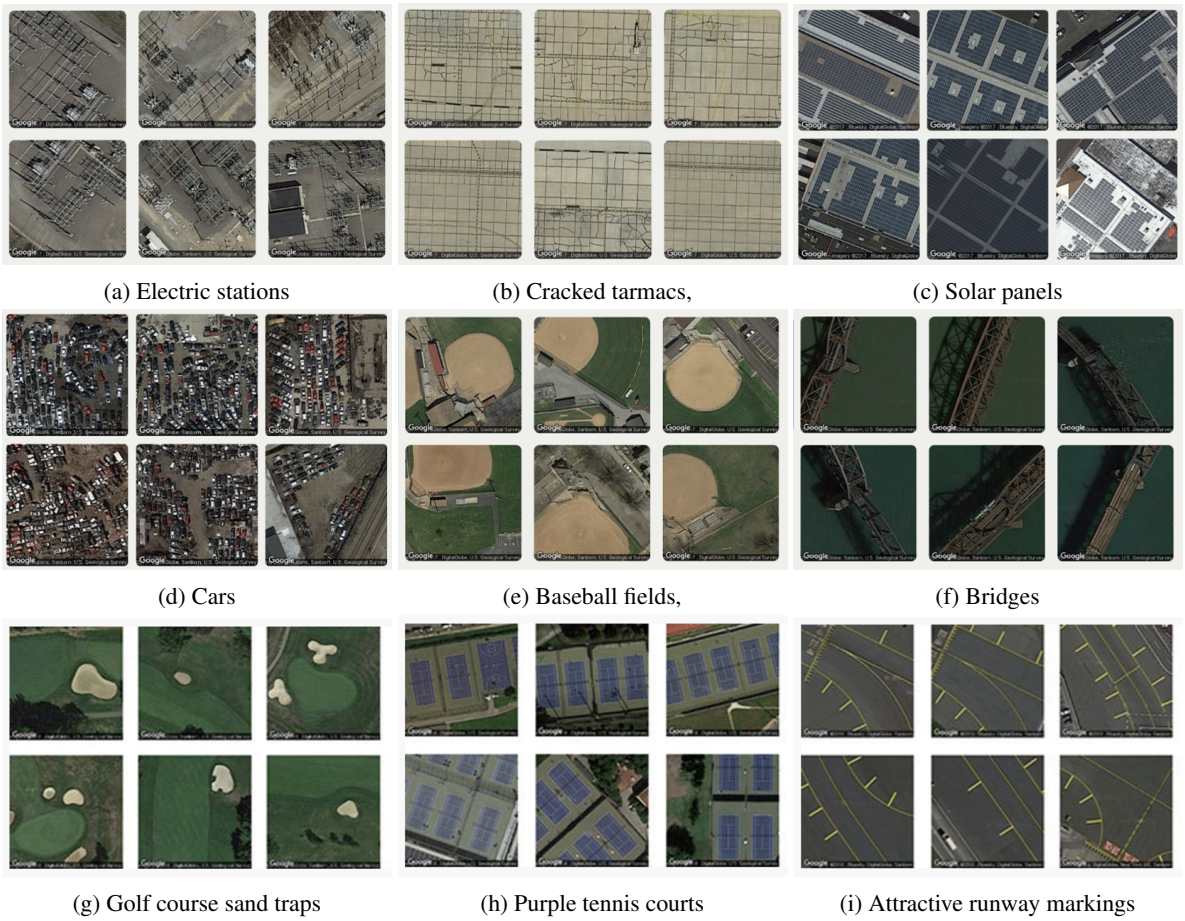
$$d(q, \tilde{\mu}_i) \geq \sum_{j=i}^{T-1} d(\mu_j, \mu_{j+1}) + d(q, \mu_j)$$

We will bound the latter part by the sum of the variances which is geometrically decreasing. □

Proof Sketch of 5. Suppose we look at some node p and its children $c_1, \ldots, c_t$, with associated variances $\sigma_1, \ldots, \sigma_t$. We know that any sample p will lie in a radius of size $t\sqrt{\sigma_i n}$ from the node c_i with probability at least $O(e^{-t^2})$. Since we have inserted at least c^T points, we expect there to be atleast one point from each cluster in the first $\log T$ levels with high probability. Since there are c^T nodes in the top T levels and each has a probability of $O(e^{-t^2})$ of failing, we can simply take a union bound to show that T levels, there will be no new nodes being created. □

E. Terrapattern Results

Some "interesting" example searches patterns are shown in Figure 7. For our purposes, "interesting" features are anthropogenic or natural phenomena that are not only socially or scientifically meaningful, but also visually distinctive–thus lending themselves ideally to machine recognition. Examples could include things like animal herds, electric stations, factories, destroyed homes, or sports ground. Many other patterns await discovery. It is important to point out that the TerraPattern system was not trained on any of the categories shown, but instead recognizes them because of their common visual features. There are a number of burgeoning, visually consistent, and in many cases worrisome phenomena which future versions of TerraPattern could be useful in tracking, e.g. concentrated animal feeding operations, uranium mill tailings deposits, Siberian methane blowholes, which are arising due to global warming, and megafauna poaching. The TerraPattern project is only a prototype-especially in its scale-and we feel we have only scratched the surface of what is possible.



(a) Electric stations (b) Cracked tarmacs, (c) Solar panels

(d) Cars (e) Baseball fields, (f) Bridges

(g) Golf course sand traps (h) Purple tennis courts (i) Attractive runway markings

Figure 7. Some examples of interesting patterns found by our method. Link to TerraPattern UI for the results anonymized.

F. Related Works

F.1. Nearest Neighbor Search

NN search has been an active area of research in many communities in computer science. We will only mention the works directly related to this paper as a full survey of this rich area is well beyond the scope of this paper. Clarkson (1999) was the first to study NN search in high dimensions where he studied points that satisfied a certain sphere packing assumption. Karger & Ruhl (2002) subsequently introduced the notion of expansion constant and studied data-structures that performed well under this assumption. Subsequently Krauthgamer & Lee (2004) provided data structure Navigating Nets that had a strong theoretical properties and worked on metrics with constant doubling dimension. A simpler data-structure Cover Trees

was proposed by Beygelzimer et al. (2006) which also performed well on metrics with small expansion constants. Har-Peled & Mendel (2006) came up with Net Trees to address approximate nearest neighbors on doubling metrics. Recently, Jahanseir & Sheehy (2016) showed that Net Trees captured Cover Trees with a particular set of parameters. Another interesting data structure with good theoretical properties was given by (Dasgupta & Sinha, 2013) which used the idea of random projections to quickly narrow down the search size.

Despite significant progress in theoretical NN search, the bulk of highly engineered software used in practice has evolved tangentially. They primarily focus approximate NN search, which are sufficiently useful for many practical problems, such as in big machine learning pipelines. The algorithms for approximate NN search can be categorized as:

- Locality sensitive hashing (LSH): It maps a high-dimensional point to a low-dimensional point via a set of appropriately chosen random projections. There are theoretical guarantees on query result quality, efficiency, and memory requirements. Software packages like FALCONN (Andoni et al., 2015) and QALSH (Huang et al., 2015) implement variants of this idea. Typically they are modifiable and fast, but have relatively lower recall.
- Space partitioning: The space is partitioned in a hierarchically manner using a tree data structure. Popular frameworks such as Annoy and FLANN (Muja & Lowe, 2009) belong to this class and are used in a commercial recommendation system. These methods are usually not modifiable once the index is constructed, and have a favorable speed-vs-accuracy tradeoff, but construction can be slow.
- Neighborhood based methods: A proximity graph consisting of neighborhood information for each individual data point is maintained. At query time heuristics are used to navigate the proximity graph. Methods in this category seldom have theoretical analysis, nevertheless, the are typically very fast with decent recall empirically. Although, construction can be very costly and the index is often not modifiable once constructed. The most efficient approximate NN search algorithm according to many open source benchmarks, HNSW (Malkov & Yashunin, 2016), belongs to this category, albeit HNSW has no theoretical guarantees.

For a more detailed survey, readers are encouraged to refer to Li et al. (2016). Since none of the existing tools fit our requirements of a fast, modifiable NN search algorithm with exact or high recall, we develop our own method.

F.2. Satellite Imagery

Access to satellite imagery, especially as it can be interpreted through the lens of machine intelligence, is currently controlled by a select few: state-level actors and (increasingly) multinational businesses. Once the exclusive domain of top-secret military surveillance, high-resolution satellite imagery has recently become heavily corporatized in public domain. In this section, we highlight a few projects which we consider to be powerful and inspirational illustrations of the democratization of machine intelligence for satellite imagery.

Corporate: At the forefront of this shift of availability of satellite images are companies like Orbital Insight, Remote Sensing Metrics and Genscape, which apply machine learning algorithms to satellite imagery in order to sell "actionable intelligence" to hedge funds and other market speculators. For example, in their "US Retail Traffic Index", RS Metrics monitors the number of cars in retail parking lots, in order to estimate the quarterly performance of big-box stores before those results have been released. Similarly, Orbital Insight's "World Oil Storage Index" consists of daily estimates of the

Figure 8. Left: Parking around shopping malls indicating performance of stores. Right: Oil storage tracking can indicate business performance of the oil company.

Figure 9. Left: Identifying illegal propping up of gold mines. Right: Identify illegal logging roads in Amazonian forest.

amount of oil held in 20,000 storage tanks-intelligence derived from the size of shadows on the interiors of tanks with floating lids.

Environmental: At the forefront of environmental efforts is the non-profit organization, Monitoring of the Andean Amazon Project (MAAP), which uses satellite imagery and computer vision to analyze the Amazonian rainforest. In some of their best-known work, supported through the Planet Labs Ambassadors Program, MAAP has successfully detected illegal gold mines, as well as illegal logging roads, which are key precursors to deforestation. Other environmental initiatives have used related techniques to, for example, bust illegal fishing operations.

Humanitarian: At the Harvard Humanitarian Initative's "Signal Program on Human Security and Technology", a series of influential projects directed by Nathaniel Raymond has used satellite imaging to investigate war crimes, genocides, and other atrocities. Raymond is among the most outspoken advocates for the use of geospatial intelligence by human rights groups and other non-governmental organizations (NGOs). In one project, Raymond and his team used machine learning techniques to automatically identify Sudanese straw-hut dwellings, known as tukuls, in satellite imagery. Their team's tukul-detector was able to successfully distinguish intact tukuls from ones which had been razed–an excellent proxy for detecting mass killings, in a part of the world where on-the-ground journalism is exceptionally risky.

In another humanitarian project, data scientists from DataKind.org collaborated with members of GiveDirectly, an NGO which gives microgrants to impoverished people in developing nations. In order to know where to focus their efforts, the team developed software to analyze the ratio of straw roofs to metal roofs in each of the districts of a Central African country. This ratio proved to be a good proxy for estimating the relative wealth of each of the districts, for a country otherwise lacking in census data of this sort.

Wildlife: The combination of satellite imaging and machine vision has also had a major impact on our ability to track animal populations. For example, one team of scientists were able to track Antarctic penguin populations and take measurements of their diets by observing their poo from space. Another team of scientists was able to locate and count families of Southern Right Whales. In another fascinating discovery, Dr. Sabine Begall, a professor of Zoology at the University of Duisburg-Essen, has discovered that ruminants have a previously undiscovered geomagnetic sense essentially,

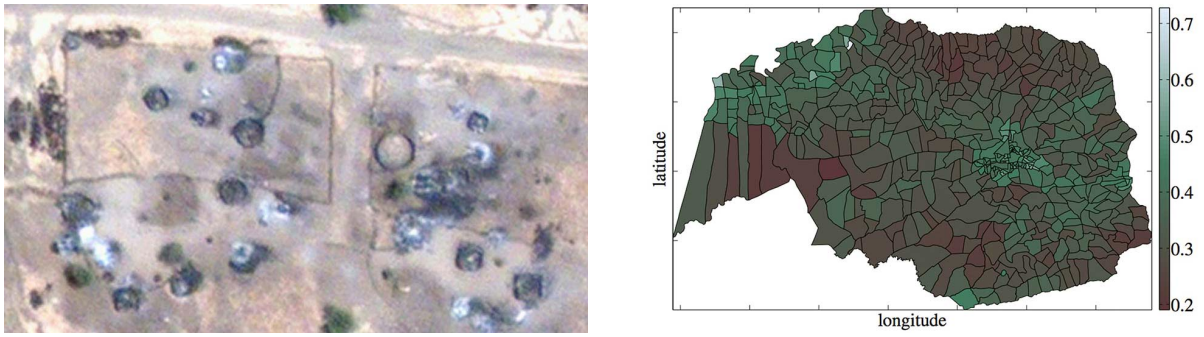


Figure 10. Left: Broken straw in satellite image indicates possibility of mass killings. Right: Counting straw vs metal roof ratio can indicate wealth of the district

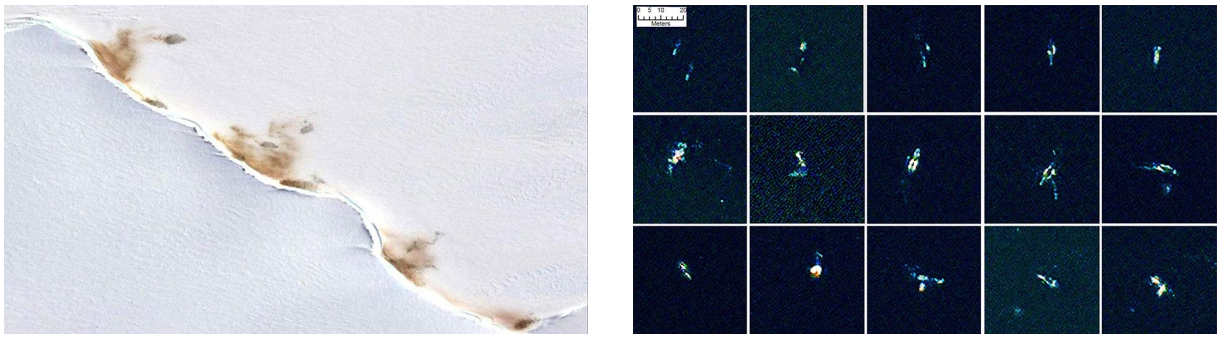


Figure 11. Left: Observing penguin poo from satellite image can track penguin flocks. Right: Detect and track rare species of Southern Right Whales.

that grazing cows align themselves with the earth's magnetic field.

Arts: The arts have the power to provide insights of an altogether different sort. A number of artists have employed various forms of human and/or machine intelligence to the domain of satellite imagery, in order to produce projects that inform, provoke, entertain, and delight. An excellent example of this is the project "Aerial Bold" by Benedikt Groß and Joseph Lee, which is a typeface wholly constructed from letterforms found in satellite imagery. Whereas Groß and Lee use a mixture of crowdsourcing and automated detection, artist Jenny Odell uses a more personalized, curatorial approach in her Satellite Collections project (2009-2011), in which parking lots, silos, landfills, waste ponds are compiled into typological collages. Of her work, Odell writes that "The view from a satellite is not a human one, nor is it one we were ever really meant to see. But it is precisely from this inhuman point of view that we are able to read our own humanity, in all of its tiny, repetitive marks upon the face of the earth. From this view, the lines that make up basketball courts and the scattered blue rectangles of swimming pools become like hieroglyphs that say: people were here."

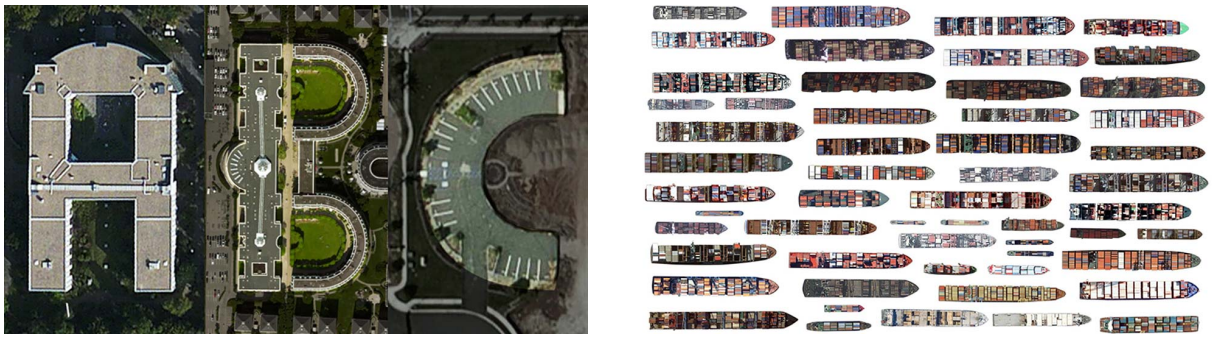


Figure 12. Left: Letterforms found in satellite imagery. Right: Collages of container yards.

In many of the examples discussed above, researchers developed bespoke visual detectors that were tightly tuned and customized for specific problems. The techniques used in Terrapattern portend a new form of highly generalized detector which can be used in searches by relative laypersons. In this new workflow, it is only important that the patterns of interest are visually consistent enough for algorithmic detection and analysis.