# Hyper-Parallel Approximate Nearest Neighbor Search

Lars Lien Ankile[1], Tanner Marsh[1], Sree Harsha Tanneru[1], and Sebastian Wiesshaar[1]

[1]Institute for Applied Computation Harvard University

May 6, 2023

**Abstract**

This project optimizes an approximate nearest neighbor (ANN) search algorithm by leveraging parallelization techniques using OpenMP and MPI libraries. ANN search is a fundamental problem in many fields, including machine learning, computer vision, and information retrieval, and efficient algorithms are crucial for many real-world applications. The project will use OpenMP to parallelize the algorithm at a shared-memory level and MPI to parallelize the algorithm across multiple nodes in a cluster. The parallelized algorithm will be evaluated on benchmark datasets and compared to existing implementations. The expected outcome is a faster and more scalable ANN search algorithm that can be used in various applications.

## 1 Background and Significance

K-Nearest neighbor search is a problem in computer science that involves finding the nearest neighbors of a given query point in a high-dimensional vector space. The exact solution to this problem can be computationally expensive and sometimes infeasible for large datasets. This computational expense stems from the fact that for every query, one has to search through all of the existing data points, resulting in linear O(n) time complexity. For many modern applications, like song recommendations on Spotify, the number of vectors is ∼100 million. Therefore, approximate methods are needed to solve this problem efficiently. An Approximate Nearest Neighbour (ANN) search algorithm aims to find a good approximation to the true nearest neighbors by trading off some accuracy and up-front index-building for inference speed.

This problem is relevant for many vector comparison tasks. For example, collaborative filtering relies heavily on vector similarity comparison in recommender systems. For large online stores with millions of articles, like Amazon or Walmart, exact solutions would result in unreasonable waiting times. Furthermore, there are a myriad of other applications. As mentioned before, Spotify relies heavily on ANN for song recommendations [3], but also medical images are classified with KNN [5], and Facebook developed an approximate nearest neighbors library to search multi-modal data [2].

## 2 Scientific Goals and Objectives

The scientific goal for this project is to understand how ANN search can leverage high-performance computing infrastructure to handle modern recommendation tasks to reduce waiting times for users.

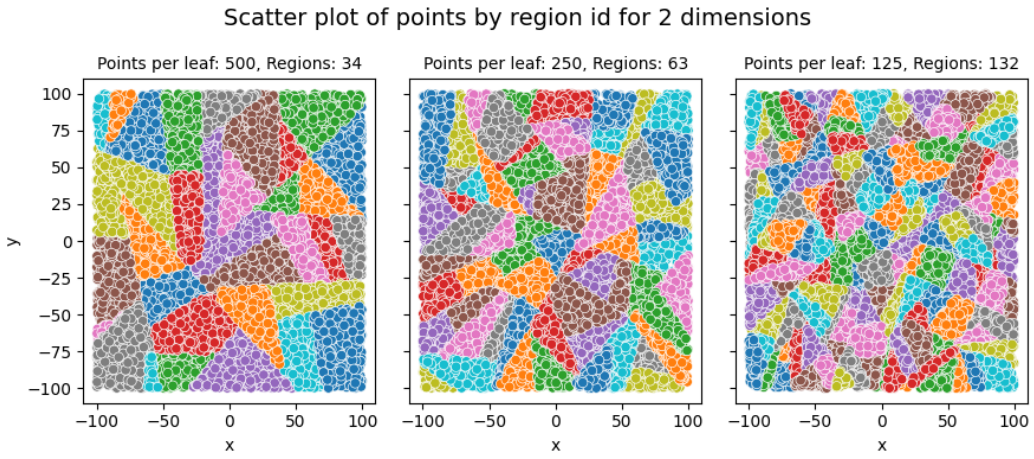


Figure 1: From left to right, a visualization of how the vector space is subdivided into iteratively smaller regions as we go deeper into the search tree and make hyperplanes that split the regions. Smaller regions make for faster inference because you have fewer points to consider, but that could lead to less accurate results.

This project interested us because of the practical relevance and the unique challenges search-tree building presents. The skills and techniques we develop by parallelizing the ANN index-building algorithm will be transferable to many applications, like graph partitioning, hierarchical applications such as molecular dynamics and n-body simulations, and databases [1]. Another interesting recent development is the growth of vector databases (e.g., Pinecone and Weaviate) as large language models and text embeddings become increasingly widespread. Also, we are interested in the performance gains using 32 cores per node which is eight times more than the standard four cores per CPU.

## 3 Algorithms and Code Parallelization

Our implementation consists of (i) building the search index and (ii) the inference method.

1. **Building the search index** involves precomputing the hyperplanes and building the search tree from a large vector dataset. The hyperplanes are a set of linear separators that divide the vector space into increasingly fine subspaces. The search tree is a data structure that stores the hyperplanes at every decision point in the tree and the data points that belong to each terminal region (the leaf nodes in the tree). To build a single tree, we divide the processing of vectors between threads with OpenMP. For better accuracy, several trees are combined into a forest of trees, and we split the tree building across nodes using MPI. Each node will be tasked with the creation of a single tree. $R$ represents the number of nodes, and $T$ is the number of trees in the forest.

2. **The inference method** returns the $k$ nearest neighbors in the search index for a given query vector. The nearest neighbors are the data points most similar to the query vector. The similarity is measured using a distance metric, typically the Euclidean distance. When the search index is built, this part is relatively straight-forward and computationally inexpensive. In short, one passes the query point to each of the $T$ trees, where the point will traverse down the $\sim \log_2(N)$.

## Data Structures

In the following sections, we will first define the abstractions in our implementation, and then dive deeper into the methods, design choices and ways to parallelize.

1. **Hyperplane:** A hyperplane is a plane in $d$ dimensions, represented with a $d$-sized array of floating points and a plane offset.

```
// this defines a plane
struct hyperplane
{
    // constant value of a hyperplane
    float offset;
    // normal vector of the hyperplane
    float *normal;
};
```

2. **SearchTree:** The search tree class constructs the tree, and also maintains the set of hyperplanes and the data points that belong to each terminal region.

```
class SearchTree
{
public:
    SearchTree(int n_points, size_t dimension,
    const float **full_data, int n_threads, int K);
    ~SearchTree();
    // part of code omitted for brevity
}
```

3. **SearchForest:** A search forest is a group of search trees. We create multiple search trees to increase the correctness of our approximate inference algorithm. As there the trees are created stochastically, increasing the forest size reduces the bias of our predictions.

## Algorithm

### Part 1. Building the Search Index

1. **Initial State**

   At each level in the tree, we need to maintain a list of active regions, the hyperplane corresponding to each region, and the region which each point belongs to. When we go to the next level, we update this list of active regions and the mapping for each data point.

2. **Sampling points from active regions**

   For each active region, we need to sample two points at random to construct a hyperplane to further divide regions into smaller regions. We use 'std::uniform_int_distribution' class template to uniformly sample two points in every region.

3. **Construct hyperplanes for all active regions**

   After sampling two points from every region, we need to construct a hyperplane. This is straightforward and embarrassingly parallel, hence we exploit OpenMP for computing hyperplanes for regions. Once the hyperplane is constructed, it is stored in the search index (omp critical is used to prevent data races here).

```
        // create hyperplanes in parallel for each of the active regions
#pragma omp parallel for
        for (auto region_id : active_regions)
        {
            make_hyperplane_(sampled_points[region_id],
            hyperplanes[region_id]);
            // associate region with hyperplane
#pragma omp critical
            {
                region_hyperplane_map.insert(
                std::make_pair(region_id, &hyperplanes[region_id]));
            }
        }
```

4. **Classify points**

   After constructing the hyperplanes for all active regions, we need to classify which side of the hyperplane does every point lie on. At each level in the tree, we need to traverse the non-terminal data points and compute the region that each point belongs to. One obvious way to parallelize this process is to assign each region to a thread and synchronize at the end of every level. However, this approach will have significant thread imbalance. For example, on a node with 32 cores, at the first level, there will be only one region, so only one core will be utilized, while the rest of 31 cores will be idle. Furthermore, as we go deeper, some threads may terminate sooner than others, as each region can have a different number of data points. This will also lead to thread imbalance.

   To address this issue, we split all the non-terminal nodes equally amongst the threads. We also store the region ID for each data point. This allows us to parallelize the traversal of the tree using OpenMP without incurring any thread imbalance.

```
    // classify points in parallel
    #pragma omp parallel num_threads(n_threads)
    {
        // my thread number
        int tid = omp_get_thread_num();
        // the first term of chunk
        int my_first = tid * chunk_size;
        // the last term of chunk
        int my_last = my_first + chunk_size;
        //part of code omitted for brevity
        classify(my_first, my_last, tid);
    }
```

Moreover, In the "signed_distance" kernel, we use '#pragma omp simd reduction' directive for SIMD instructions to perform vectorized computations on a loop.

```
inline float SearchTree::signed_distance(Hyperplane &hp,
const float *point)
{
    float distance = hp.offset;
    #pragma omp simd reduction(+ : distance)
    for (int d = 0; d < D; d++)
    {
        distance += hp.normal_vec[d] * point[d];
    }
    return distance;
}
```

5. **Reduce the region counts**

   As discussed in step 1, each thread operates on a chunk of data. After each thread computes the region that its data points belong to, we want to do a sum reduction to identify data counts for every region. This is a synchronization barrier.

6. **Identify terminal regions**

   If a region has fewer than $k$ data points ($k$ denotes the leaf size), we stop splitting it, and mark the node as a terminal node. The corresponding data points are not processed further.

7. **Move dead points to the end**

   For the terminal nodes identified in the previous step, we don't want to split them further, so we move them to the end of the array by swapping.

8. **Update active regions**

   (i) All currently active regions should be marked inactive

   (ii) Any region that was just reduced should become an active region if it has more than $k$ points (i.e; if it's not a terminal region)

   (iii) If there are no active regions, search index building is complete and we terminate the algorithm

9. **Prepare for next iteration**

   We swap the region id mapping from previous set of region ids to the new set of region ids.

10. **Parallelization of the forest**

    As previously mentioned, we create multiple trees to improve correctness of the solution. This is an embarassingly parallel problem, which makes it trivial to use MPI for parallelism.

```
SearchForest::SearchForest(int n_trees, int n_points,
size_t dimension, const float **full_data, int max_threads, int K)
    : n_trees_(n_trees)
{
    // Initialize MPI with thread support
```

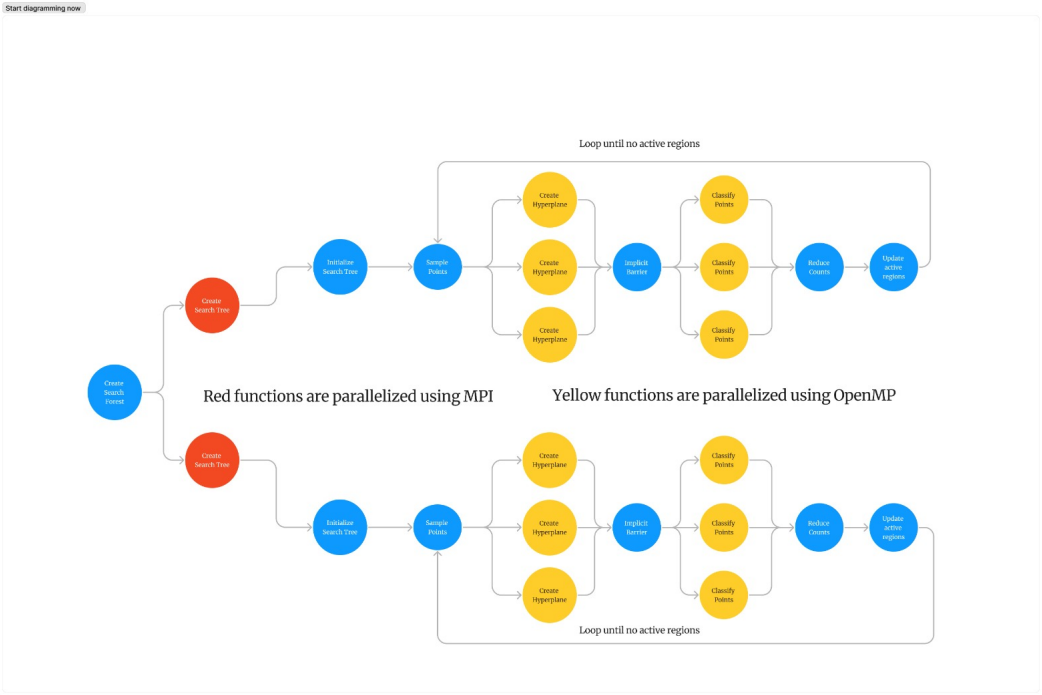


Figure 2: Schematic overview of search index building algorithm with indication of parallel regions

```
    MPI_Init_thread(nullptr, nullptr, MPI_THREAD_FUNNELED, nullptr);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank_);
    MPI_Comm_size(MPI_COMM_WORLD, &size_);

    // Create and distribute trees across MPI ranks
    for (int i = 0; i < n_trees; ++i)
    {
        if (i % size_ == rank_)
        {
            SearchTree* new_tree = new SearchTree(n_points, dimension,
            (const float **)full_data, max_threads, K);
            // part of code omitted for brevity
        }
    }
}
```

A schematic overview of the algorithm can be found in Figure 3. The figure shows which elements of the code are run in parallel and which serially.

**Part 2. Inference**

There are two parts to the inference method. First is traversing the tree to find the relevant terminal regions, and secondly, finding the nearest neighbours within a terminal region.

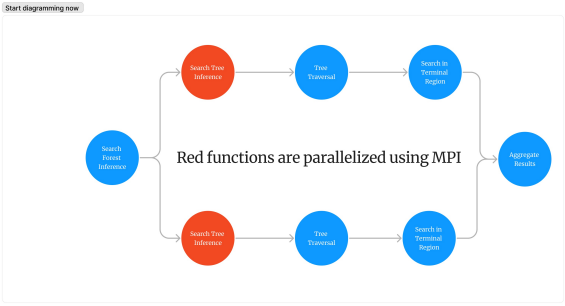


Figure 3: Schematic overview of inference algorithm with indication of parallel regions

1. **Tree Traversal** The search algorithm traverses through the tree and at each node decided whether to use the left or right child node for the next step. If at a node, the number of required neighbours is more than the nodes in that region, we search on both left and right sub regions.

2. **Calculate distance** When we reach a leaf node, all available points in that node are used to find $k$ nearest neighbors. We calculate the distance for each point in the leaf to the input node and use a priority queue to sort them based on distance and subsequently return the top $k$ nearest points.

3. **Parallelization** Searching across trees in the forest is an embarassingly parallel problem, just like building the forest, which makes it trivial to use MPI for parallelism.

### Validation, Verification

In this study, we aim to assess the performance of our proposed algorithm for approximate nearest neighbor (ANN) search, in comparison to two other implementations: a k-nearest neighbor (KNN) implementation provided by scikit-learn [4], and our own serially implemented ANN algorithm. Our comparison will be conducted on multiple vector sizes, with the goal of evaluating the performance of our algorithm as the dimensionality of the data increases.

Scikit-learn is a widely-used library for machine learning algorithms in Python, including a KNN implementation that serves as a benchmark for comparison. Our serial implementation of ANN serves as a baseline for comparison with our proposed parallel algorithm.

## 4 Performance Benchmarks and Scaling Analysis

We ran two different test cases on the cluster to evaluate the performance of our algorithm.

- **Test case A:** A smaller-scale experiment of sufficient size to be useful and to validate the scaling analysis.

- **Test case B:** A larger-scale production-level run that will result in an index we can use to test the index building time for bigger problems.

The results of these experiments can be found in Table 1. The table shows the wall clock time to build indices for different number of $n$ data points. Due to capacity restrictions, the experiments could not be run for bigger problems. This also explains the relatively short wall clock time. We observe that the wall clock for index building increases almost linearly with the number of points. This near linear relationship can be explained by the large number of points. The ratio $\frac{log(200{,}000)}{log(100{,}000)} = 1.060$ which is an almost linear

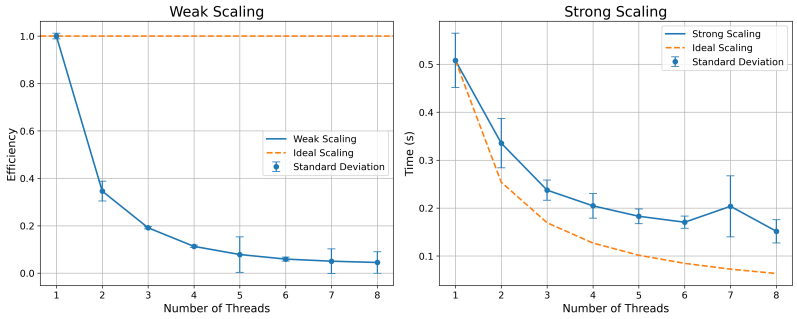


Figure 4: Strong scaling to the left and weak scaling to the right.

relationship. Therefore the doubling in $n$ dominates the relationship and we expect to see the wall clock to double. Due to randomness in the node performance we observe a slightly different outcome, yet the result is very close to the one we expect from theory.

| | Test case A | Test case B |
|---|---|---|
| Typical wall clock time (seconds) | 0.311 | 0.590 |
| Number of points (n) | 100,000 | 200,000 |
| Typical job size (nodes) | 1 | 1 |
| Memory per node (MB) | 52.9 | 105.8 |

Table 1: Workflow parameters of the two test cases used during the project development. During the experiments the dimensionality of each point $d = 128$.

The scaling plots Figure 4 provide insight into the performance of an algorithm in terms of strong and weak scaling. Because of the stochasticity of the tree building, we have run the analysis multiple times and show the standard deviation in the plots as well. For strong scaling we ran the analysis 50 times and for weak scaling 10 times. The difference is caused by the higher computational complexity of a weak scaling anylsis.

In the case of strong scaling, the size of the problem remains constant, while the number of threads used to solve the problem is increased. For our particular algorithm, this implies building a tree for the same number of data points but with a greater number of threads assigned to a compute node.

The scaling plot for strong scaling displays a significant drop in execution time when using up to 4 threads. However, after this point, the speedup decreases drastically, with execution time remaining nearly constant. This is because certain components of the algorithm are serial in nature, such as sampling points and constructing hyperplanes, which can only be executed one at a time. Hence, while the parallel aspect of the code can be improved with additional threads, the serial component dominates the execution speed. Since this component cannot be parallelized, the scaling curve flattens out, demonstrating that the benefits of parallelization have reached their limit.

In the context of parallel computing, the scalability of an algorithm refers to its ability to maintain or improve its efficiency as the size of the problem and the number of processors used to solve it are increased. The weak scaling analysis is a method used to evaluate scalability, in which the problem size is increased proportionally to the number of processors used.

In the presented weak scaling analysis, the problem size is increased according to the relationship $p = n \cdot log(n)$, where $n$ is the base problem size and $p$ is the number of processors used. This relationship

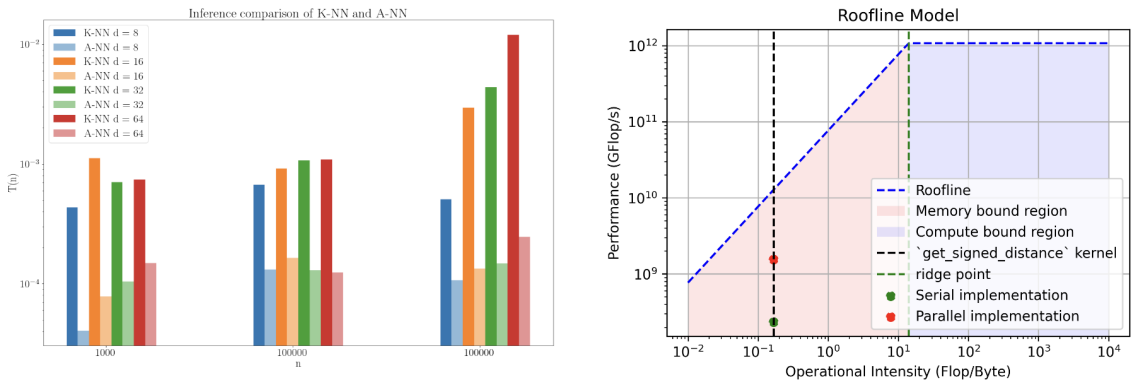


Figure 5: Left: Inference wall time comparison between K-NN and A-NN at different values of $n$ and $d$. $T(n)$ is on a log scale. Scikit-learn is used as a reference K-NN inference. Right: Roofline analysis of the optimized compute kernel. We observe significant performance boosts, but ample room for improvement, though in the memory-bound region.

captures, up to a constant factor, the number of computations required to build a tree for a problem of size $n$.

As depicted on the right side of Figure 4, the execution time of the algorithm only doubles when the number of threads is increased sixfold. This demonstrates a favorable performance of the algorithm in terms of scalability. However, it is worth noting that the performance may still be limited by factors such as communication overhead, load balancing, and memory access patterns. Therefore, further optimization and analysis may be necessary to fully exploit the potential scalability of the algorithm.

When we perform the roofline analysis, we observe a marked improvement in the realized flops per second, as seen to the right in Figure 5. The bulk of this improvement is realized by thread-level parallelism. Within each pass through all points, there is no communication necessary, so the kernels can be parallelized without much synchronization. Still, since we still have serial code that could be optimized, there is room for improvement. Better handling of cache coherency and NUMA awareness would also be natural places to look for more gains.

We also show a comparison of inference times of a naive K-nearest neighbors (K-NN) search implementation, and our approximate nearest neighbors (A-NN) search implementation in Figure 5. For the sake of simplicity, we restrict to $k = 1$ (i.e., first nearest neighbor), and vary the number of points $n$ and the dimensions of vectors $d$. As our algorithm involves inherent stochasticity, we use mean wall time from 10 runs. As we can see, A-NN search offers $\approx 2$ orders of magnitude speedup compared to K-NN search. Furthermore, the speedup $(\frac{T_{\text{k-NN}}(n)}{T_{\text{A-NN}}(n)})$ increases further at higher values of $n$. We also observe that the speedup is consistent with different dimensions at any particular value of $n$. There are a few anomalies in the plot, where at a particular $n$, wall time decreases with $d$. A few possible reasons for this behaviour is either memory alignment or cache block size working better for a particular value of $d$, or the stochasticity in generating the dataset.

The results of this experiment demonstrate that A-NN search is a significantly faster alternative to K-NN search, especially for large datasets. This is because A-NN search uses a more efficient algorithm for finding the nearest neighbors. In addition, A-NN search is more scalable than K-NN search, as the speedup increases with the number of points. These results suggest that A-NN search is a promising approach for large-scale machine learning applications.

## 5 Resource Justification

Our algorithm consists of two parts that can be parallelized. Each node uses OpenMP to parallelize the building of a single tree. With MPI a forest of trees gets build across nodes. However, this second part is embarrassingly parallel. There is no communication needed between the different nodes to compute the trees individually. Therefore we make the assumption that index building will parallelize perfectly over different nodes. This assumption is used in our request for node hours. We request 16 nodes to build a forest consisting of enough trees to reduce the variance. An overview of our request for node time can be found in Table 2. In short, the resources required for one run would be roughly 0.0829 node minutes = 16 nodes $\times \frac{0.311s}{60\frac{s}{\text{min}}}$.

| | Test case A | Test case B |
|---|---|---|
| Search trees per dataset | 16 | 16 |
| Number of points (n) | 100,000 | 200,000 |
| Iterations per tree | $1.6 \times 10^6$ | $3.6 \times 10^9$ |
| Node seconds per tree | 0.311 s | 0.590 s |
| Total node seconds | 4.976 | 9.44 |

Table 2: Justification of the resource request

Table 2 shows a very limited number of node time requested. This is caused by the fast index building. However, the code might have to run often due to changing circumstances. Adding new points to the index is straightforward. One just traverses down the tree and appends the new point to the leaf node. If the leaf node is at maximum capacity, we split the leaf node and create two new nodes. However, the vectors themselves can change because, for example, preferences of customers change. Than one has to rebuild the index to compensate for the distribution shift. Therefore, we expect the ANN to be run often. If we deploy our algorithm in industry rather than academia we expect much higher annual node times.

## References


[1] I. Al-Furajh, S. Aluru, S. Goil, and S. Ranka. Parallel construction of multidimensional binary search trees. *IEEE Transactions on Parallel and Distributed Systems*, 11(2):136–148, 2000.

[2] J. Johnson, M. Douze, and H. Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 7(3):535–547, 2019.

[3] W. Li, Y. Zhang, Y. Sun, W. Wang, M. Li, W. Zhang, and X. Lin. Approximate nearest neighbor search on high dimensional data—experiments, analyses, and improvement. *IEEE Transactions on Knowledge and Data Engineering*, 32(8):1475–1488, 2019.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] R. Ramteke and K. Y. Monali. Automatic medical image classification and abnormality detection using k-nearest neighbour. *International Journal of Advanced Computer Research*, 2(4):190, 2012.