Efficient Parallel Setup of Eigenvalue Problems in the Manifold Learning Framework Datafold

Effizientes Paralleles Aufstellen von Eigenwertproblemen im Manifold Learning Framework Datafold

Master's Thesis in Informatics
at the Department of Informatics of the Technical University of Munich.

Supervisor	Prof. Dr. Hans-Joachim Bungartz
Advisors	Dr. rer. nat. Tobias Neckel
	Dr. rer. nat. Felix Dietrich
Submitted by	B.Sc. Michael Grad
Submission date	14.02.2022

Appendix 1

Declaration

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Ruderting, 14.02.2022, Signature

Contents

1. Abstract

In this thesis, portions of the datafold framework are optimized in regards to runtime duration. Bottlenecks that occur before the eigensolver are located, replaced with improvement candidates and assessed for performance gains. These include the acceleration of a modified min-max-Search as well as various distance calculation methods.

In order to reduce the dimensionality of a dataset, the manifold machine learning framework datafold needs to set up an eigenvalue problem. The underlying distance matrix represents distances from any point in the dataset to any other point within a given radius r in one of a variety of metrics.
The task of finding a suitable value for this radius r, the so called cut_off value, is one of great importance, as it influences the quality of the results. Determining this cutoff radius r has proven to take significantly longer for a higher number of data points.
The sparse neighborhood matrix containing distances undergoes some transformation before it is passed to an eigensolver. Since the eigensolver part has previously been optimized to work on distributed environments such as clusters, this thesis is supposed to assess ways of efficiently parallelizing the prior part, to further utilize the resources of multicore consumer systems and, if possible, clusters.
As a result, the base performance is benchmarked, solutions offered by various frameworks are tested, timed and ultimately compared against each other, for all identified bottlenecks, that are detected to suffer from poor scaling:

The optimize_parameters(...) function is fitted with a Dask implementation, implementing a more performant selection algorithm, that scales significantly better on parallel systems.

Options to optimize the distance calculation are assessed and tested against the current implementations using the hypercube example. A new approach is tested in form of an approximate k Nearest Neighbor(s) (kNN) framework: PyNNDescent.

2. Problem Statement and Project Definition

Expanding the numbers of data points and their dimensionality, that can be feasibly be handled by consumer as well as prosumer and cluster machines is an ever relevant objective of the data sciences. This goal is fueled by an ever increasing availability of data. Considering the ability and trend to fit increasing amounts of sensors to various machines and appliances or the variety of user data, one can quickly oversee the enormous relevance of data analysis techniques. Hence, pushing the bounds of which problems can be solved efficiently, opens new possibilities. Data sets for which the processing would not have been feasible, come closer to reach and the runtime of smaller sets can be affected positively.
The influence factor, that majorly affects the runtime of machine learning algorithms, is the dimensionality of a dataset, often increasing the complexity exponentially. This phenomenon has been denominated as the "Curse of Dimensionality" and is a major hurdle in the data sciences that brings a variety of inconveniences for certain applications.

2.1. The Datafold Framework

Therefore one might be interested in reducing the dimensionality of a given dataset while preserving as much of the information as possible. One framework taking over this task is called "Datafold".
Datafold describes itself as a "Python package containing operator-theoretic, data-driven models to identify dynamical systems from time series data and to infer geometrical structures in point clouds" [8].

In summary, [19] describes the functionality of the datafold framework as follows:
Rather than a randomly generated set of data points, real-world data often contains some underlying structure within itself, called a "manifold". Applying a parametrization to this underlying geometric structure enables mapping of the original high dimensional data points onto a much lower dimensional space, hence reducing the dimensionality and therefor its computational complexity. Finding such structures within a given dataset and mapping the high-dimensional data points onto a lower dimension is the goal of datafold.

Some of the main features of the framework are described in [8]:
It comes with relevant data structures to map point clouds onto manifolds as well as "Time Series Collections". An implementation of the so called "Diffusion Maps" algorithm enables the user to let the framework infer structures within the high dimensional data space, enabling dimensionality reduction of a given dataset, preserving as much of the original information as possible. Also, interpolation on the manifold allows for "out-of-sample" queries.

The manifold learning framework relies on the calculation of a distance matrix, containing all pairwise distances (i.e. the distances between any two points of the set), which are below a certain threshold (called the cut-off parameter). The scipy.cKDTree, that has been the go-to algorithm for said distance calculation so far, has proven to scale rather poorly, utilizing only one core, but remaining quite conservative concerning memory usage. While these attributes makes it a good choice for "laptop" use, where big datasets combined with a lack of RAM might quickly overcharge your system. Datafold requires these distance calculation to be executed on high-dimensional datasets ($d \gtrsim 2,000$), while trying to maximize the number of data points.

2.2. Project Definition

This thesis comes with a kind of scouting character, as it does not allow to precisely describe its scope. In some sense, this thesis can be viewed as an agile "scrum" project with the backlog being "improve performance". The paths which this project took were carefully discussed and aligned in frequent meetings, which ultimately lead to the results which are presented here. This also meant, that the scientific fields which would be required for optimization had to be scouted out from little to go. In that regard, the task at the beginning of this thesis was to research, implement and assess optimizations for parts of the datafold framework where-ever it seems to be necessary and doable.

The objective of this thesis is defined rather openly, as it does not state a specific implementation to be added to the framework. On one hand, this enables the possibility to choose and assess frameworks freely. On the other hand, it adds an agile aspect to this thesis and the freedom to explore a broader scope of favorable solutions. Since it is neither set nor known which frameworks can offer performant solutions for the identified problems, the respective fields need to be scouted from the ground up. As a result, certain paths and objectives had to be reformulated during the project, as will be depicted later in 7.

The open ended goals of this thesis therefor are to:

- detect and confirm suspected bottlenecks
- take baseline benchmarks for a status quo assessments and comparison purposes
- do literature and framework research to find algorithms and implementations that might reduce the runtime of the identified functions
- implement the identified candidates
- test these candidates and benchmark them and see, if they improve the runtime
- compare the candidates to the baseline benchmarks to assess the performance gains

- identify and, if possible, alter parts of the code that appear to be in need of maintenance or are deprecated, etc.

for use-cases ranging from laptop to multi-node clusters. Note that the scope is not intended to produce such frameworks from scratch, but rather to find existing solutions to the identified bottlenecks. Since many currently used solutions are from sophisticated sources, such as SciPy, scikit Learn, etc., optimizing an original implementation and beating said frameworks would therefore blow out of proportion.

As a result from this approach described above, next to some fully implemented solutions, some artifacts exist, capturing an idea, that was not further investigated for one reason or another. Still it is important to understand which decisions were made and why, for which an overview will be given in 7.

2.3. Deployment Cases

When scouting for solutions, multiple deployment variants were kept in mind. Optimally, all new implementations would bring support for local and cluster machines, i.e. distributed as well as shared memory systems. As will be shown, finding readily available framework for our tasks, would not always be possible.

Shared Memory Systems

On the one hand we have more traditional shared memory systems, as can be found typically in laptop or workstation scenarios. Due to the widespread use of such (typically) single CPU- / multi thread-systems, many frameworks such as NumPy, SciPy, scikit Learn, etc. often support efficient utilization of multiple cores "out of the box" for certain functions.

Distributed Memory Systems

On the other hand, distributed memory systems are mainly seen on cluster or super computing hardware. Containing multiple nodes, possibly more than one CPU per node and many cores, more advanced multiprocessing mechanisms, such as the Message Parsing Interface (MPI), are required.

2.4. Structure of the Thesis

In the following, a short overview of how this thesis is structured will be given.

Firstly, benchmark data from a previous project will be demonstrated to show where the suspicions for the bottleneck come from and a general use case for datafold will be depicted. Afterwards the suspected bottlenecks will be confirmed and the responsible functions will be located. From there on out, the thesis will focus on the two areas of improvements that have been found, namely the optimize_parameters(...) function and the distance calculation, in a successive fashion.

Beginning with the optimize_parameters(...) function, the baseline implementation is explained and benchmarked. A literature and framework research will be given in the theoretical background section, contextualizing the function and exploring alternatives. The new implementation is then explained and benchmarked, finally comparing it to the previous implementation.

The same structure is followed for the work on the distance calculations. Examining the baseline implementations and their performance, researching the theoretical backgrounds after which the new implementations will again be explained and benchmarked, allowing for detailed comparisons. Finally the impact of a crucial optimization parameter will be investigated.

Finally a recap is given of the project, explaining certain decisions along the way. An outlook is given, presenting artefacts and possibilities on how they can be further pursued in future work.

A summary and conclusion concludes the thesis.

3. Previous Work

3.1. Previous Findings

As the preliminary goal was to generally reduce the computation time of a given input shape, the first step was to identify bottlenecks. Having implemented a SLEPc Eigensolver into the framework in a prior Interdisciplinary Project (IDP) (see [13]), general benchmark data already gave a rough overview of functions that are capable of improvement. Since back then, the main focus was on a higher number of data points rather than their dimensionality, a certain bottleneck of this specific use-case was discovered, which would turn into the first part to be optimized. Also, general usage of the framework showed, as expected, that the calculation of the pairwise distances and setup of the distance matrix took a considerable amount of time. However, before these bottlenecks are described in more detail, it should be recalled on how a typical application of datafold looks like, which functions are called and how they are connected to the bottlenecks.

3.2. Datafold Example Use-Case

```
1  # Generate an s-curve dataset (representative for any input dataset)
2  X = make_s_curve(...)
3  # Create a PCManifold object from the Dataset
4  X_pcm = pfold.PCManifold(X)
5  # Calculate a suggestion for the cutoff and epsilon parameter
6  X_pcm.optimize_parameters()
7
8  # Initialize a Diffusion Maps Object, with a given kernel and additional
       parmeters
9  dmap = dfold.DiffusionMaps(
10     kernel=pfold.GaussianKernel(epsilon=X_pcm.kernel.epsilon),
11       n_eigenpairs=9,
12       dist_kwargs=dict(cut_off=X_pcm.cut_off))
13
14 # Fitting creates the distance matrix and finds relevant eigenvectors and
       values
15 dmap = dmap.fit(X_pcm)
```

Figure 1 Example use case of datafold based on the included s-curve example.

As shown in figure 1, a typical application of the datafold framework comprises a couple of key function calls. In this example, we assume the dataset to be a generic, three dimensional s-curve with an arbitrary amount of sample points, as created in line 2 of figure 1.

The next step is to pass the input data to datafold, by creating a PCManifold objects, which wraps the numpy.ndarray with some additional functionality. It collects relevant information

such as the original data with a kernel, distance parameters as well as the cutoff, which is calculated in the next step.

Line 6 calls the "optimize_parameters(...)" function on the newly created PCManifold object. At this point, two important parameters are advised, namely the "cut_off" and "epsilon" value. The cut_off mainly comes into play during the calculation of the distance matrix, as it defines the distance within which two points are considered neighbors, resulting in an entry in the distance matrix. Hence, when the distance between two points exceeds the cut_off value, the entry of their pairwise distance is set to 0 in the distance matrix, disregarding any neighborhood connection between them. As a direct consequence, this renders the distance matrix sparse.
Furthermore, the creation of a DiffusionMaps object in lines 9 through 12 contains the selection of a specific kernel (in this case a gaussian kernel, other kernels like cKNN exist), the amount of eigenvectors to be searched for, as well as the previously mentioned cut_off and epsilon value.
Choosing this cut_off value appropriately is gaining an alignment on quality and calculation time. An important aspect of the dimensionality reduction is the preservation of neighborhood information. Setting the cut_off value too low, might harm the quality of the outcome, as too little neighbors are considered for each points. By choosing a low cut_off value, the distance matrix becomes more sparse, while less neighbors are considered, losing some of the neighborhood information. On the opposite, a higher cut_off value results in a more dense matrix, considering more neighbors, requiring more computations which lead to an increased execution time.
Regarding epsilon, the kernel (in this example: a Gaussian kernel) transforms the distance matrix D as shown in 3.1. Thereby low distance values (two points that are in close proximity to one another) are interpreted as heavily connected (as 3.1 grows when the values of D approach 0), whereas higher distances are penalized by a lower connection weight.

$$K = \exp\left(\frac{-1}{2\epsilon} \cdot D\right) \tag{3.1}$$

Finally by calling the fit(...) method in line 15, starts the computation of the kernel matrix according to the kernel selection in line 10 as well as the calculation of the specified amount of eigenpairs, as chosen in line 11.

In the intention to identify bottlenecks, it can inferred from this exemplary overview of the datafold pipeline, that those can be found mostly in line 6 and line 15. Since the remaining functions only contain object instantiations which hardly impose any computationally intensive tasks. The optimize_parameters(...) and dmap.fit(...) function however contain calculations and analyse the input dataset and therefore bear the possibility of poor runtime scaling. This

has proven true in the benchmarks that have been run in the prior work (see [13]), implementing the SLEPc eigensolver. These observations lead to the intention to further optimize the codebase prior to the eigensolver and therefor anything in front of the second half of the dmap.fit(...) method that imposes a significant runtime penalty.

3.2.1. Optimize Parameters

As mentioned before, the project [13] prior to this thesis focused more on the number of datapoints rather than their dimensionality (3 to 7 dimensions, 12,500 to 3,200,000 data points). Since this prior project focused on the solver that follows after the distance calculation, the dimensionality of the input data would not affect runtime significantly, as the sample size is the only significant remaining scaling factor, after the distance calculation reduces the high dimensional data down to a scalar distance matrix.

The benchmark in figure 2 (taken from [13]), that resulted from the prior project, shows the two functions before the eigensolver, that exceed 1 second of execution time and plots their weak-scaling behavior. You can clearly see, that the optimize_parameter(...) function does not scale well for a fixed ratio of cores to input size.

3.2.2. Distance Calculation

A similar behavior has been observed with the compute_kernel_matrix(...), which contains the calculation of the distance matrix. Hence, it was to be expected to find a considerable amount of runtime at this part of the code. Looking at the data of the corresponding function in figure 2, it is visible, that the compute_kernel_matrix(...) shows an even steeper increase than the optimize_parameters(...) function. Assessing the average over $\frac{t_n}{t_{n-1}}$ for the timings $t_1, ..., t_n$ shows an average of 2.05 for optimize_parameters(...) and 2.53 for the compute_kernel_matrix(...) function.

A theoretical background in form of a literature research will be given in each of the chapters focusing on the identified bottlenecks (see 5.3 and 6.3), as their nature is inherently different.

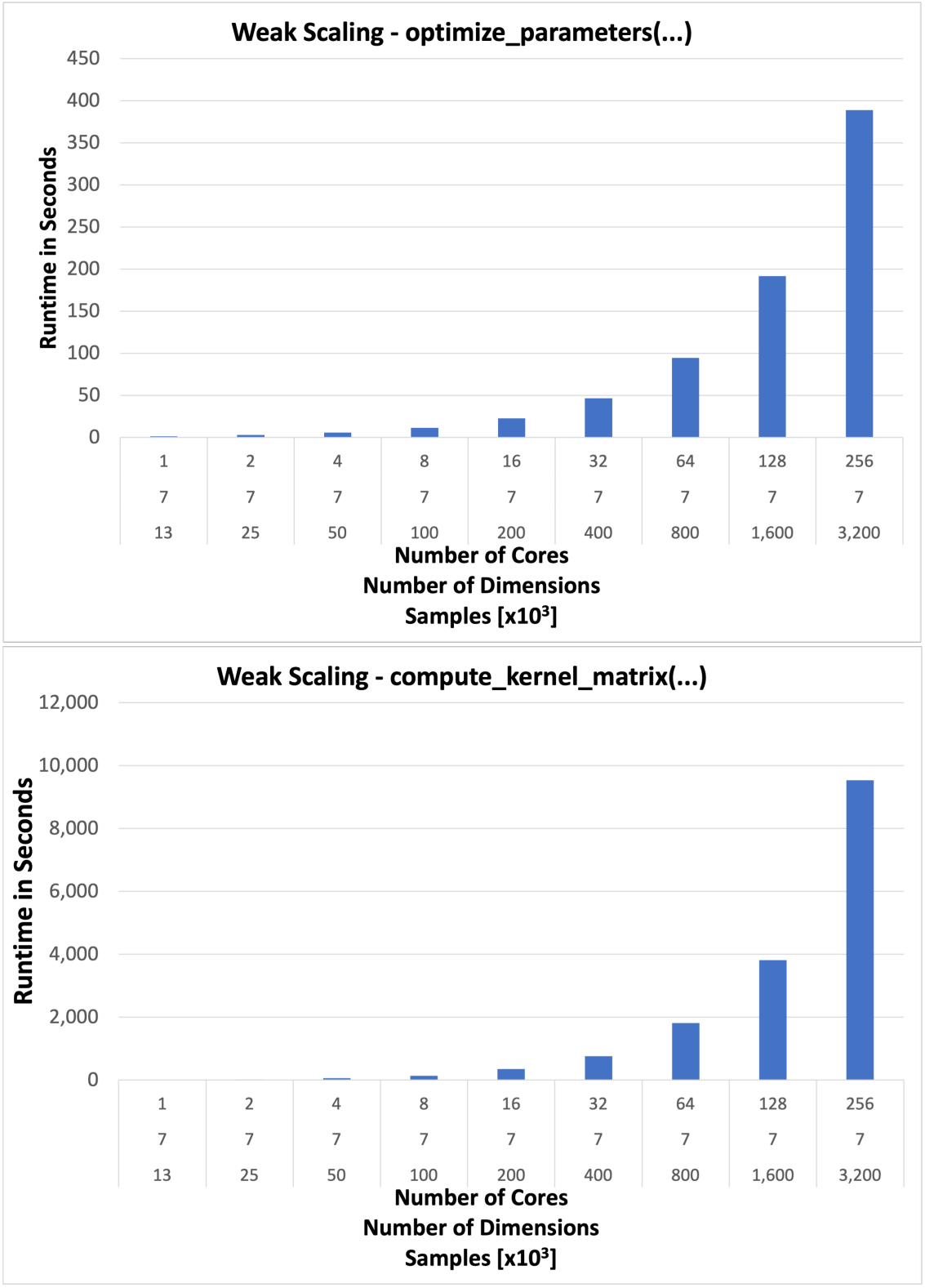


Figure 2 Cluster benchmark results from [13] show, that optimize_parameters(...) does not perform well in the weak-scaling benchmark, showing a (more than) quadratic runtime increase. A similar runtime behavior can be observed with the compute_kernel_matrix(...) function.

4. Bottleneck Detection

4.1. Optimize_parameters(...)

In order to gain a more detailed insight on where the bottlenecks of the previously identified functions are located, a benchmark script was created around a sample datafold pipeline (similar to code example 1) which wrapped all datafold steps with a Python cProfile instance. The underlying datasets originates from a random hypercube.
This allowed to later visualize the profiling results and identify which function calls would be responsible for high execution times.
It should be noted, that these profiling runs were performed on a different machine as the baseline benchmarks that follow in 5.2.2 and 6.2, as some of the bottlenecks were especially present on certain hardware configurations. The machine, which produced the following profiling results was set up as follows (in the course of this thesis referred to as the "laptop" machine):

- Laptop: Apple MacBook Pro (15", 2018) running a Docker container
- CPU: 6 core, 12 threads Intel Core i7, 6 threads available to the container
- RAM: 16 GB DDR4; 12GB available to the container
- macOS Monterey 12.0.1

Starting with the function call that occurs first, the findings of [13] showed, as illustrated in figure 2, that for a high number of datapoints, optimize_parameters(...) would cause a vast increase in execution time. Analyzing its profiling results, as depicted in figure 3, grants the following insight:

In figure 3 you can see the call-stack that is generated when calling the optimize_parameters(...) function. Hence, you read the graphic from top to bottom, where the horizontal width of the rectangle depicts the proportional share in the overall runtime of the function shown directly above it. Below the function name, the time spent inside the respective method is depicted. Note that functions below a runtime of $1/1,000$ of the overall runtime are omitted. Figure 3 depicts a scenario of low dimensionality with higher number of datapoints (500,000).

Therefor it can be concluded, that the optimize_parameter(...) function spends most of its time (71.4 seconds out of a total of 83.0 seconds) inside of another function, called "_kth_nearest_neighbor_dist(...)", which can be identified as a first candidate for optimizations.

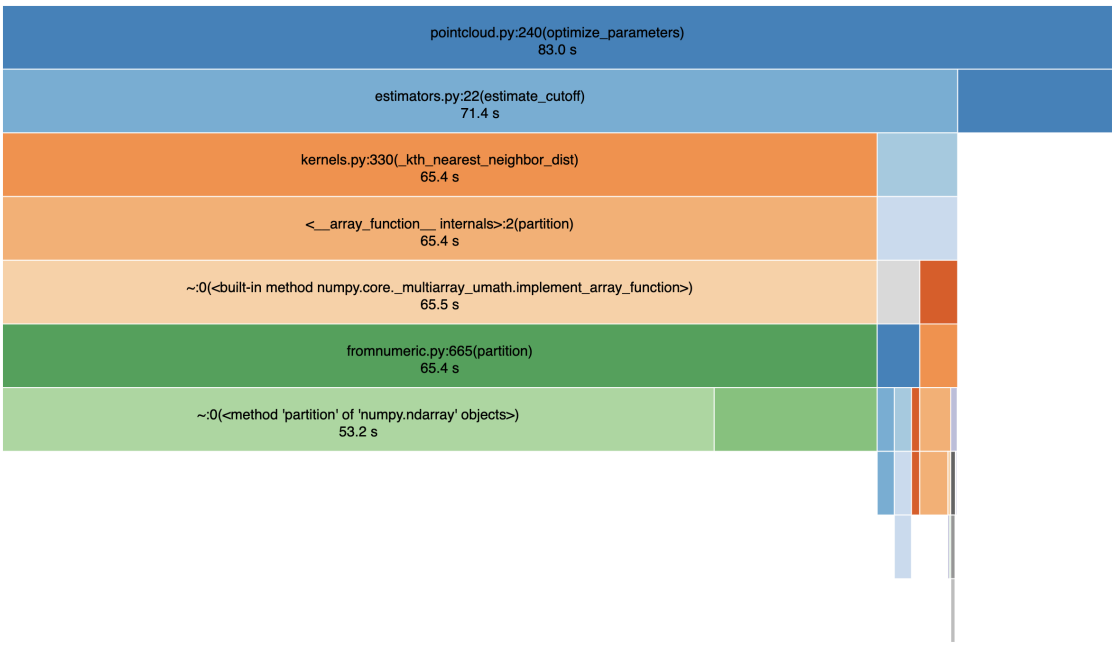


Figure 3 Profiling results of the original optimize_parameters(...) method as visualized by the "snakeviz" tool.

4.2. Distance Calculations

Same has been done for the remaining function that has been observed to cause long execution times: dmap.fit(...). For apparent reasons it is to be expected, that the distance calculation is the main contributor to the runtime. A brief look at the results of a profiling run (see figure 4) comprising fewer data points (20,000) of higher dimension (1,000) confirms this hypothesis.

As expected, for higher dimensional data, the calculation of the pairwise distances, i.e. the (sparse) distance matrix becomes computationally rather challenging. Out of the total 238 seconds elapsed in the dmap.fit(...) method, 234 seconds are caused by the original low level implementation of the pairwise distance calculation, while its parent method "distance.pdist(...)" or any of the function above in the call-stack barely add any runtime overhead. This leads to the conclusion, that any improvements to the distance calculation need to focus on more efficient low level distance calculation approaches, as opposed to the transformations and functions that take place after the initial distance matrix is set up. Therefor the main focus concerning the optimization of dmap.fit(...) will shift to distance.pdist(...). Since distance.pdist(...) has shown to be more relevant in this case and many of the distance framework in question also offer functions which can serve cdist(...), the focus regarding implementation and benchmarking will lie on distance.pdist(...).

With this confirmation, the two main bottlenecks for the cases: low dimensional / high amount of data points, as well as high dimensionality / low number of datapoints are found. Further

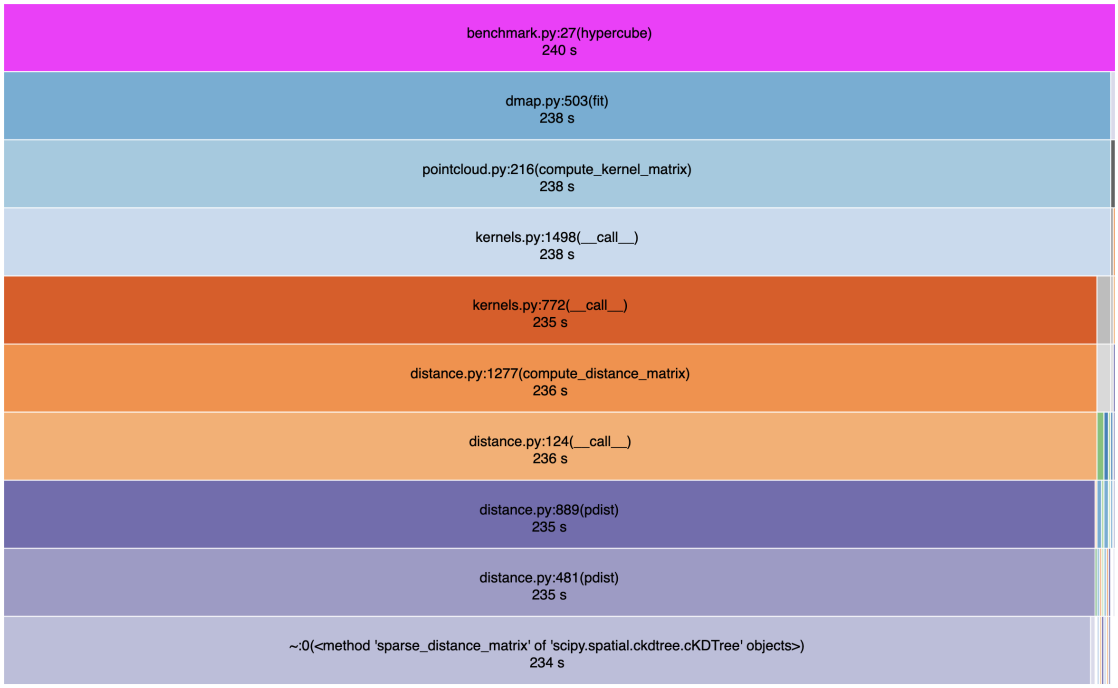


Figure 4 Profiling results of the original dmap.fit(...) method as visualized by the "snakeviz" tool. It is apparent, that almost the entirety of the runtime duration of dmap.fit(...) is spent in one low level function call.

complications that would arise from an increase of both, are subject to future work, as their application becomes more relevant when optimization of the above mentioned parts is completed and allows for a higher dimensional data set with many datapoints.

From here on out, the shown bottlenecks will be gone over in succession. For both of the identified bottlenecks, baseline benchmark(s) will be generated, candidates of improvement will be implemented and finally benchmarked and compared to the baselines.

5. Results of Optimizing optimize_parameters(...)

5.1. Baseline Implementations

As mentioned previously, optimize_parameters(...) calculates goto values for the cut_off and epsilon parameters. In order to do so, two functions are used, namely estimate_cutoff(...) and estimate_scale(...). estimate_cutoff(...) in return calls "_kth_nearest_neighbor_dist(...)", which has previously been proven to be computationally intensive for a large amount of datapoints.

To understand why the implementation leads to its runtime, it needs to be understood on how the cut_off is determined.

```
1  def estimate_cutoff(pcm, n_subsample: int = 1000, k: int = 10,
       random_state: Optional[int] = None,distance_matrix=None) -> float:
2      n_points = pcm.shape[0]
3      n_subsample = np.min([n_points, n_subsample])
4
5      if distance_matrix is None:
6          perm_indices_all = np.random.default_rng(random_state).permutation
       (n_points)
7
8          distance_matrix = compute_distance_matrix(
9              pcm[perm_indices_all[:n_subsample], :],
10             pcm,
11             metric="euclidean",
12             backend="brute",
13             kmin=k,
14             # for estimation it is okay to be not exact and compute faster
15             **dict(exact_numeric=False)
16         )
17
18         k = np.min([k, distance_matrix.shape[1]])
19         # need to transpose the matrix here to correctly work with
20         # _kth_nearest_neighbor_dist
21         k_smallest_values = _kth_nearest_neighbor_dist(distance_matrix.T,
       k)
22     else:
23         # distance matrix is assumed to be symmetric here (no transpose
       required)
24         k_smallest_values = _kth_nearest_neighbor_dist(distance_matrix, k)
25
26     est_cutoff = np.max(k_smallest_values)
27     return float(est_cutoff)
```

Figure 5 A section of the estimate_cutoff function. (Edge cases, and parameter assertions are omitted.)

As can be deduced from figure 5 (lines 1-3), the default cut_off estimation is performed by subsampling 1,000 datapoints, or the entire dataset, if it contains less then 1,000 points (to be referred to as "n_subsample"). Assuming that no distance matrix has been calculated yet, a

random permutation of these n_subsample datapoints is created (line 6). The following brute-force distance matrix computation (lines 8-16), calculates the distances between the (1,000) chosen subsamples and all other points in the dataset. This kind of distance calculation, where the reference set of points does not equal the query set, is internally referenced as a component wise distance calculation (or cdist(...)), in line with the SciPy nomenclature. Since this function gives an estimation to a suitable parameter, the function refrains from demanding exact numerics (line 15). As the distance_matrix is of size $n_{subsamples} \times n_{datapoints}$, the min(...) function in line 18 checks, if the underlying distance matrix contains distances of the chosen sample points to at least k points. The function default for k is 10, the optimize_parameters(...) function however specifies a default k of 25. At this point (before line 21 is called), the distance_matrix variable contains an 2D numpy.ndArray of shape $n_{subsamples} \times n_{datapoints}$, where each row contains the distances from one sample point to all other datapoints in the set (ordered by index). Finally, the function in question, "_kth_nearest_neighbor_dist(...)" is reached. Its purpose is to select the k^{th} smallest value in each row of the distance_matrix. This operation can be computationally challenging, when each row contains a high amount of entries, which is the case with big datasets. Hence, it explains why this function scales rather poorly with a high number of datapoints. Since 1,000 randomly chosen points are sampled at most, the brute forcing does not suffer too much from a higher dimension ($\sim 2,000$).

```
1  dist_knn = np.partition(distance_matrix, k - 1, axis=1)[:, k - 1]
```

Figure 6 The partition function used by the _kth_nearest_neighbor_dist(...) function.

The "_kth_nearest_neighbor_dist(...)" function performs some input checks and offers supporting sparse distance matrices. Most importantly, it relies on the "numpy.partition( )" function (see figure 6) to get the k^{th} smallest element. The functionality of this method, partitioning the array for a given index k, could be described as a "pseudo-sort", where the k^{th} element is at its correct position (i.e. at the position it would be if the array was sorted). The remaining values are stored (unordered): behind the k^{th} index if they are bigger, or in front, if they are smaller (see figure 7).

```
1  >>> a = np.array([3, 4, 2, 1])
2  >>> numpy.partition(a, 3)
3  array([2, 1, 3, 4])
```

Figure 7 An example of the numpy.partition(...) function. Note the unordered values before the three (of the result in the last line). Further values beyond the third index would have no guarantee on their sorting

As can be seen in figure 6, for all rows, only the element on index k - 1 are picked and stored in a 1D array, effectively containing the distances from all subsampled points to their k^{th} closest neighbor (in the default case: to their 25^{th} closest neighbor).

This 1D Numpy array is returned to the estimate_cutoff(...) function, which picks the maxi-

mum value of this list.

To come back to the findings of the previous profiling session (see figure 3), you can see, that the aforementioned numpy.partition(...) function causes this bottleneck and therefor would need attention to be optimized.

5.2. Benchmarking

In order to be able to assess potential improvements in runtime, it is necessary to first capture the performance of the previous implementation. Therefor the functions that cause the bottlenecks which were detected in 4, are now going to be benchmarked with a fixed sample input in form of a n-dimensional hypercube.

5.2.1. Benchmark Scenario

In order to preserve some form of comparability between this thesis and the previous project [13], the same exemplary case will be used, while slightly adapted to measure relevant timings. The benchmark scenario structurally resembles the pipeline shown in figure 1, however, another input dataset is used, which parameters (dimensionality and number of datapoints) can be easily modified.

```
1 _rng = np.random.default_rng(1)
2 lower = np.linspace(0,.2, n_dimension)
3 higher = 1-lower
4 # generate point cloud
5 X = _rng.uniform(low=lower, high=higher, size=(nr_samples, n_dimension))
6
7 X_pcm = pfold.PCManifold(X)
```

Figure 8 The dataset generation used in the majority of benchmark runs. Note that this example was provided, while working on [13].

As can be seen in figure 8, a simply parameterized n-dimensional "rectangle" with an arbitrary amount of data points can be generated by specifying "n_dimensions" and "nr_samples". Therefor, higher and lower bounds are set and enforced. The resulting dataset X is then turned into a PCManifold (line 7) and is ready for computations. Note how the seed of the numpy.random.default_rng(...) is fixed to 1 (see line 1), leading to the same values over different runs.

One aspect to note would be, that due to fixing the benchmark case to the one described above, the density of the data is not varied. While this might impact the performance of some implementations more than others, we need to fix some variables, agreeing on their implications, to not blow up the number of cases that need to be run.

In order to control which code execution is enabled (i.e. to force certain functions to run), environment variables like "DF_DIST_BACKEND" or "DF_ESTIMATOR_DASK" are used and set accordingly. Also, in order to maximize interoperability between different operating systems, a development environment was set up using Docker, which hosts a jupyter lab instance on ubuntu 18.04. While a slight overhead is to be expected, it is significantly smaller than using a Virtual Machine (VM). Furthermore, since all benchmarks are conducted inside the container, they all contain same overhead offset, allowing for a fair comparison.
The used Docker container is a derivation of the one used in [13].

The host system comprises the following specs (in the following referred to as the *Desktop Machine*):

- CPU: 8 core, 16 threads Ryzen 3700X at 3.9GHz (turbos to 4,5GHz)
- RAM: 32GB DDR4; 25GB are available for the container
- Windows 10 Home, Version 21H2; 19044.1466
- WSL2 activated

5.2.2. Baseline Benchmarks

Figure 9 shows the measured baseline benchmark for the optimize_parameter(...) function. These numbers focus on the _kth_nearest_neighbor_dist(...) function, therefor all following timings were measured right before and after this function call. As longer runtimes begin to occur for higher number of datapoints, we fix the dimensions to 7 (to preserve comparability to the IDP data) and scale up the number of points. The desktop system used, is the one described above, while the laptop results originate from the system outlined in 4.

It has to be kept in mind, that this timing does not take the distance calculations between the 1000 random samples and all other points into account. By default, this is done using the brute force distance backend.

You may note, that the laptop curve shows a lower runtime for the 500,000 sample case, which was profiled and shown in figure 3. This deviation occurred reliably when activating the profiler. The relevant comparison to the new implementation will still depict the gained improvements, as they originate from the same benchmark process, excluding the profiling.

5.3. Theoretical Background

Now that the bottleneck was detected and its current implementation has been assessed, the kind of problem at hand should be categorized and researched in oder to find information

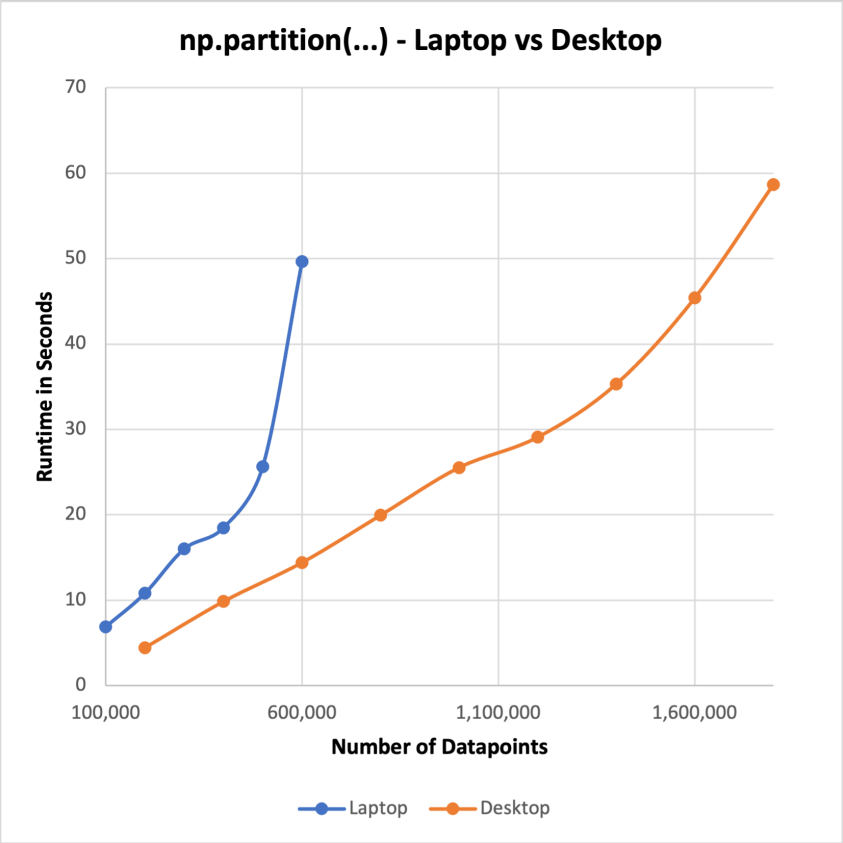


Figure 9 Runtime of the _kth_nearest_neighbor_dist(...) function on a laptop and a desktop system.

about their current state of the art.

5.3.1. Selection Algorithms

The task that is taken over by the _kth_nearest_neighbor_dist(...) function, is to find the k^{th} smallest element in each row of the input array. This is generally referred to as a "selection algorithm". It might seem counterintuitive that a *partition* method is used for a selection problem, however the two problems are in some ways related. Partitioning, as opposed to selection problems promise splitting an input array into two subsets, one with the values that are smaller than a given element and the ones which are greater or equal. In that regard, np.partition(...) delivers both, partitioning and (indirectly) selecting.

Since the selection task can be achieved by sorting the respective array and selecting the given index, this function can be naively implemented with an $\mathcal{O}(nlog(n))$ average runtime. This comes to be, as Quicksort is capable of sorting in an average runtime of $\mathcal{O}(nlog(n))$ and the furthermore required array access lies within $\mathcal{O}(1)$. However optimized algorithms with an improved runtime behavior exist for this specific task.

Tony Hoare's: Quickselct

One implementation for a selection technique originates from Tony Hoare [15], the developer of the famous Quicksort algorithm. Hoare also published a selection algorithm, called Quickselect (or simply "find"), which is derived from its sorting relative, Quicksort.

[11] summarizes the widely known approach by [15]. The code example provided in [11] is shown in figure 10. Due to their similarity, Hoare falls back to the same partition method, which he used in Quicksort, differing only in the recursive path. Hence, the sub-routine can be seen as given and it can be assumed, that it partitions the array within the bounds given and returns the index of the pivot element (i.e. the element which is now on its correct position w.r.t. a fully sorted array). The partition algorithm can be substituted rather easily and its choice has been subject of research with alternative solutions existing, like the Lomuto partitioning scheme (see [12]).

As can be seen in figure 10, Quickselect mostly relies on the partitioning and only recurses into the subarray, which contains the k that is searched.

```
# arr = input array
# l = left index
# r = right index
# k = as in the kth smallest element to find
def kthSmallest(arr, l, r, k):
    index = partition(arr, l, r)
    # if position is same as k
    if (index - l == k - 1):
        return arr[index]
    # If position is more, recur for left subarray
    if (index - l > k - 1):
        return kthSmallest(arr, l, index - 1, k)

    # Else recur for right subarray
    return kthSmallest(arr, index + 1, r,
                       k - index + l - 1)
```

Figure 10 The quickselect algorithm as a Python implementation. Original structure taken from [11], some parts have been changed.

Overall, Quickselect suffers and benefits from the same aspects as Quicksort. It provides good average runtime performance in $\mathcal{O}(n)$, but rather bad in worst case examples of $\mathcal{O}(n^2)$ ([11]). However, according to [5] "selection can be performed in linear time, in the worst case".

5.3.2. Numpy Implementation - Introselect

Taking a step back and looking at the algorithm that Numpy implements, one comes across the "Introselect" algorithm. As does Hoare's implementation, Introselect stems from a sort-

ing algorithm, introduced as Introsort. Presented by [21], it claims to reduce the worst case runtime down to a $\mathcal{O}(nlog(n))$ or even to $\mathcal{O}(n)$ by introducing a depth limit to Hoare's "find" algorithm and switching from Quickselect to a linear, worst case implementation (Median-of-3). [21] therefor suggests two approaches for when to switch algorithms. Introducing a constant bound on the depth of partitions is deemed suboptimal, as any input with sufficient length would eventually trigger the switch. He rather suggests "to require that the sequence size is cut at least in half by any k consecutive partitions, for some positive integer k" ([21] p.9). If this requirement is not met, the switch to the worst case linear algorithm is performed. Bounding the depth of the partitioning to $k\lceil log_2 N\rceil$ and overall runtime to $\mathcal{O}(n)$ ([21] p.9).

The Numpy documentation [25] confirms this worst case runtime of $\mathcal{O}(n)$. Also note, that the implemented Introselect is not stable, which however is not of concern for the required functionality in datafold. Since the partition function therefore seems to implement a rather good selection algorithm, the focus for its optimization shifts to how the partition function is applied.

5.4. Implementation - DASK

As the Numpy partition function has been proven to run within a reasonable worst-case runtime of $\mathcal{O}(n)$ (see 5.3.2), another solution was necessary to be found.
After some research, the so called "DASK" framework appeared to deliver the solution to the problem at hand. While, according to [6], DASK still uses the numpy.partition(...) method, it has shown to significantly improve runtime performance.

DASK describes itself as a "general purpose parallel programming solution" ([7]). One of the main advantages that come with using DASK, are, that it

A) enables powerful parallel functionality for systems of all sizes and

B) makes transitioning between them considerably smoother.

Supporting Numpy Arrays, DASK internally splits up big arrays into so called "chunks" and offers the required mechanisms to apply calculations on these chunks efficiently in parallel. A major upside to this approach, is the following flexibility to work on datasets, which would not fit into the memory of less capable machines. Along with those features comes an advanced scheduling system, in which computations are not executed with every function call but upon a discrete call of the .compute(...) function (or whenever the result is plugged into third party functions, which expect an actual result). Hence, before any computation is started, a compute-graph is set up, which optimizes the computation in respect to the chunks generated from your input data and the requested functions to be applied on them. The DASK

scheduler then handles the computations, following the compute-graph generated. In most use-cases on consumer machines, a thread pool is used to fulfill all tasks as parallel and efficiently as possible.
Since the scheduler is somewhat independent of the compute-graph setup, it can be substituted by a more advanced version (the so called "distributed scheduler"), which offers execution on distributes systems, such as High Performance Computing (HPC) clusters, Kubernetes or Docker instances.

The implementation of DASK into datafold in order to solve the slow partitioning problem can be achieved in a few simple steps.

Old implementation using np.partition(...):

```
1 if isinstance(distance_matrix, np.ndarray):
2         dist_knn = np.partition(distance_matrix, k - 1, axis=1)[:, k - 1]
```

New Implementation using DASK:

```
1 if isinstance(distance_matrix, np.ndarray):
2     distance_matrix_dask = dask.array.from_array(distance_matrix,
              chunks=(-1, 'auto'))
3     dask_partition = distance_matrix_dask.topk(-k, axis = 1)
4     dist_knn = dask_partition.compute()[:, k-1]
```

Figure 11 The new vs. old implementation of the selection algorithm. Remember that k describes the index in which our desired value would be in a sorted array, or: we are looking for the k^{th} smallest element in each row of the distance_matrix.

As you can see in figure 11, the DASK approach starts by converting the input distance matrix into a dask.array (line 2). Note the "chunks" parameter that lets the user specify how the array might be split up. Since the function we want to implement is intended to partition the 2-D array row-wise, slicing within rows by setting the first value to -1 is disabled. In order to let DASK decide on a good value for the chunk size in Y direction (i.e. how many rows are in one chunk), the second value is set to "auto".
As a next step, the relevant partitioning function is called in line 3, called .topk(...). According to its documentation, it returns the k largest elements from the array and returns them in a sorted list. By calling the function with a negative k, the function returns the k smallest values. Since, in this case, we need to find the k^{th} element, i.e. the k^{th} smallest element, we use this feature to get to our desired outcome. Finally, since our matrix distance_matrix is a two dimensional array, we need to state the axis along which the partitioning algorithm should work.
Since dask_partition now is a 2D array with the number of rows as distance_matrix, and k elements per row. Hence finally, only the last element of each row is selected, leaving us with our desired result: the k^{th} smallest distance value for each point.
Unfortunately the DASK framework at the moment does not yet support the scipy.sparse.csr

matrices that are used in the lower part of the function. This support however is planned according to Scipy [31].

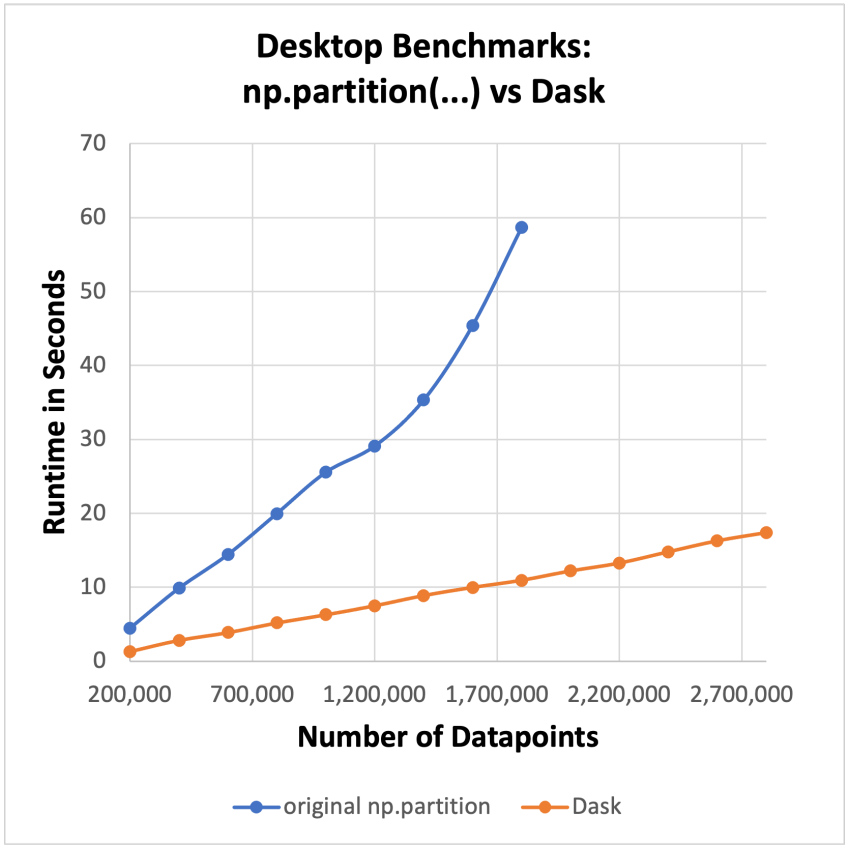
The general runtime improvements that are provided by the DASK framework have grabbed the attention of many developers in this field. Therefor, in my opinion, the project should be monitored and further implementation of its features throughout datafold should be considered, as soon as compatibility to scipy.sparse matrices is provided, as it could further improve the runtime and scalability of the whole datafold framework.

5.5. Comparison to Benchmarks

In the following, the runtime improvements from the implementation of the Dask framework will be shown.

As can be seen in figure 12, the Dask version of the _kth_nearest_neighbor_dist(...) function yields decent improvements for both runtime and computability. For some cases, the Dask version runs up to 5 times faster, while enabling the laptop and desktop to increase the number of datapoints from 600,000 to 1,000,000 and 1,800,000 to 2,800,000 respectively, before reaching memory constraints.

Similar to the graphic which was given in 5.2.2, figure 13 compares the performance between laptop and desktop. As expected and already seen in the baseline benchmark, the more powerful desktop is capable of processing more than double the number of points.
It can be seen, that the laptop-curve is considerably steeper than the desktop-curve for any amount of data points. Possible causes might include the increased number of threads on the desktop machine.



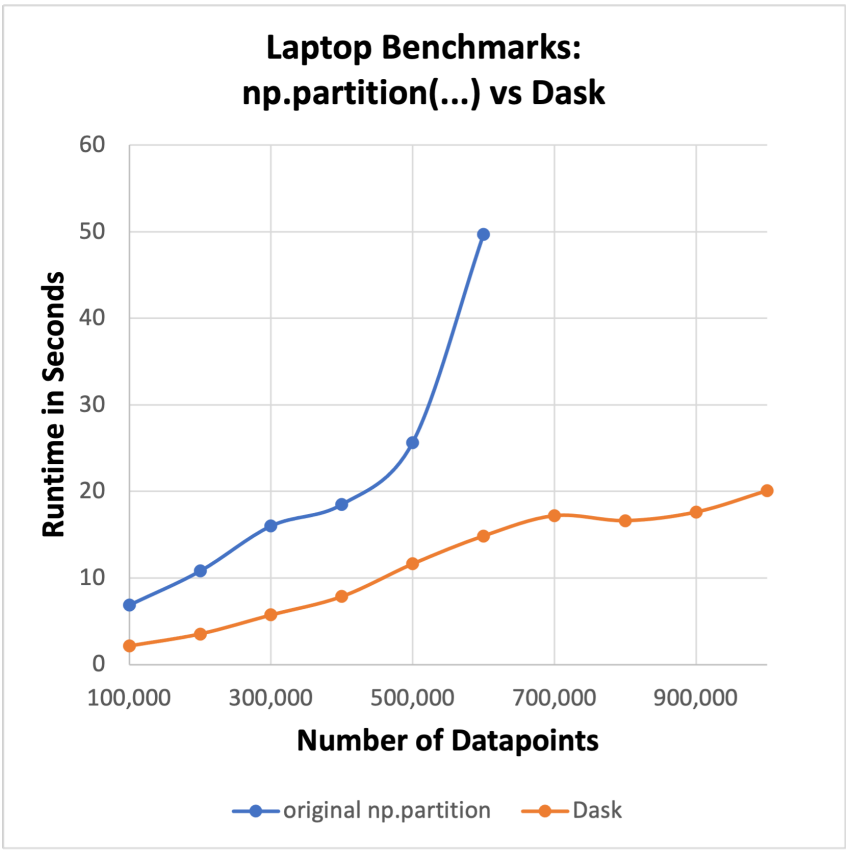


Figure 12 The runtime improvements of the DASK implementation. The upper plot compares results from desktop runs, whereas the lower plot shows laptop runs of the _kth_nearest_neighbor_dist(...) function.

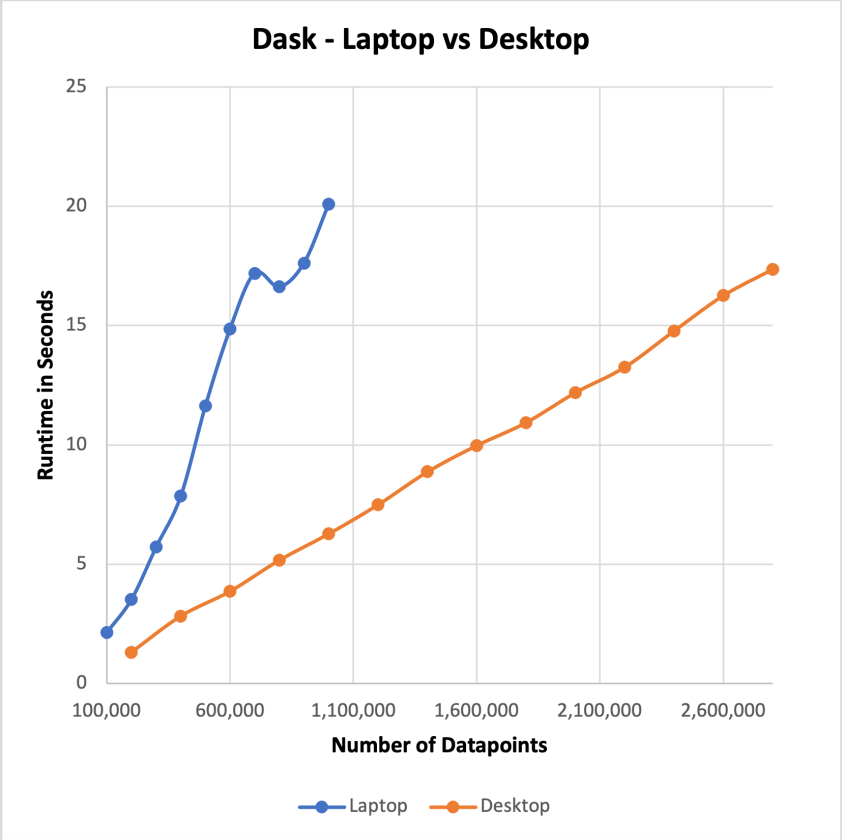


Figure 13 The runtime of the _kth_nearest_neighbor_dist(...) function, comparing the laptop and desktop run.

6. Results of Optimizing Distance Calculations

6.1. Baseline Implementations

So far, datafold has used a strategy pattern, choosing one of 4 distance backends:

- Brute Force
- SciPy cKDTree
- Scikit Learn ball-tree
- RDist

The base of this strategy pattern is a common interface, defining method signatures and attributes. An additional class "GuessOptimalDist" then checks which backends are available and, if no backend is specified in the backend parameters, tries to choose the best option. Also a selection of metrics is available, mainly Euclidean and squared Euclidean, while certain backends can support a bigger variety like Chebyshev, Hamming, etc..

All backends must implement a method called "pdist(...)", which takes the arguments X for data points and a cut_off distance, as it calculates pairwise distances between all points within X, which distances are lower than the specified threshold.
A second necessary function is "cdist(...)", which additionally takes a second set of points (Y). Distances are computed component wise for Y, in regards to the reference dataset X, again considering whether they fall below the cut_off distance.

This nomenclature is adapted from the existing scipy distance calculation methods, implemented in scipy.spatial.distance, which are currently also implemented as available backends.

All distance matrices returned by pdist must adhere to a number of points:

- The distance matrix is square
- The distance matrix is symmetric
- The diagonal of the distance matrix must contain explicit zeros
- Distance above the cut_off are not stored in the Compressed Sparse Row (CSR) matrix. Hence, implicit zeros are treated as "cut_off value exceeded"

The logic behind the backend selection is depicted in figure 14. As can be seen, whenever a cut_off is applied, RDist is the preferred backend, followed by the Scipy KDTree implementation. Only when metrics other than (squared) Euclidean are requested does the brute force approach calculate all distances and remove any distances above the cut_off.

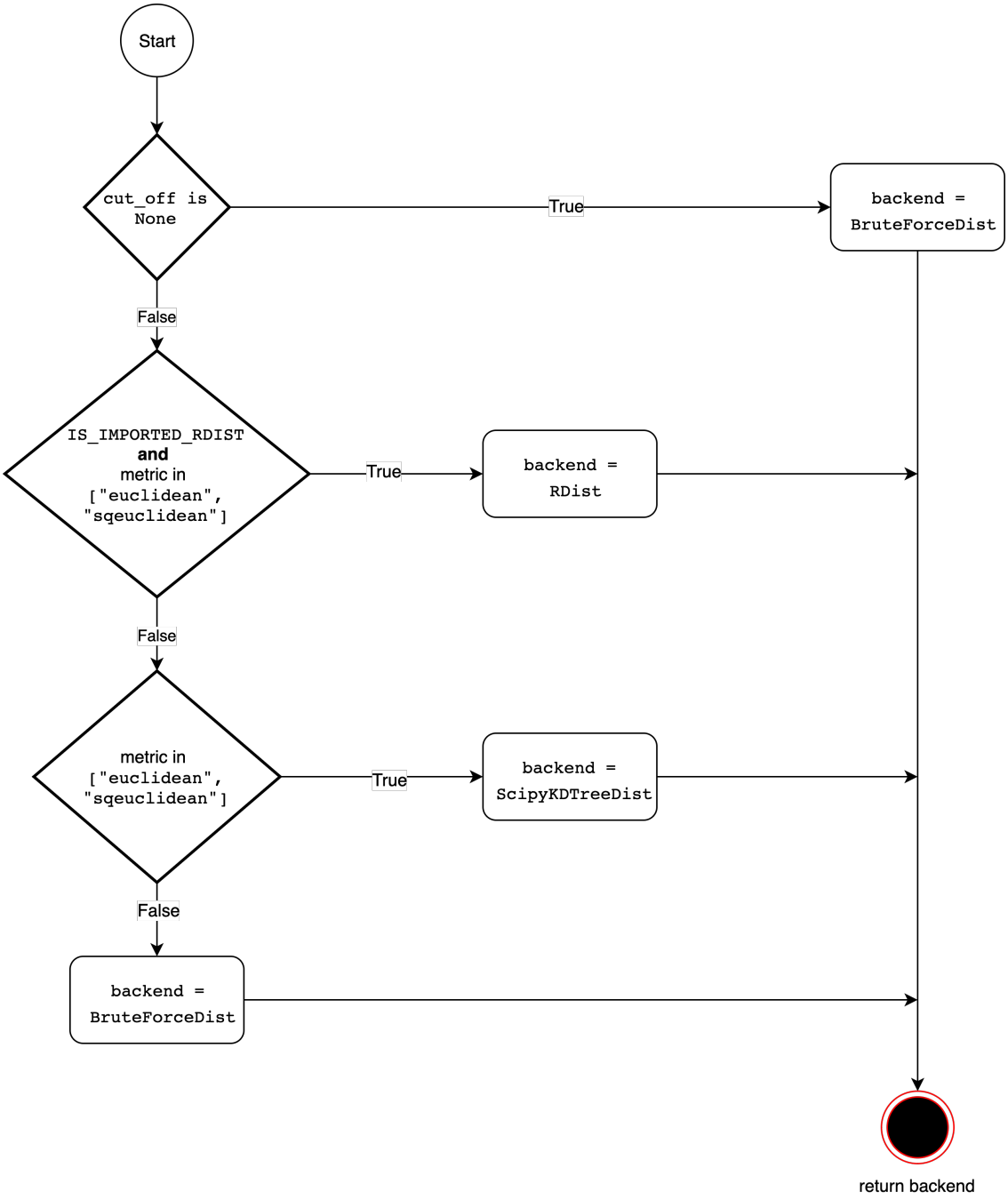


Figure 14 The original logic behind the selection of the distance calculation backend in datafold (simplified, some lines omitted).

6.1.1. Brute Force

The most straight forward approach is brute-forcing all distances. One apparent drawback of this approach is, that it calculates all pairwise distances, between any two points in the dataset, temporarily disregarding wether or not they are within the given cut_off. Looking at

the brute-force pdist(...) implementation in figure 15, it can be observed how some optimizations for specific use cases are already in place.
As observed in 5, the brute force pdist(...) implementation offers a faster, numerically slightly deviating version. Considering this requirement, brute-force pdist(...) either uses SciPy's pdist(...) implementation for numerically exact results (figure 15 line 11), or an scikit learn implementation called pairwise_distances(...).
Since both implementations are, as described, of brute-force nature, they do not account for a cut_off, creating a dense matrix, up until line 18 and 19, which convert the Numpy matrix to a sparse CSR matrix, setting any distances above the chosen threshold, to zero.

```
1  def pdist(
2      self,
3      X: np.ndarray,
4      cut_off: Optional[float] = None,
5      exact_numeric: bool = True,
6      **backend_options,
7  ) -> Union[np.ndarray, scipy.sparse.csr_matrix]:
8
9    if exact_numeric:
10         X = np.array(X)
11         _pdist = scipy.spatial.distance.pdist(X, metric=self.metric)
12         distance_matrix = scipy.spatial.distance.squareform(_pdist)
13     else:
14         # sklearn uses an numeric inexact but faster implementation
15         distance_matrix = sklearn.metrics.pairwise_distances(
16             X, metric=self.metric, **backend_options)
17
18         if cut_off is not None:
19             distance_matrix = self._dense2csr_matrix(distance_matrix,
     cut_off=cut_off)
20
21         return distance_matrix
```

Figure 15 Brute-Force pdist(...) implementation, adapted (omitting some comments and adding input sources).

In parallel, the brute-force implementation of cdist(...) uses the corresponding scipy.spatial .distance.cdist(...) function, and the same overloaded sklearn.metrics.pairwise_distances(...) method respectively.

6.1.2. SciPy cKDTree

The next backend implementation is a k-d tree, supplied by SciPy, more exactly, the scipy.spatial .cKDTree. This implementation is the go-to choice when the specialized RDist package is not installed and if the metric is either Euclidean or squared Euclidean (compare figure 14). As will be discussed in 6.3.1, a k-d Tree data structure generally fits the problem at hand, enabling efficient nearest neighbor and range queries, however its performance depends on the shape of the dataset.

The main application of this implementation consists of two function calls, as shown in figure 16.

```
1 kdtree = scipy.spatial.cKDTree(X, **backend_options)
2 dist_matrix = kdtree.sparse_distance_matrix(
3             kdtree, max_distance=max_distance, output_type="coo_matrix")
```

Figure 16 Setting up the cKDTree and calling the distance calculation function.

Immediate advantages of this implementation are, that it natively takes the cut_off into consideration with its max_distance parameter and outputs the resulting distance matrix in a sparse distance format, in COOrdinate Form (COO). From here, the required CSR formatted matrix can easily be derived.

By looking into the source code of the cKDTree implementation (see [32]), you can see, that it falls back to a C++ implementation with a Cython wrapper.

Also, the documentation (see [30]) states, that, since SciPy version 1.6, cKDTree remains implemented for compatibility reasons: "cKDTree is functionally identical to KDTree. Prior to SciPy v1.6.0, cKDTree had better performance and slightly different functionality but now the two names exist only for backward-compatibility reasons. If compatibility with SciPy < 1.6 is not a concern, prefer KDTree." ([30]). Hence changing the existing codebase from cKDTree to KDTree should not harm or benefit performance or change any of the functionality, as KDTree now wraps the previously superior cKDTree, but should still be considered for deprecation reasons and to keep up to the current SciPy recommendations.

The corresponding cdist(...) function uses the same methodology, creating two k-d trees instead of only one, as does pdist(...) (see figure 16, line 1), which distances are then calculated using the same overloaded sparse_distance_matrix(...) function.

6.1.3. Scikit Learn Ball-Tree

The scikit Learn Ball-Tree implementation uses another specialized data structure in form of a ball-tree. As the name suggests, the SklearnBalltreeDist backend class makes use of a scikit Learn implementation for its distance calculation. Due to its connection to the "Nearest Neighbor" problem - finding the closest neighbor up to a defined point - they share a certain problem domain. Conveniently, the used scikit Learn functionality allows for a "radius neighbor graph" query which perfectly fits the required use case.

How interlocked these problems are, can be seen in figure 17, where an actual "sklearn.neighbors.NearestNeighbors" object is used. Having fitted the dataset in line 5, the Near-

```
nn = NearestNeighbors(radius=max_distance,
                      algorithm="ball_tree",
                      metric=metric,
                      **backend_options)
nn.fit(X)
distance_matrix = nn.radius_neighbors_graph(mode="distance")
```

Figure 17 The core of the scikit Learn backend implementation. Note that max_distance requests the (modified) cut_off.

estNeighbor object offers the functionality to request a "radius_neighbors_graph(...)". While the modes parameter allows for the selection of a "connectivity" graph, containing boolean values, indicating a distance below the "cut_off", "distance" can be chosen instead, retrieving a sparse distance matrix in CSR format, that takes the given max_distance into account (see [27]).

The cdist(...) variant of this implementation takes advantage of the possibility to specify an additional array of query points for which distances should be calculated.

Internally, the scikit Learn framework, just like SciPy, uses a Cython C++ implementation for the ball-tree data structure and, depending on the use case, also on joblib.parallel. This can be seen, by inspecting the source files in [29].

6.1.4. RDist

Finally, RDist is a backend option, that, at the time of writing this thesis, is still under development. As it was specifically designed to efficiently calculate distances between points with a high dimensionality, it is expected to yield good results and therefor is the preferred choice due the backend selection logic that is implemented (see figure 14).

This is also noticeable when inspecting the required function calls to RDist, which consists of only two parts.

```
# build tree, currently not stored, backend options are handled to here.
_rdist = rdist.Rdist(X, **backend_options)

# compute distance matrix, these options are not accessible from outside
    at the
  # moment.
distance_matrix = _rdist.sparse_pdist(
    r=max_distance, rtype="radius", **self._get_dist_options())
```

Figure 18 Creating an RDist instance, specifying your dataset and requesting a sparse distance matrix with pairwise distances.

As you can see in figure 18, the retrieval of the sparse, pairwise distance matrix is straight forward and is of a similar structure like the backends that have previously been shown.

According to [18], RDist implements the C++ code of a k-d tree by mlpack using Cython, making the required functions callable from Python. This way, computationally intensive tasks, mainly high dimensional distance calculations, are performed inside the C++ domain.

6.2. Baseline Benchmarks

In order to make benchmarking a wide variety of cases easier, a sequence was implemented, which iterates through the benchmark suite diagonally, as shown in figure 19. As the cases get computationally more intensive from the upper left towards the lower right of the shown matrix, it makes sense to let the benchmarks run, from "easy" to "difficult". Since cases with low number of datapoints or low dimensionality are less likely to abort due to memory or runtime boundaries (if present), iterating through cases diagonally should result in more successful runs, than a line wise for loop.

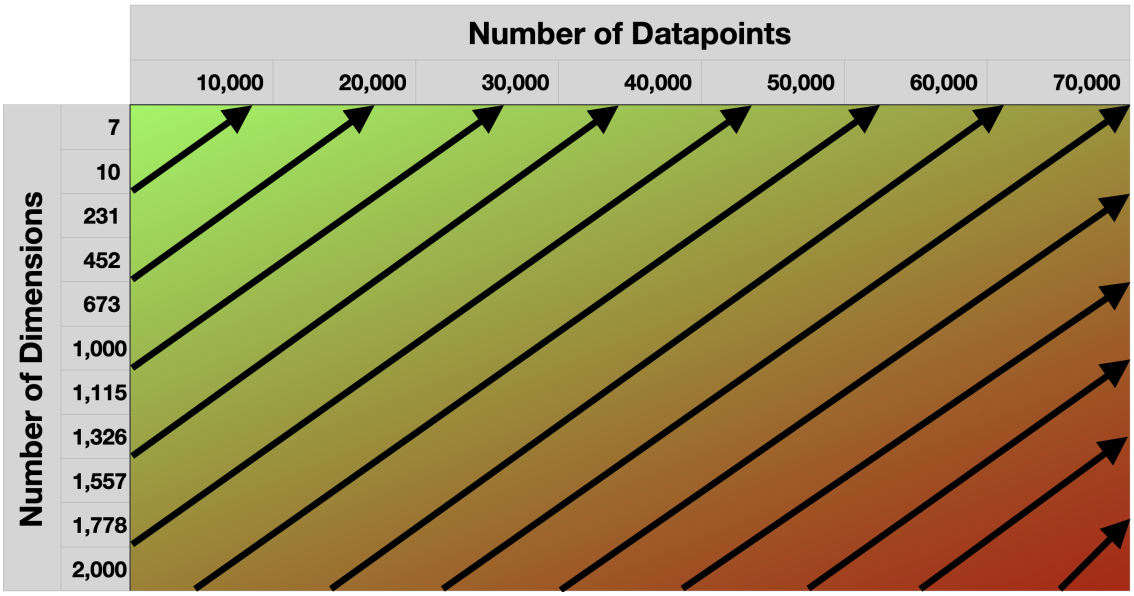


Figure 19 The automatic benchmark iteration starting from the upper left going to the lower right. The Y axis depicts the number of dimensions while the X axis is supposed to represent the number of data points. The background gradient represents the complexity of the cases from less (green) to more (red).

As a result, there are a total of 77 different variations of number of datapoints and dimensionality. Since every case has its own parameters in form of cut_off and epsilon, it was decided to include the optimize_parameter(...) function in every run (with DASK disabled). In order to preserve comparability, the random sate of the optimize_parameter(...) function, which usually generates a random permutation to select 1,000 datapoints at random was fixed, resulting in identical cut_off and epsilon for each of the 77 benchmark cases. Same holds true for the random generator used to generate the datapoints of the hypercube, resulting in a deterministic output for each case. This should eliminate any randomness in the program,

providing identical scenarios for multiple runs with different backends.
Further parameters were mostly kept at their defaults, such as the standard k=25 for _kth_-nearest_neighbor_dist(...). A Gaussian kernel was chosen, initialized with "n_eigenpairs=20". The eigensolver part has been disabled for this benchmark to reduce overall runtime, as we are concentrating on different parts of the code.

The following numbers are referring to the distance calculation only (in this case: the pdist method). The final function that is interface to the rest of the codebase is called "compute_-distance_matrix(...)". It includes the choice of the backend and other input checks and initializing. A part of this function is shown in figure 20. As can be seen, the selection of the backend and the required function is not part of the timing. Instead, only the relevant "distance_method(...)" is timed and the result is stored in an environment variable which is later accessed and saved in a .csv file.

```
backend_class = get_backend_distance_algorithm(backend)
distance_method = backend_class(metric=metric)
t1 = time.time()
distance_matrix = distance_method(X, Y, cut_off=cut_off, **backend_kwargs)
t2 = time.time()
os.environ['DF_DISTANCE_TIME'] = str(t2-t1)
```

Figure 20 Part of the "compute_distance_matrix(...)" augmented by time measuring mechanisms.

All depicted baseline benchmarks in this chapter were performed one the *Desktop Machine* as described in 5.2.1. The respective frameworks were installed using conda, including openMP, OpenMPI and openBLAS.

6.2.1. SciPy k-d Tree

Since it is the go-to variant so far in datafold (excluding the under development RDist), this baseline benchmark bears special importance. Figures 21a, 21b and 21c visualize the data in some different ways to put runtimes into perspective. Empiric observations of the resource monitor showed, that the SciPy k-d tree did not utilize multiple cores, with no explicit option to change a parameter like n_jobs.

It can be deduced from the plots, that the k-d tree scales near perfectly linear in the number of dimensions for the tested number of datapoints (compare figure 21c, left plot). Scaling in the number of datapoints however, shows a rather non linear runtime curve (compare figure 21c, right plot).

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	0.51	1.49	2.83	4.18	6.05	7.71	12.23
10	1.02	3.70	6.96	12.92	17.99	24.59	37.35
231	13.23	51.77	100.99	203.32	292.14	399.11	662.73
452	24.53	97.71	191.55	389.06	556.15	755.09	1,270.55
673	36.28	143.56	281.76	574.79	818.70	1,116.00	1,880.44
1,000	53.31	212.57	415.87	850.26	1,214.35	1,659.45	2,778.65
1,115	59.74	237.67	464.75	946.67	1,353.94	1,844.89	3,096.93
1,326	70.31	280.40	547.27	1,119.71	1,595.14	2,175.48	3,662.08
1,557	82.89	329.59	643.35	1,316.15	1,871.38	2,562.00	4,301.06
1,778	94.20	374.78	733.10	1,494.96	2,138.05	2,916.29	4,913.09
2,000	106.54	423.45	823.36	1,682.92	2,406.45	3,288.48	5,519.80

Figure 21a The raw runtime data of the orgiginal SciPy k-d tree.

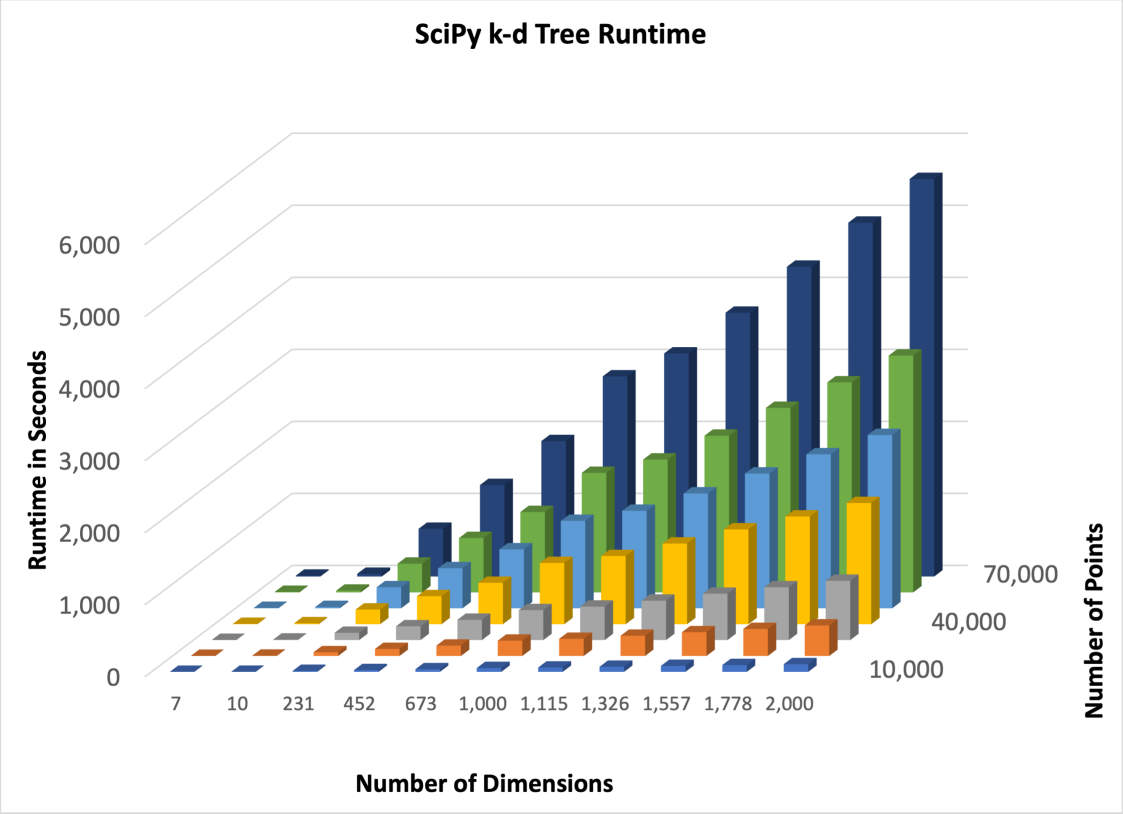


Figure 21b A 3D plot of the SciPy k-d tree runtimes based on number of dimensions and number of points.

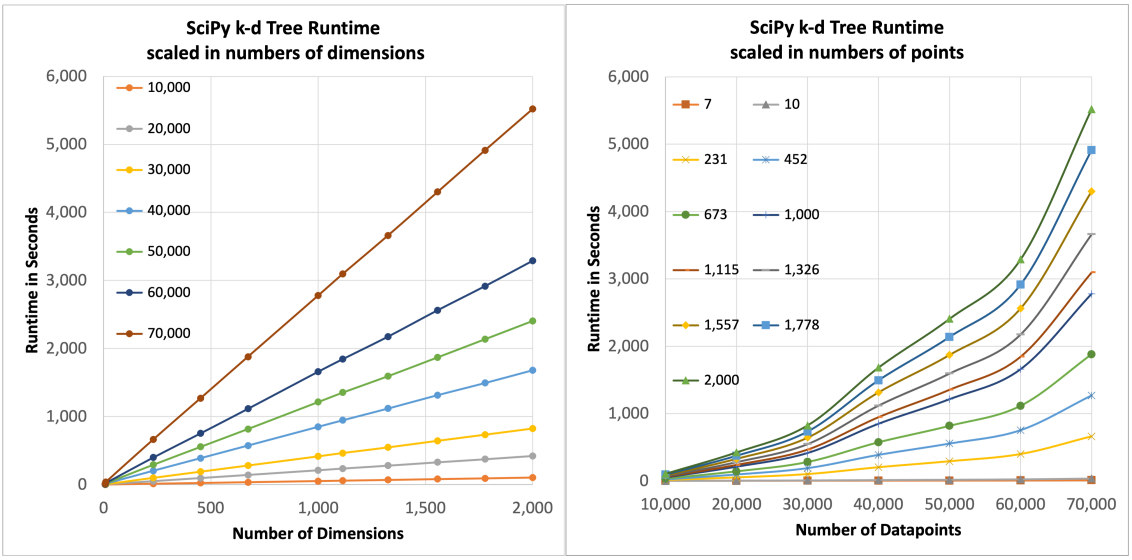


Figure 21c The left plot shows the SciPy k-d tree runtime in regards to an increasing number of dimensions. Different lines represent a different amount of points. The right plot shows the same data, but from the perspective of increasing the number of datapoints, where each line represents one dimension.

6.2.2. Scikit Learn Ball-Tree

Next, the ball-tree by scikit Learn will be evaluated.

For many cases, the scikit Learn ball-tree performs better than the SciPy k-d tree. Again, a near linear behavior can be seen increasing the number of dimensions, while scaling in the number of points causes an higher than linear impact. Empiric observations of the resource monitor showed, that the scikit Learn ball-tree uses only one core at first, but later utilizes all available threads. The n_jobs parameter was set to "all cores", i.e. -1.
The benchmark runtime of the scikit Learn ball-tree can be seen raw in figure 22a, or plotted in figures 22b and 22c.

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	0.63	1.49	2.99	3.70	5.92	7.54	10.33
10	1.37	4.04	7.77	10.83	15.25	20.98	26.47
231	4.39	16.24	35.41	65.20	107.55	152.39	214.56
452	5.14	22.64	58.33	111.40	181.91	265.30	398.19
673	10.16	30.63	80.27	153.64	247.54	445.31	619.12
1,000	10.18	49.16	119.86	247.05	440.46	673.01	991.67
1,115	11.29	52.24	135.75	283.59	478.71	750.96	1,110.08
1,326	13.33	63.65	174.95	351.54	593.16	896.52	1,354.89
1,557	21.70	79.41	217.06	439.69	740.04	1,142.24	1,624.78
1,778	26.19	95.21	243.71	546.73	853.12	1,296.63	1,901.31
2,000	30.75	123.23	299.88	615.90	1,014.17	1,477.85	2,262.57

Figure 22a The raw runtime data of the original scikit Learn ball-tree.

6.2.3. RDist

RDist holds a special position in this benchmark case, as the chosen hypercube example can be interpreted as a worst case example for the backend.
The RDist backend reaches its efficiency by projecting the dataset onto lower dimensions, making the distance calculations easier. It can therefor be said, that RDist runs most efficiently when the underlying dataset fulfills the manifold assumption well. Hence, great runtimes can be achieved on sample data like the s-curve or the Swiss-roll. The extend to which this manifold assumption is fulfilled for generic use cases with which the datafold framework might be confronted, varies from dataset to dataset, but is also a premise on which the whole datafold framework works.

Applying the hypercube benchmark example on RDist has proven this, leading to an increased memory usage and runtime of RDist. However since this should not be taken as a representative example of RDist, it was decided to not include the timings based on the hypercube into the baseline numbers. Following the installation guide of RDist, including mlpack and openMP, it was observed, that, similarly to the scikit Learn ball-tree, following a setup phase on only one core, all available cores are used. Observing the resource monitor and the logs of RDist confirmed, that all 16 threads were available to and used by it.

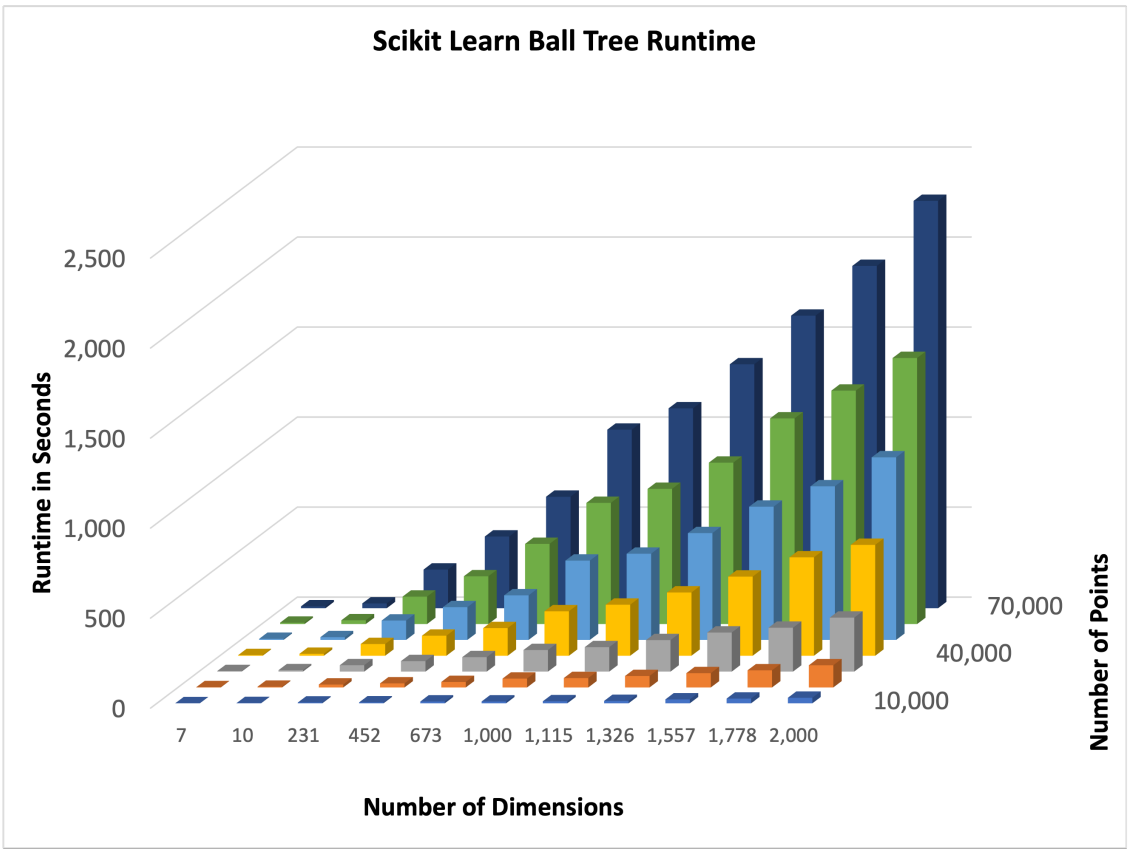


Figure 22b A 3D plot of the runtime based on number of dimensions and number of points.

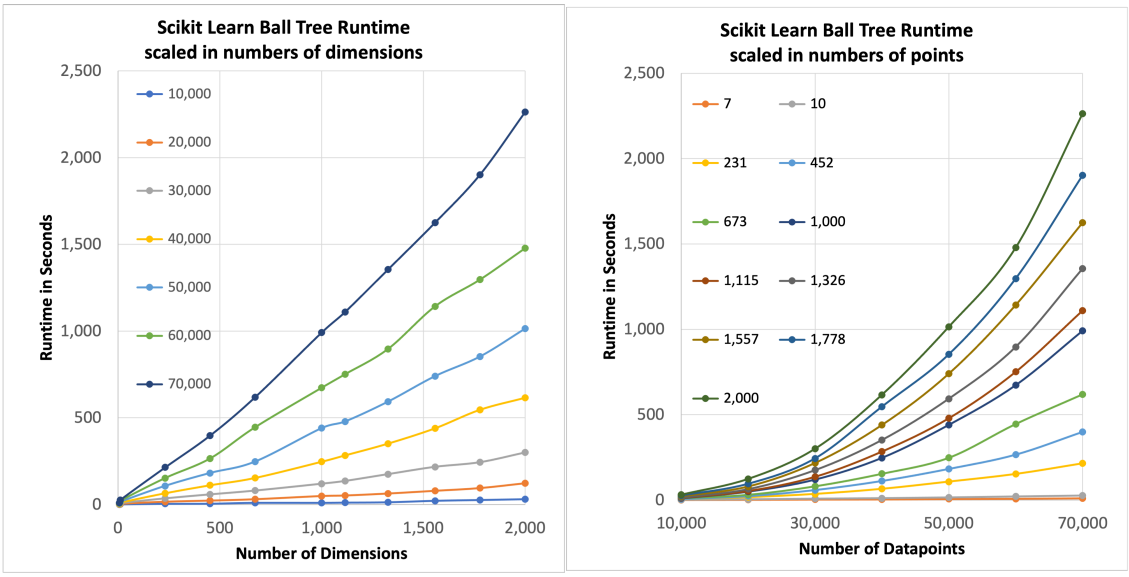


Figure 22c The left plot shows the ball-tree runtime in regards to an increasing number of dimensions. Different lines represent a different amount of points. The right plot shows the same data, but from the perspective of increasing the number of datapoints.

Instead, the attention should be focused on the final benchmarking results where the distance backends are tested with the MNist dataset. The real world dataset fits the RDist backend better, while also depicting a more realistic use case.

It should be noted, that RDist represents a major feature of datafold, which extent can not be done justice in this thesis as a wider variety of datasets need to be considered to give a final assessment. Correspondence with Daniel Lehmberg ([18]) gave an insight into the strengths of this particular backend, whose experience was, that it significantly out-performs the baseline implementations on a wide variety of datasets.

Hence, it was observed, that that the hypercube benchmark is rather unsuited for RDist, which on the other hand clearly out-performed any baseline implementation for the real world dataset MNist, as will be shown in 6.5.1.

6.3. Theoretical Background

[9] describes the task of fixed radius neighbor search as follows: Given a list of indices $I = 1...n, n \in \mathbb{N}$, a fixed radius $r > 0, r \in \mathbb{R}$, points $p_i \in \mathbb{R}^d$ for a dimensionality $d \in \mathbb{N}$ and $i \in I$ in an Euclidean space and a norm $|| \cdot ||$, "[t]he near neighbors of point i are the distinct points $j \in I, i \neq j$ within radius r denoted as:

$$N_r(i) = j \in I | j \neq i, ||p_i - p_j|| \leq r. \text{ "(see [9])}$$

This search is referred to as a "Fixed-Radius Neighbor Search". This problem itself has been researched for a long time. An early report ([3]) from 1975 lists a series of approaches and data structure, which, to this day, are commonly found.

Naively the easiest implementation would be a brute force calculation of all distances and then applying a filter on the distance array, removing all distances greater than r. Such an approach would need $\mathcal{O}(n^2)$ distance calculations (where n equals the number of datapoints), which can be seen as an upper bound. All mechanisms designed to solve the fixed radius neighbor search should provide a runtime, better than quadratic in n.

[3] states, that Cyrus Levintahl "used a fixed radius nearest neighbor search in his interactive computer graphic study of protein molecules" ([3] p.1). The field molecular science research nowerdays uses a well refined algorithm for this problem, namely the "cell list" algorithm which is a bucket sort approach. It achieves linear $\mathcal{O}(n)$ runtime in the number of datapoints ([9]). Modelling the ranges of forces between molecules, which get weaker as distance increases and, at some point, are considered 0, calls for fixed radius neighbor searches, usually in two or three dimensional space. The downside to this approach is, that it suffers when a high

number of dimensions is required, as the number of cells grows exponentially with growing dimensionality of the data. This makes it unsuitable for the application in datafold which requires a rather high number of dimensions.

It is apparent, that the problem at hand is connected to the k nearest neighbor problem. Efficiently getting a sorted list of closest points would allow an algorithm to iterate through potential neighbors from close to far and stopping, once a point falls out of the cut_off distance value. However this problem is also not free of the curse of dimensionality. On one hand it becomes more difficult to calculate distances and finding the nearest neighbor, while on the other hand the correlation between similarity and proximity becomes less reliable ([33]), to name only some of the problems that arise in higher dimensions. [17] presented in their findings that, for certain data distributions: " as dimensionality increases, the distance to the nearest neighbor approaches the distance to the farthest neighbor" ([17] p.217) further complicating kNN and fixed radius neighbor search problems.

Researching frameworks and implementations that are suitable for this task has shown, that the aspect that receives the most attention, is the underlying data structure in which the data points are stored. They commonly are designed for fast neighbor searches, attempting to reduce the number of distance computations and comparisons, reducing runtime. Hence some of these data structures will be covered in the following. Note that these are chosen based on their usage in available frameworks, as this was the focus of this thesis.

6.3.1. Data Structures for High Dimensional Data Storage

Hanan Samet explained different data structures and their relevance for various use cases in his famously known book "Foundations of Multidimensional and Metric Data Structures" ([23]). He stresses how queries involving distance calculations are especially compute intensive, while queries which do not require this metric, like point and range search are considerably "easier".

Implementing fast (fixed-radius) neighbor searching approaches often focuses on the underlying data structure in which the points are stored and can efficiently be queried for their neighbors. Various tree ([30], [27], [14]) and graph options (e.g. [20]) are available which are optimized for searching. While different kinds of data structures come with different recommendations regarding number of points in relation to dimensionality, whether or not the structure can be dynamic augmented or their query behavior, it makes sense to take a closer look into the most widely used options. Also, mainly being part of approximate nearest neighbor approaches, two techniques will be mentioned rather quickly in form of the k-NN graph and local sensitive hashing.

K-D Tree

Having been introduced in 1975 by Jon Bentley (see [2]) the k-d tree was designed to hold multidimensional datapoints. Note that the author intended for k to represent the number of dimensions (usually denoted: d) so a k-d tree containing three dimensional data points could be called a 3-d tree (compare [23]).
[2] suggests algorithms for range and nearest neighbor queries, latter of which reached a logarithmic runtime in the number of data points in the set. However it was originally: "efficient only for the Minkowski ∞ metric" ([2] abs. p.514, rel. p.6), later iterations by [10] improved the prior work. It falls under the category of binary search trees.

Since the k-d tree holds a very special place in the data structure space, being a well known go-to for multidimensional data, an overview on its properties and mechanisms will be given as well as why it may not necessarily be the best choice for data with a high dimensionality.

In its internal structure, a k-d tree subdivides the k-dimensional space into two parts, iterating through the axis for subsequent tree depths. Hence when working on two dimensional data (i.e. x, y), the root node would split the space in x direction, its direct two sibling nodes would split the space along the y axis, while their respective four nodes partition the space in x direction again (see figure 23).

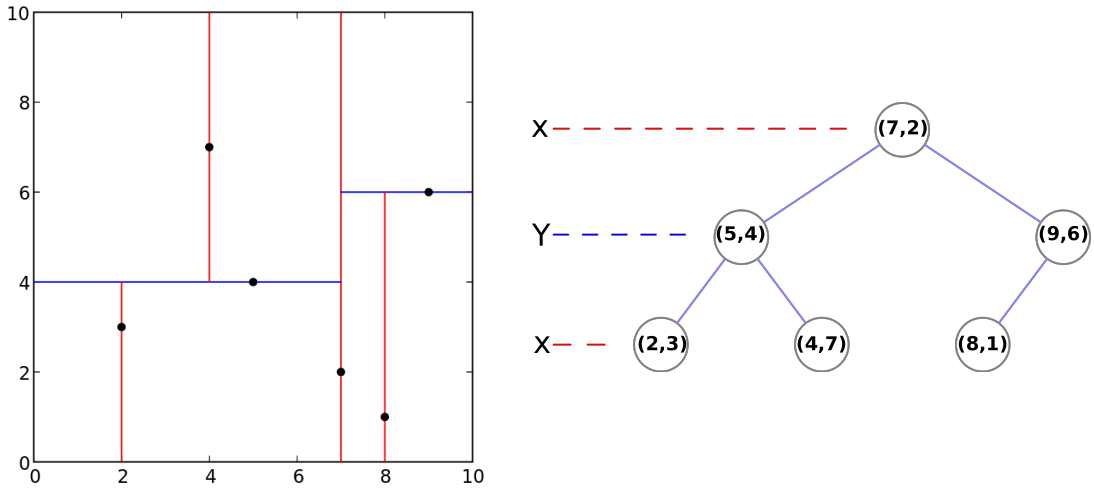


Figure 23 An illustration of a point-cloud (left) showing the lines along the axis for which each point divides the space and the corresponding tree structure (right) with the coordinates of the pints. Note that it is stressed how all points with equal depth split the space along the same axis. Graphics taken from [34].

This leads to the fact, that all nodes with the same depth divide the space along the same axis, as is depicted in a 2-d example in figure 23 (right). This comes with certain implications and complications for balancing operations, therefor a k-d tree need not necessarily be balanced. Variants that implement a mechanism that deviate from this aspect, are generally referred to as "adaptive k-d trees".

The runtime of operations on the tree is depicted in figure 24. The tree stores one (multidimensional) datapoint in each node.

Algorithm	**Average**	**Worst Case**
Space	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Search	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Insert	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$
Delete	$\mathcal{O}(\log n)$	$\mathcal{O}(n)$

Figure 24 Runtime for operations on a k-d tree (data taken from [34])

It should be mentioned, that postings (including developer forums, framework developers, [34] etc.) advise to have a dataset of size n, so that: $n \gg 2^k$ (where k again stands for the dimensionality). Intuitively, this makes sense, as the k-d tree splits every dimension at least in two, hence having less then 2^k elements would lead to a situation, where no more points would be left to further split dimensions in two parts. This explains why the k-d tree delivered decent performance for lower dimensions and many datapoints, as the previously stated in-equation is true. However, for use cases with $\sim$2,000 dimensions, a dataset of size $2^{2,000}$ ($\approx 1,1 \times 10^{602}$) would be needed to fulfill this recommendation, which is unrealistic.

[23] lists a couple of advantages of the k-d over implementations like a quad-tree (often used for 2D data, as each "square" is subdivided into four smaller chunks; analog in 3 dimensions: Oct-tree). Firstly the number of comparisons at each depth is reduced to one. The k-d tree is also more space efficient as a quad tree holds many null entries. The recursive complexity reduces, as each node only has two children, as opposed to four on the quad tree. However he also states one disadvantage that is rather harsh. While the quad tree requires multiple comparisons for each node, these can be parallelized. [23] therefor concludes: "Therefore, we can characterize the k-d tree as a superior serial data structure and the quad-tree as a superior parallel data structure" ([23] p. 49).

Ball Tree

One data structure which is used widely by frameworks like scikit Learn is the ball-tree. As explained in [22]: "A balltree is a complete binary tree in which a ball is associated with each node in such a way that an interior node's ball is the smallest which contains the balls of its children". While the interior nodes of the tree enable fast searches, all of the datapoints themselves are stored in leaf nodes. One "ball" is characterized by its center point and its radius. [22] also stresses, that sibling hyper-spheres can intersect with each other, which is a major difference compared to k-d or quad-/oct- trees.

[22] suggests an off-line construction algorithm working top-down that leans on the k-d tree setup. Similarly to the procedure described in 6.3.1, each step divides the collection of points into two groups, choosing the axis and splitting value: "in which the balls are most extended and the splitting value is chosen to be the median" ([22] p.9). This construction algorithm yields $\mathcal{O}(n \log(n))$ runtime (where n = number of datapoints).

[16] describes some differences and explains the workings of k-d and ball trees (see figure 25).

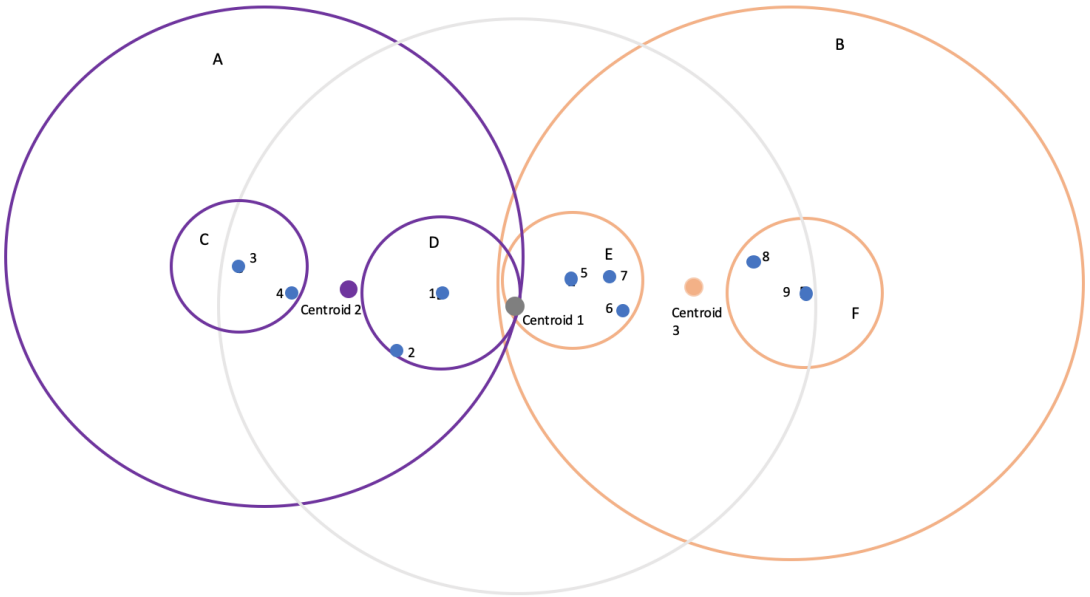


Figure 25 A 2D example of how a ball tree stores its datapoints in k dimensional spheres (graphic taken from [16]).

As shown in figure 25, the ball tree first picks a center point (Centroid 1) and encapsulates all datapoints in a sphere. It then looks for the points furthest away from Centroid 1, which in this case are points 3 and 9. It chooses point 3 as the Centroid for the next "cluster" (25) (A, purple circle) and point 9 for cluster B (orange) so, that all the points of the dataset are either in cluster A or B. Then, only considering the points in cluster A, it determines a new Centroid 2, which is done using the median. Having analyzed the furthest points as before, new clusters (C and D) are created the same way, after which the same routine is performed with the points in cluster B and applied recursively. The corresponding tree structure as presented in 25 is shown in figure 26.

Multiple resources, including [16], say, that the Ball tree is currently one of, if not the best, data structure to store high dimensional data and perform neighborhood queries on. While many (often empirical) case studies in online forums point towards implementations from SciPy and

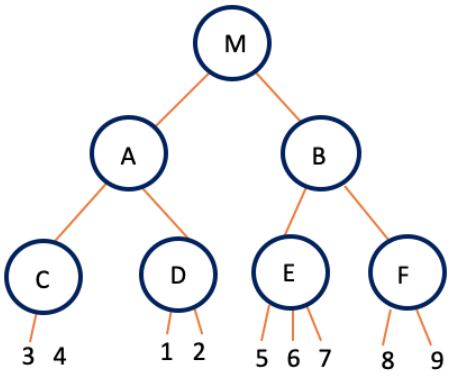


Figure 26 The corresponding tree structure to the point example shown in figure 25. Note that M denotes the sphere around Centroid 1 (graphic taken from [16])

scikit Learn.

K-NN Graph

Organizing the dataset in a graph, [1] explains a kNN graph in the following fashion: A kNN Graph is a directed graph where the datapoints are represented as vertices which share an edge with their true nearest neighbor. When querying the neighbors of a point, thanks to the graph and its nearest neighbor information, you can efficiently scan the "neighborhood" with a greedy search. When a new point is to be queried, the algorithm traverses over the graph and, due to the given neighborhood information, converges towards the query point until no more closer vertices can be found.

Frameworks like [26] construct this graph by starting out with a rather bad version of a graph, which is not yet really good for nearest neighbor searches. But by applying nearest neighbor searches on the points in the graph itself, using the "bad" graph as the index, the edges in the bad graph can be updated, refining the graph. Applying this technique over multiple iterations leads to an evermore accurate graph, which in turn improves accuracy and query time.

6.3.2. Approximate Nearest Neighbor and ANN Benchmarks

Since Nearest Neighbor(s) (NN) applications are one of the main use cases of high dimensional distance calculations, one oftentimes comes across developer forums advising the switch to Approximate Nearest Neighbor(s) (aNN) algorithms, focusing on the choice of NN approach, rather than backend distance calculations. This does not exactly model our problem of fixed radius neighbor searches, as aNN generally does not take a cut_off into consideration. Since aNN focuses on traditional nearest neighbor queries, respective frameworks most of the time take a training data set as an input to then enable fast queries on test / real world samples. By sacrificing accuracy, which might cause some results to NN queries to not be the actually closest element, speedups in the query time can be reached, which is

especially useful for higher dimensions. For these aNN applications, you could notice different choices for data structures, often choosing graph based solutions or Locality-Sensitive Hashing (LSH).

[1] has set up an impressive collection of ANN frameworks, benchmarking them on a selected range of well known sample datasets, such as: MNist, fashion MNist, GloVe or NYTimes.

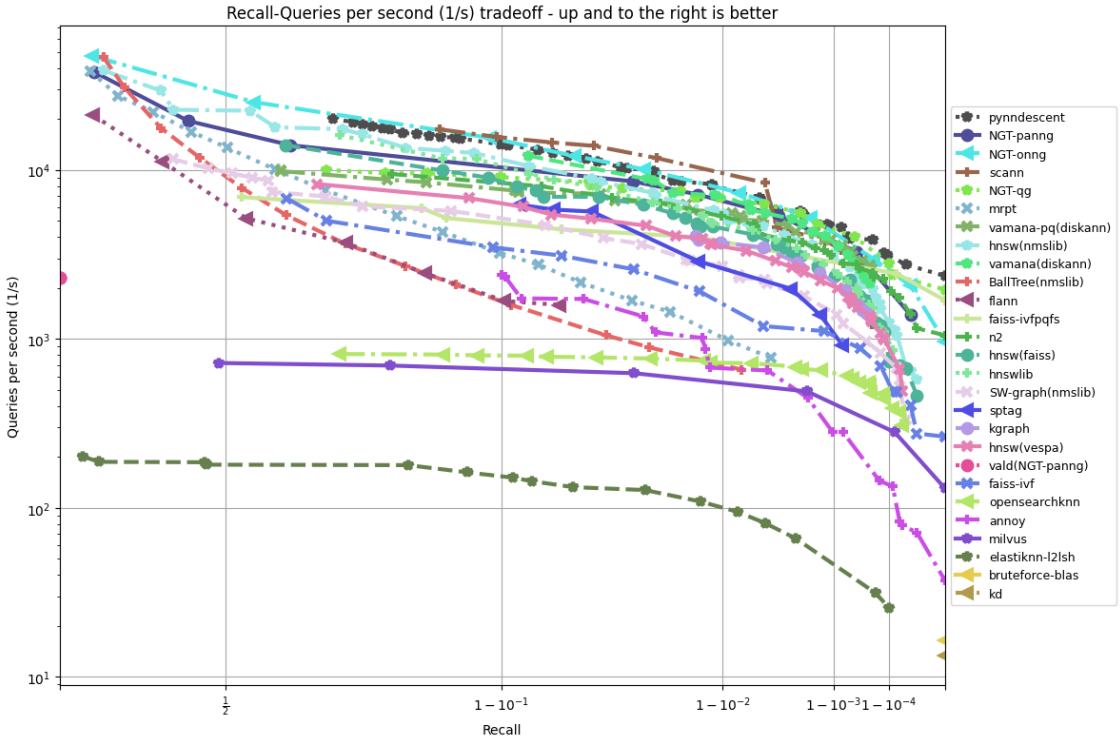


Figure 27 The benchmark results of the ANN Benchmark project ([1]) running the fashion MNist dataset. It shows the queries per second sustained over a given recall (a measure of accuracy). Hence higher values are better as well as keeping high values for lower recall values (i.e. right and top is best). Graphic taken from [24].

While sighting the frameworks that are benchmarked in [1], one especially sparked interest. "PyNNDescent" appears to be one of the few frameworks, that support calculation of distances to nearest neighbors for all points in the dataset. While PyNNDescent still insists on an amount of neighbors to find, not allowing for radius based neighbor search, it raises the question if a k ANN approach might be a viable solution for the distance backend. The PyNNDescent framework has shown to be one of the fastest ANN implementation featured in [1] (also see [26]). Furthermore, PyNNDescent is used by other frameworks like openTSNE (t-Distributed Stochasitc Neighbor Embedding) or UMAP (Uniform Manifold Approximation and Projection), which might indicate, that it might be suitable for datafold as well. Its capabilities and current uses were the motivation to consider it in our testing.

This however would require a new heuristic, as the cut_off approach obviously does not result

in the same number of neighbors in the distance matrix, as PyNNDescent would deliver. This happens still disregarding the inaccuracy of the approximate nearest neighbor approach. Choosing a good value for the number of neighbors to be retrieved for each point is not a trivial task and would require some major design choices. Finally the resulting distance matrix would not necessarily equal the one a k-d tree would generate for an identical problem, which makes comparing the two approaches more difficult.
Still, since the optimize_parameter(...) function gives rather an approximative value, finding a meaningful cut_off remains to be the task of the user. It should be noted, that in our talks with users of the framework, we encountered use cases, in which the optimize_parameter(...) is not called at all. Instead a suitable cut_off is chosen by the user directly.

Despite these obstacles, PyNNDescent could be a good option for use in datafold. It allows to specify "pruning" factors which can benefit runtime at the cost of accuracy. Exposing these parameters to the user, as well as letting them choose the number of neighbors, might also be an option, confronting them with known concepts of machine learning and offering the choice between cut_off and number of neighbors to be considered.

To end this section on literature research, we can conclude, that the field of efficient nearest neighbor / fixed radius neighbor searches is still topic of ongoing research. While some data structures are optimized for handling multi-dimensional data, many struggle from the general implications of high dimensionality (i.e. the curse of dimensionality) and its effects on distances, distance concentration and distribution of the datapoints themselves. To quote the documentation of SciPy, one of the leading scientific calculation frameworks:

> For large dimensions (20 is already large) do not expect this [(the k-d tree)] to run significantly faster than brute force. High-dimensional nearest-neighbor queries are a substantial open problem in computer science.
>
> - SciPy cKD-Tree Documentation (see [30])

6.4. Implementation

6.4.1. Framework availability

Given the fact, that this thesis concentrated on multiple different problems which come with an increased overhead, e.g. researching multiple areas of literature, it was decided to stick to frameworks, that are already available in Python. Personal correspondence with Daniel Lehmberg (see [18]), developer of the RDist backend brought up the fact, that fast tree structures exist also in the C++ machine learning framework MLPack, some of which are used in

RDist. Providing a wrapper to those methods could lead to good results, however was not the goal of this thesis.

Finding a suitable framework, that would deliver on efficient fixed radius neighbor search proved to be difficult, as both, calculating distances in high dimensions as well as high dimensional nearest neighbor search, appear to be an open problem, while vastly important for the data sciences and generally heavily researched.
Hence, a big part of this thesis needed to be trial and error, testing different frameworks, that would lead to the desired runtime improvements. A direct consequence of this was, that a great number of trial implementations were tested and disregarded. Since there would not be a major added value from going deeper into these frameworks, they will be covered rather quickly later on, in order to focus on the respective implementation that produces the best results.

While many solutions exist for k-Nearest-Neighbor and approximate nearest neighbor problems, they usually do not support querying all points that fall within a certain radius to the query, but rather the amount of nearest neighbors you want to find. While there might be an approach, that gathers the information of how many neighbors fall within the cut_off of each point its efficiency remains questionable. Since the assessment on which distances need to be calculated for each point, oftentimes relies on distance calculations itself, it would impose a redundant overhead to calculate certain distances multiple times.
By the end of this project, the idea came up to circumvent unnecessary distance calculation and gather the indexes of points that are within each others cut_off distance using local sensitive hashing. Thus, a framework like FALCONN, which implements this technique, could be used to generate this information, which would then have to be used to calculate the distances. Since it is not certain, that this approach would produce good runtime results and its implementation overhead would have interfered with the scope of this project, it will not be sought after in this thesis. Furthermore, correspondence with the ANN benchmark developers did not encourage further investigations regarding this framework (more in 8.2).

Ultimately, many indicators pointed towards SciPy and scikit Learn frameworks for fast distance calculations in Python. As presented in 6.1, some of the distance backends that come with datafold already use these libraries. As the mentioned frameworks offer multiple classes and functions (with different parameterizations), that share internal structures, different variations of those were assessed.

6.4.2. Radius Neighbor Transformer

One class from the scikit Learn framework, that appeared to be especially suited for this application came into view as the so called: Radius Neighbors Transformer (RNT). Its description:

"Transform [a dataset] X into a (weighted) graph of neighbors nearer than a radius." ([28]) fits the problem at hand perfectly.
While the existing "SklearnBalltreeDist" implementation does things very similarly as the RNT, choosing the latter could be considered a more "sustainable" choice, as it ensures easy interoperability to the scikit Learn framework. As opposed to generating a ball tree and performing calculations on it, using the RNT objects better fulfills the object oriented black box approach, offering a service (in this case fixed radius nearest neighbor search). Thus, when scikit Learn receives an update, that especially efficiently implements a new fixed radius nearest neighbor search, the interface of the RNT is likely to remain the same, offering a more performant backend, solely by updating scikit Learn.

The implementation of the RNT is presented in figure 28. Having adjusted the cut_off to the requested metric, the radius neighbor transformer object is instantiated. It features two modes: connectivity and distance, generating a boolean or distance matrix respectively. The configuration of this object has shown to majorly impact the resulting runtime.
Here, as opposed to restricting oneself to a ball tree implementation, the responsibility of choosing a suitable approach is passed to the scikit Learn framework, by setting the algorithm option to *auto*.

```
1  metric, cut_off = self._map_metric_and_cut_off(cut_off)
2
3  #Setup RNT and get distance matrix
4  max_distance = self._numeric_cut_off(cut_off)
5  rn = RadiusNeighborsTransformer( mode='distance', radius=max_distance,
       algorithm="auto", metric=metric, n_jobs=-1, **backend_options)
6  rn.fit(X)
7  distance_matrix = rn.radius_neighbors_graph(mode="distance")
8
9  #Make matrix symmetrical
10 distance_matrix.data[distance_matrix.data == 0] = self._invalid_dist_value
11 distance_matrix_triu = scipy.sparse.triu(distance_matrix, k=0)
12 distance_matrix = distance_matrix_triu + distance_matrix_triu.T
13
14 if self.metric == "sqeuclidean":
15     distance_matrix.data = np.square(
16         distance_matrix.data, out=distance_matrix.data)
17 distance_matrix = self._set_zeros_sparse_diagonal(distance_matrix)
18 return distance_matrix
```

Figure 28 The implementation of the radius neighbor transformer.

Line 6 calls the fit(...) method, passing the dataset to the newly generated object. The following line 7 retrieves the sparse neighborhood matrix, containing the distances in CSR format. The empiric accuracy of the results, compared to the SciPy k-d tree, maxes around $4{,}0 \times 10^{-14}$ with a mean deviation (of non zero values) of $6{,}3 \times 10^{-16}$ for Euclidean distances. One slight downside is, that the returned distance matrix is originally not perfectly symmetrical, deviating around the range of accuracy. Hence, the upper trigonal matrix is taken and a symmetric matrix is built in lines 12 and 13 by adding the upper trigonal matrix to its trans-

posed version. Note how in line 12 the distance_matrix_triu stores the trigonal upper part of the distance_matrix including the diagonal. Since the diagonal does not include explicit zeros, it will not intervene with the following transformations. Since the addition of the two halves removes all explicit zero values, it is important to set true zero values (occurring for duplicates) to the invalid_dist_value, as is done in line 10, which later on will be turned into a zero again, when calling _set_zeros_sparse_diagonal. Forcing this symmetry does not harm the result, as the matrix entries are supposed to be 0 and non zero, symmetrically (as lies in the nature of fixed radius nearest neighbor search). Considering that the deviations are small anyways, the distance from point A to B must equal the distance from B to A.
Finally, the results are squared if the requested metric was squared Euclidean. As the radius neighbor transformer does not include zero distances along the diagonal, they are added after which the final distance matrix is returned.

6.4.3. PyNNDescent

Approximate Nearest Neighbors - Implications

Considering that approximate nearest neighbor approaches were much more represented than fixed radius neighbor search frameworks, it makes sense to test at least one candidate of this class. Some of the implications of moving from a range based NN (cut_off), to a k-NN (number of neighbors) approach have already been lightly addressed in 6.3.2. While it is not guaranteed, nor likely, that both method produce numerically identical results (out of the box), they both serve the main purpose for the diffusion maps algorithm, transforming the dense case into a sparse one by reducing the entries in the distance matrix to those, which fall within a given proximity of one another.

One major hurdle in the implementation of a kNN approach is the fact, that such an algorithm does not guarantee a symmetrical matrix. Intuitively, picture a dataset containing a cluster of 10 points and one outlier. When querying the 8 closest neighbors for each points, all points within the cluster have each other as the closest points, while the outlier only has neighbors inside the cluster.
Since the datafold design is built around a symmetrical distance matrix, it would take a considerable amount of changes within the program and would finally not allow for the usage of a symmetrical eigensolver. Still the primary goal first of all is to assess whether or not such an approach would deliver good runtimes for distance calculations.

Therefore, an alternative path was chosen to implement PyNNDescent as a distance backend. This was done following the requirements towards a distance matrix, which are:

- Square
- Symmetrical

- Zeros along the diagonal
- True zeros indicating duplicates

As a consequence, this is no strict kNN approach, making the matrix symmetrical and therefore altering the raw kNN results of PyNNDescent. The mechanics to the PyNNDescent pdist functions will be explained in the following.

Implementation

There are multiple ways in which the matrix can be forced to be symmetric, each with their respective (dis-)advantages.
The method, that was used and depicted in 6.4.2 hard-copied the values of the upper trigonal part of the distance matrix onto the lower part. This is non critical for a distance matrix that was generated using a fixed radius nearest neighbor search, as it is supposed to be symmetrical anyway, with deviations occurring only within a small error. Therefor, for fixed radius nearest neighbor searches, the distance matrix should always be symmetrical in the regards of whether the entries are zero or non zero. This leads to the fact, that mirroring the upper trigonal matrix onto the lower part should not erase any non-zero values.
This way of making the distance matrix symmetric could be referred to has a "hard" symmetry, as it would erase non-zero values, if some preconditions were not fulfilled. Figure 29 shows the concept along with an example of how the matrix is made symmetric to compensate for slight numeric deviations that might occur in the distance calculation. Note that the deviations shown in figure 29 are exaggerated. The deviations that were observed were much lower (mean: 6,3$\times 10^{-16}$ for euclidean).

The main advantages of this approach are, that it is easy to implement (as can be seen in figure 31 lines: 37 and 38) and does not require any comparisons or checking for zero entries, as it is simply a hard copy.
A major disadvantage however is, that it relies on the strict precondition, that the original matrix should be symmetric in regards to zero and non zero entries. If the original matrix does not fulfill this property, non zero entries might be overwritten by a zero. This would still make the matrix symmetrical, but would also erase actual and desirable distances. Searching for a good compromise between calculation time and quality, it makes no sense to erase some of the calculated distances by making the matrix symmetric and losing accuracy in doing so.

With the ambition to test an aNN approach, it can not be assumed that the distance matrix returned by PyNNDescent is symmetric, as mentioned above. In order to preserve the calculated distances and still return a symmetrical matrix, a more advanced approach has been chosen.
For obvious reasons, the concept of copying distance values from the upper trigonal part of

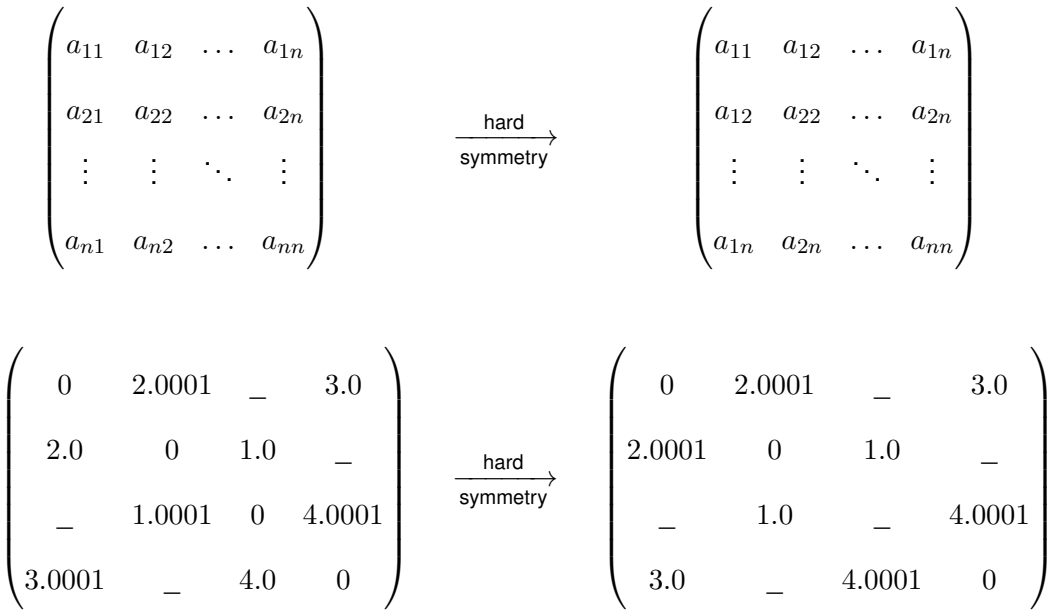


Figure 29 An example of how the distance matrix is made symmetric for the fixed radius nearest neighbor search. Note how non-zeros values remain non-zero due to the inherent symmetry of the radius neighbor search. Zeros represent actual zero distances (i.e. duplicates); _ means: out of cut_off bounds

the matrix into the lower (and vice-versa) is still required. However in order to preserve non-zero entries, this is only performed, for the non-zero entries in the distance matrix. This way, the number of non-zero entries only increases in the process of making the matrix symmetric, preserving calculated distances and accuracy. Hence, most importantly, no calculated non-zero entry is removed. The difference between the two approaches is shown in figure 30. Note that the 3.0 on position 2,1 is not erased in the soft case, instead it replaces the _ entry in 1,2.

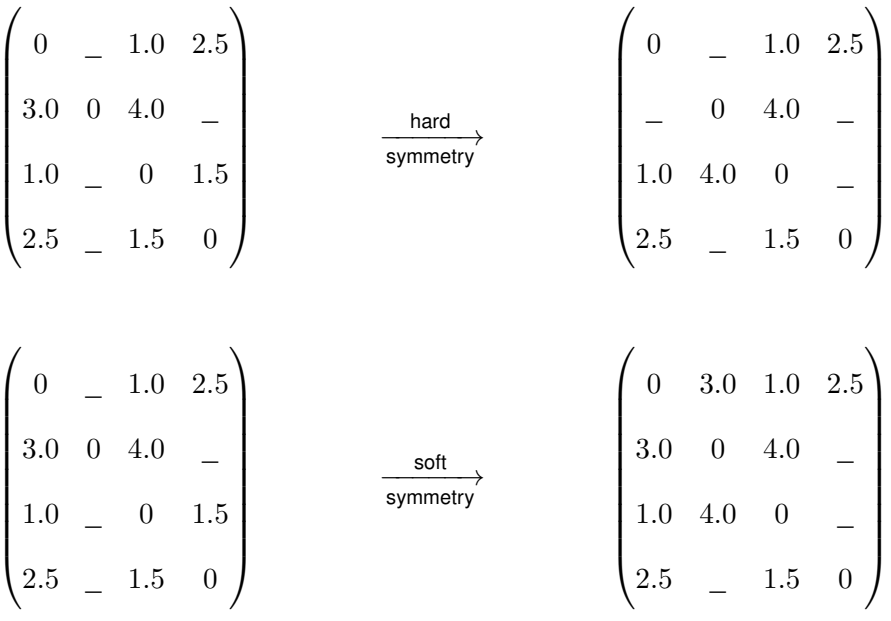


Figure 30 A comparison on the different approaches to make the distance matrix symmetric for a fictional aNN example. Note how positions 2,1 is erased in the hard symmetry variant, but not in the lower, soft case. Again zeros represent duplicates and _ stands for points that are disregarded due to their distance.

```
1  def pdist(self, X: np.ndarray, cut_off: Optional[float] = None, **
       backend_options ) -> scipy.sparse.csr_matrix:
2
3      soft_symmetry = True
4
5      max_distance = self._numeric_cut_off(cut_off)
6      metric, cut_off = self._map_metric_and_cut_off(max_distance)
7
8      index = pynndescent.NNDescent(
9              X,
10             n_neighbors=1000,
11             diversify_prob=0.0,
12             pruning_degree_multiplier=3.0,
13             metric= metric
14         )
15     distance = index.neighbor_graph
16
17     rows = np.asarray([np.full(shape = len(elem), fill_value = index) for
18            index, elem in enumerate(distance[0])]).flatten()
19     cols = distance[0].flatten()
20     data = distance[1].flatten()
21     data[data == 0] = self._invalid_dist_value
22     data[data >= cut_off] = 0
23     distance_matrix = scipy.sparse.csr_matrix((data, (rows, cols)),
24          shape=(X.shape[0], X.shape[0]), dtype='float64')
25     distance_matrix.eliminate_zeros()
26
27     # Make Matrix symmetric
28     if soft_symmetry:
29         rows, cols = distance_matrix.nonzero()
30         nz = set(zip(rows, cols))
31         ntz = set(zip(cols,rows))
32         fillers = np.array(list(ntz-nz))
33         if fillers.shape[0] != 0:
34             distance_matrix[tuple(np.transpose(fillers))] =
35                 distance_matrix.T[tuple(np.transpose(fillers))]
36     else:
37         rows, cols = distance_matrix.nonzero()
38         distance_matrix[cols, rows] = distance_matrix[rows, cols]
39     distance_matrix.data[distance_matrix.data == self._invalid_dist_value]
40          = 0
41     if self.metric =="sqeuclidean":
42         distance_matrix.data = np.square(distance_matrix.data)
43
44     return distance_matrix
```

Figure 31 The implementation of the PyNNDescent pdist function.

Figure 31 shows how PyNNDescent has been integrated to calculate a distance matrix, that conforms to the given requirements.
Line 3 stems from the development phase and allows you to choose between the symmetry mechanisms described above. The following lines 5 and 6 adapt the correct cut_off and metric. The setup of the index in line 8 to 14 specifies all information that PyNNDescent needs to return the distances as desired. Note that parameters like pruning_degree_multiplier and diversify_prob influence the performance and accuracy of the approximate nearest neighbor approach. For this purpose, they were set to produce accurate results, sacrificing runtime performance. Starting the distance calculation of all points to their n_neighbors closest neigh-

bors, the complete neighbor_graph can be queried, as is done in line 15. As PyNNDescent returns the result in a special format, the data needs to be prepared in order to create a CSR matrix in line 23 and 24. Before however, true zeros are set to the invalid_dist_value in line 21 as is known from existing backends, after which all values above the cut_off are removed. Whether or not this should be done is debatable, in this case, maximum comparability to the existing backends was sought, hence the cut_off is considered. These explicit zeros (due to the application of the cut_off) are removed in line 25.

At this point the matrix needs to be made symmetric. Since the "hard" symmetry case is simply a hard copy of values, mainly the "soft" symmetry case will be elaborated upon here.
At first, in line 29, all indices of non-zero entries in the CSR matrix are queried. These two arrays are "zipped" and converted to a set (see line 30), which leaves a set of tuples representing index pairs of non zero entries in the distance matrix. Same is done in reverse order in line 31. This is done to represent the index pairs that the transposed distance matrix would have. Subtracting one set from the other leaves the indices, that need to be filled in with the value from the transposed distance matrix. If this is the case, the copying for the detected indices is performed in lines 34 and 35.
Finally the true zeros, which were set to the invalid_dist_value in line 21 are converted back to true zeros in line 39. The result is squared if the squared-euclidean metric is desired and returned.
Making the matrix symmetric does take some time, as will be assessed in the benchmark comparisons, however, it makes kNN approaches compatible to the given sparse and symmetric requirements.

Accuracy of the Implementation

Due to the differences described above, it is unknown which effect a kNN approach has on the numerical accuracy. Therefore a test has been performed, comparing the numerical performance for different number of neighbors values to a (SciPy) k-d tree reference in form of comparing the resulting eigenvalues.

Base of this numerical test was an s-curve generated by scikit Learn. The randomness of the s-curve was fixed with a random_state of 3 and fixed to 60,000 points. The optimize parameter function was activated in the k-d tree reference with a random_state of 0.
During the k-d tree reference run, the number of nonzero values was read and saved as an orientation. A total of 1363062 nonzero values was recorded in the 60,000 rows long distance matrix. This equates to an average of 68.1 non zero entries per row (which were rounded down to 68 for the following steps). Also the 20 biggest eigenvalues were stored for future reference.
To now assess the accuracy depending on the number of neighbors which PyNNDescent takes into consideration, multiple runs were conducted with PyNNDescent's n_neighbors pa-

rameter set to 17, 34, 51, 68 and 85 which equates to 25%, 50%, 75%, 100% and 125% of the k-d tree average numbers of neighbors per row.
As can be seen in table 1, the error drops quickly, as is plotted in figure 32. It can be seen, how increasing the number of neighbors in PyNNDescent decreases the error by a magnitude of 10 for the first four values, however a further increase to 85 (i.e. 125% of the average k-d tree reference value: 68) shows the biggest improvement in accuracy.

n_neighbors	L2 Norm of difference to k-d reference
17	2.976E-03
34	5.053E-04
51	6.905E-05
68	4.317E-06
85	4.589E-9

Table 1 The deviation from the reference k-d tree eigenvalues. The 20 biggest eigenvalues were stored as a vector and subtracted from the k-d reference values. The right column depicts the L2 norm of the differences to the k-d reference. As expected, the error drops as a number of neighbors increases, approximating the k-d tree reference.

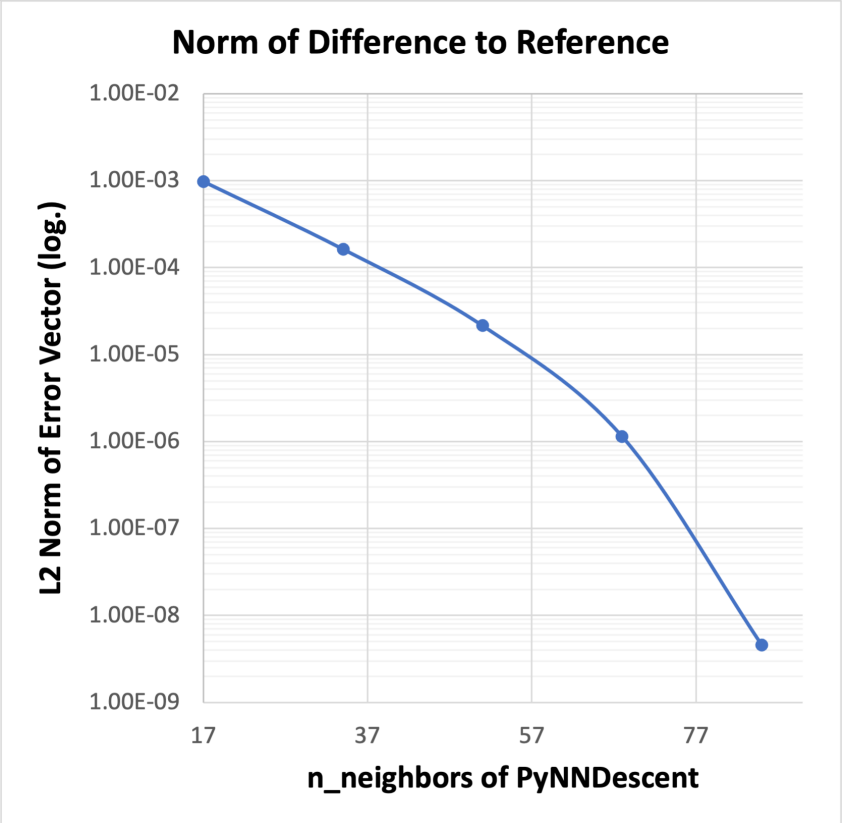


Figure 32 A logarithmic plot of the data shown in table 1. You can clearly see how the error drops especially quickly for n_neighbors greater than 68.

6.5. Benchmark Results and Comparison to Baselines

6.5.1. Radius Neighbor Transformer

Hypercube Benchmark Comparison

Applying the benchmark suite described in 6.2 to the newly implemented Radius Neighbor Transformer, significant improvements were noticeable, especially for higher dimensions. The empirically observed thread utilization showed frequent and quick jumps in the utilization between 1 and all 16 threads throughout (resembling a sawtooth pattern).

The results shown in figure 33a, 33b and 33c already show greatly reduced runtimes over the baselines. For this specific benchmark case, the hypercube, it is the fastest backend that has been tested for most cases, especially for dimensions greater 10. Considering all distance backends that were tested using the described benchmarking suite, only the originally implemented scikit Learn ball-tree was faster in some lower dimensional cases, as is color indicated in figure 33e and figure 33d.

To put the runtime improvements further into perspective, figure 33d shows the quotient of $\frac{\text{runtime RNT}}{\text{runtime sklearn balltree}}$ for each benchmark case, while figure 33e shows the corresponding absolute time saved. You can see how the performance gains is especially big for higher dimensions. For dimensions 7 and 10, the original ball-tree was faster within a few seconds, which become negligible considering the performance gains in higher dimensions. Note that is possible, that the *auto* parameter might have chosen the ball-tree backend, since different runtimes might also be caused by the forced symmetry routine.

Naturally it would be interesting to know which backend algorithm is chosen for a given case. The respective runtimes for a 2,000 dimensional data with 70,000 datapoints have been run with all available algorithms fixed to compare their runtime. The result of this is shown in table 2. Due to the similar times of the *auto* and *Brute Force* run, it can be inferred, that the *auto* run chose the *Brute Force* backend. The small deviation in runtime can be traced back to macro-effects (i.e. Operating System (OS) or background applications).

Scikit Learn RNT Algorithm	**Runtime in Seconds**
KD-Tree	2,351.95
Ball-Tree	2,107.39
Brute Force	157.37
Auto	157.65

Table 2 Runtimes after fixing the algorithm of the RNT for a fixed benchmark example (2,000 dimensions, 70,000 samples). Since the "brute force" and "auto" variants show an almost identical runtime (note that these were separate runs), it can be inferred, that the RNT chose the brute force algorithm in the "auto" case.

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	0.78	1.79	3.18	4.05	5.96	7.41	11.73
10	1.95	5.59	8.96	13.91	18.18	23.83	34.61
231	3.41	9.65	18.90	29.95	47.15	65.50	83.20
452	3.02	10.03	23.39	37.30	54.57	69.82	93.56
673	3.47	11.42	25.59	39.55	55.09	76.27	107.91
1,000	3.54	11.68	24.94	41.67	63.42	87.82	119.51
1,115	4.26	12.64	28.26	42.40	67.84	93.70	122.99
1,326	4.52	13.54	29.37	49.24	69.37	99.24	141.83
1,557	4.53	15.29	29.56	51.23	76.64	112.07	142.52
1,778	4.53	15.63	32.08	60.71	89.83	119.80	206.63
2,000	5.57	16.45	34.36	60.45	92.21	125.41	185.48

Figure 33a The raw runtime data of the newly implemented radius neighbor transformer.

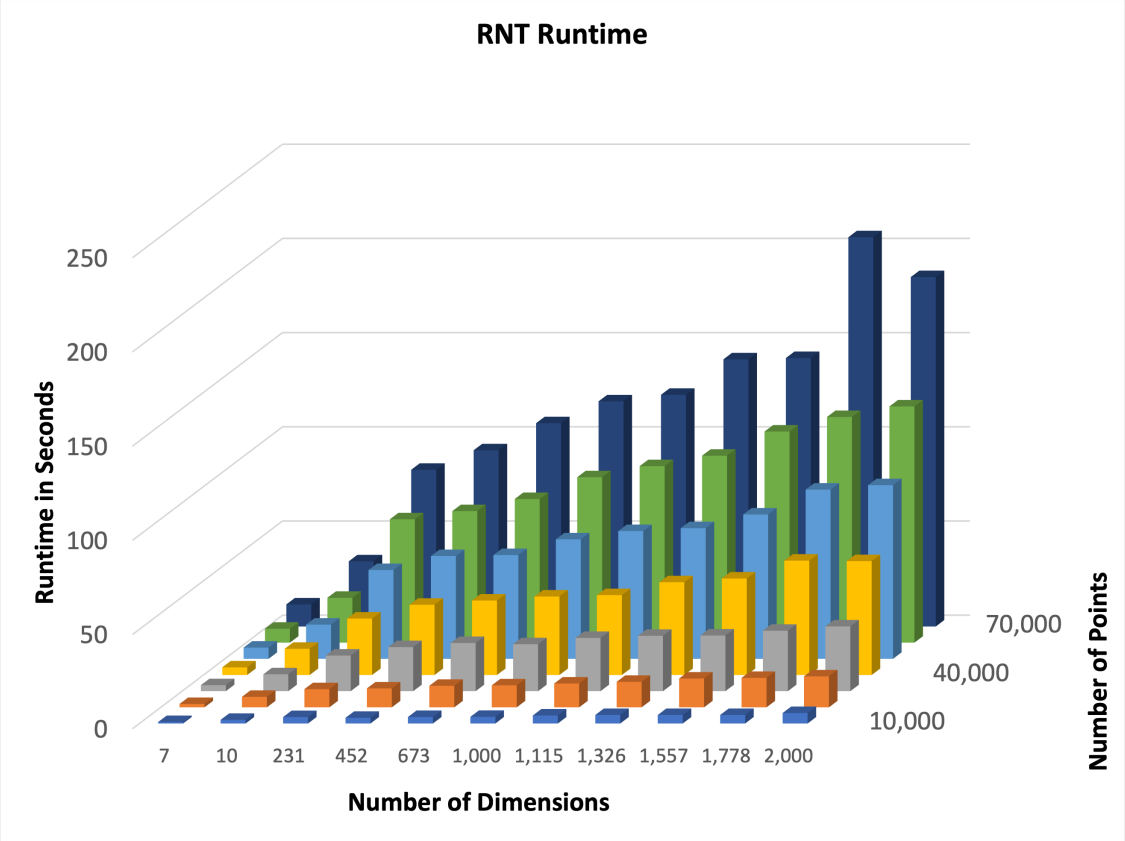


Figure 33b A 3D plot of the runtime based on number of dimensions and number of points.

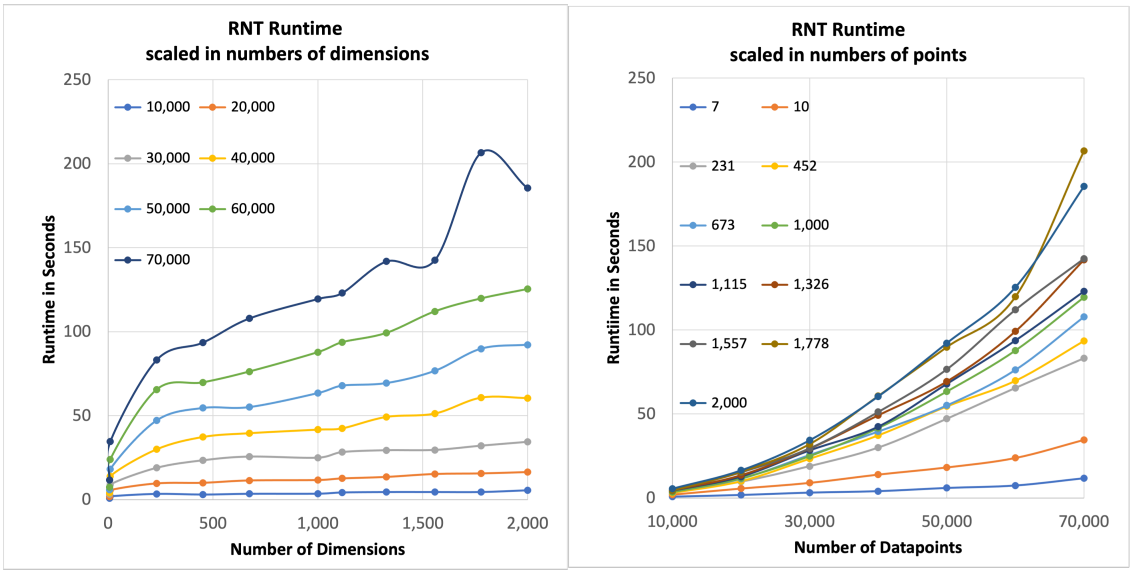


Figure 33c The left plot shows the ball-tree runtime in regards to an increasing number of dimensions. Different lines represent a different amount of points. The right plot shows the same data, but from the perspective of increasing the number of datapoints.

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	125%	120%	107%	109%	101%	98%	114%
10	142%	138%	115%	128%	119%	114%	131%
231	78%	59%	53%	46%	44%	43%	39%
452	59%	44%	40%	33%	30%	26%	23%
673	34%	37%	32%	26%	22%	17%	17%
1,000	35%	24%	21%	17%	14%	13%	12%
1,115	38%	24%	21%	15%	14%	12%	11%
1,326	34%	21%	17%	14%	12%	11%	10%
1,557	21%	19%	14%	12%	10%	10%	9%
1,778	17%	16%	13%	11%	11%	9%	11%
2,000	18%	13%	11%	10%	9%	8%	8%

Figure 33d The relative runtime improvement of the RNT in relation to the original scikit Learn ball-tree implementation. A green coloring of a cell indicates, that the RNT was faster, while red indicates the opposite.

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	-0.15	-0.30	-0.20	-0.35	-0.04	0.13	-1.40
10	-0.58	-1.55	-1.19	-3.08	-2.94	-2.85	-8.14
231	0.98	6.59	16.51	35.25	60.39	86.88	131.36
452	2.12	12.61	34.95	74.10	127.33	195.48	304.63
673	6.70	19.21	54.68	114.09	192.45	369.03	511.21
1,000	6.65	37.48	94.91	205.38	377.04	585.19	872.15
1,115	7.03	39.59	107.49	241.19	410.87	657.26	987.08
1,326	8.81	50.11	145.58	302.30	523.79	797.28	1213.06
1,557	17.18	64.12	187.50	388.45	663.40	1030.17	1482.26
1,778	21.66	79.58	211.62	486.01	763.29	1176.83	1694.69
2,000	25.19	106.78	265.52	555.45	921.97	1352.45	2077.09

Figure 33e The absolute time saved (in seconds) by the RNT relative to to the original scikit Learn ball-tree implementation. A green coloring of a cell indicates, that the RNT was faster, while red indicates the opposite.

MNist Benchmark Comparison

Finally, to show some numbers generated on a real world dataset, the baseline implementations as well as the RNT has been tested on the MNist dataset of hand written digits.

The popular dataset is often used for benchmarking machine learning algorithms. In this case, we only assess the runtime durations of the distance calculation backends. The data used comprised the 60,000 samples of the training dataset with a dimensionality of 784 (i.e. the 28×28 pixel graphics flattened into one vector). In this case, the cut_off radius was fixed to 2,000.

Table 3 shows the runtimes of the different distance calculation backends in the MNist case. At this point, you can see how RDist beats all previously implemented backends in form of the SciPy KD-tree and the scikit Learn ball-tree. Still, the newly implemented RNT also shows a good runtime.

Distance Backend	**Runtime in Seconds**
SciPy KD-Tree	1,551.20
Scikit Learn Ball-Tree	462.44
RDist	338.73
Scikit Learn RNT	99.26

Table 3 The runtime of different distance calculation backends in the MNist handwritten digits example. You can see, that RDist out-performs all previously implemented backends. But also the RNT shows competitive times.

6.5.2. PyNNDescent

Hypercube Benchmark Comparison

The graphics of figure 34 depict the runtimes of the PyNNDescent backend applying the hypercube example. All PyNNDescent parameters were chosen to prioritize accuracy over runtime with the n_neighbors set to 1,000. Note that a considerable amount of time is spent making converting the matrix to a CSR and making it symmetric (see figure 34d). These times were generated using parameters for PyNNDescent that value accuracy over speed. It is to be expected that tweaking these parameter might provide some more faster times. Generally, PyNNDescent was slower than the RNT for all cases, but faster than the scikit Learn ball-tree for cases that combined both, a high number of dimensions and many datapoints.

Figure 34c shows that the PyNNDescent framework delivers decent scaling, with near linear growth for both scaling in number of dimensions and number of points. Some fluctuations in the runtime graphs can be seen. Comparing these numbers to figure 34d, which shows the runtime, that is caused by conversions and making the distance matrix symmetrical, you can see how it contributes to the fluctuations.

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	40.22	67.18	102.57	138.35	176.56	216.85	264.15
10	33.11	72.67	107.75	143.67	181.47	220.06	267.84
231	40.74	97.44	151.60	199.47	268.79	318.96	358.29
452	39.79	96.63	173.29	226.18	289.51	322.64	451.32
673	43.57	100.88	174.44	228.10	274.61	344.68	416.62
1,000	42.38	101.45	169.21	231.35	306.21	363.80	438.06
1,115	46.59	109.65	182.76	229.83	308.47	375.77	444.27
1,326	46.45	115.83	184.23	259.69	310.26	390.01	505.70
1,557	50.15	122.41	185.39	259.16	332.84	429.48	494.19
1,778	49.90	131.29	209.23	300.12	389.95	458.68	584.20
2,000	57.58	129.62	214.17	311.48	400.42	486.37	534.80

Figure 34a The raw runtime data of the newly implemented PyNNDescent backend.

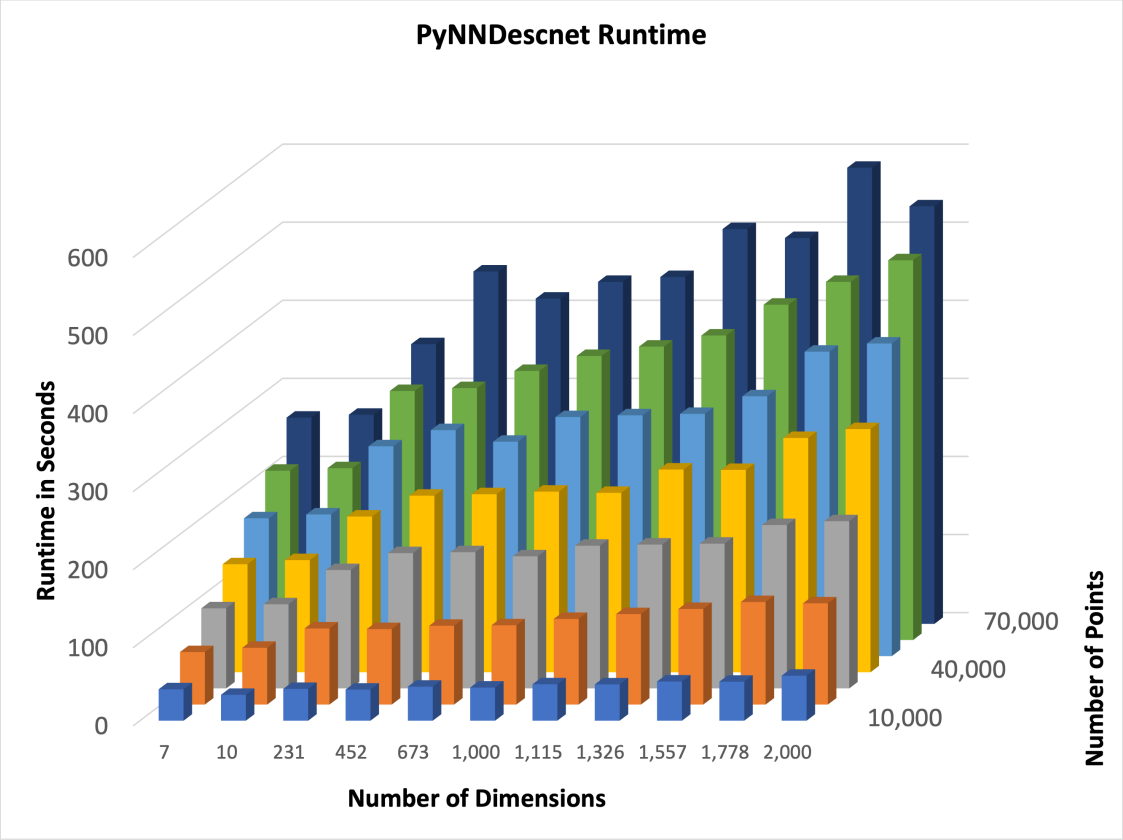


Figure 34b A 3D plot of the runtime produced by PyNNDescent, based on number of dimensions and number of points.

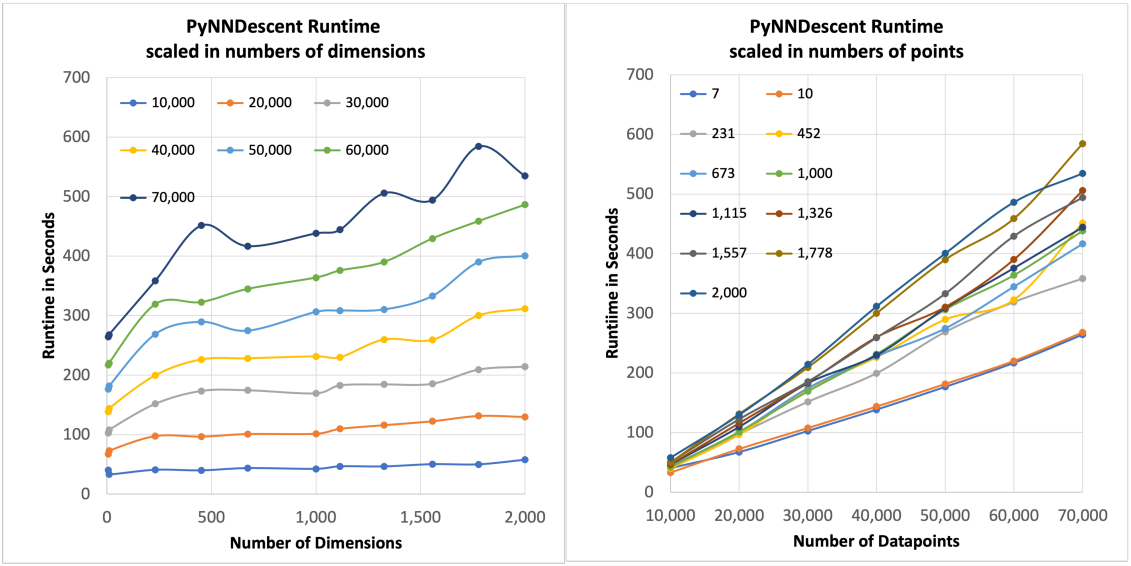


Figure 34c The left plot shows the PyNNDescent runtime in regards to an increasing number of dimensions. Different lines represent a different amount of points. The right plot shows the same data, but from the perspective of increasing the number of datapoints.

Dimensions \ Nr. of Points	10,000	20,000	30,000	40,000	50,000	60,000	70,000
7	1.45	2.60	5.59	6.99	11.67	15.00	29.09
10	3.44	8.03	9.60	12.35	15.64	20.08	35.49
231	10.28	24.14	33.26	44.39	71.37	82.41	82.91
452	6.96	20.27	54.93	71.39	90.71	83.44	138.84
673	9.60	21.65	54.65	68.94	70.06	90.36	130.09
1,000	8.22	19.35	41.20	59.46	84.22	94.63	124.28
1,115	10.36	23.72	49.94	50.30	75.42	98.11	115.49
1,326	8.37	24.95	42.01	69.17	60.52	86.35	151.55
1,557	10.19	25.17	33.47	52.68	66.00	105.25	112.16
1,778	7.47	23.75	45.73	80.85	104.77	109.43	175.10
2,000	10.73	20.21	43.05	73.96	92.46	112.19	96.63

Figure 34d This table shows the times when subtracting the PyNNDescent function call times from the overall pdist function time. Thus these values show the time taken by all of the transformations (creating the CSR matrix and making it symmetric).

Overall, the runtime is not competitive with the ones that the RNT generated. Still it has been shown, that a kNN approach can be implemented as a valid backend. Its parameterization using an amount of neighbors rather than a radius might be desirable in certain use cases.

Since all of the benchmarks above were conducted with PyNNDescent's n_neighbor parameter set to 1,000, one benchmark case (Hypercube: 1,000 dimensions, 30,000 datapoints) was representatively chosen and applied to a varying amount of n_neighbors, the result of which can be seen in figure 35. Note that the numbers of figure 35 were produced on the *laptop machine*, as opposed to the numbers shown previously, which stem from the *desktop machine*, hence they should be interpreted rather qualitatively than quantitatively. Also, the numbers plotted in figure 35 only include the PyNNDescent function calls, disregarding the transformations, that would be necessary afterwards. Despite some fluctuations, it seems like

increasing the number of neighbors in PyNNDescent has a (near) linear runtime impact.

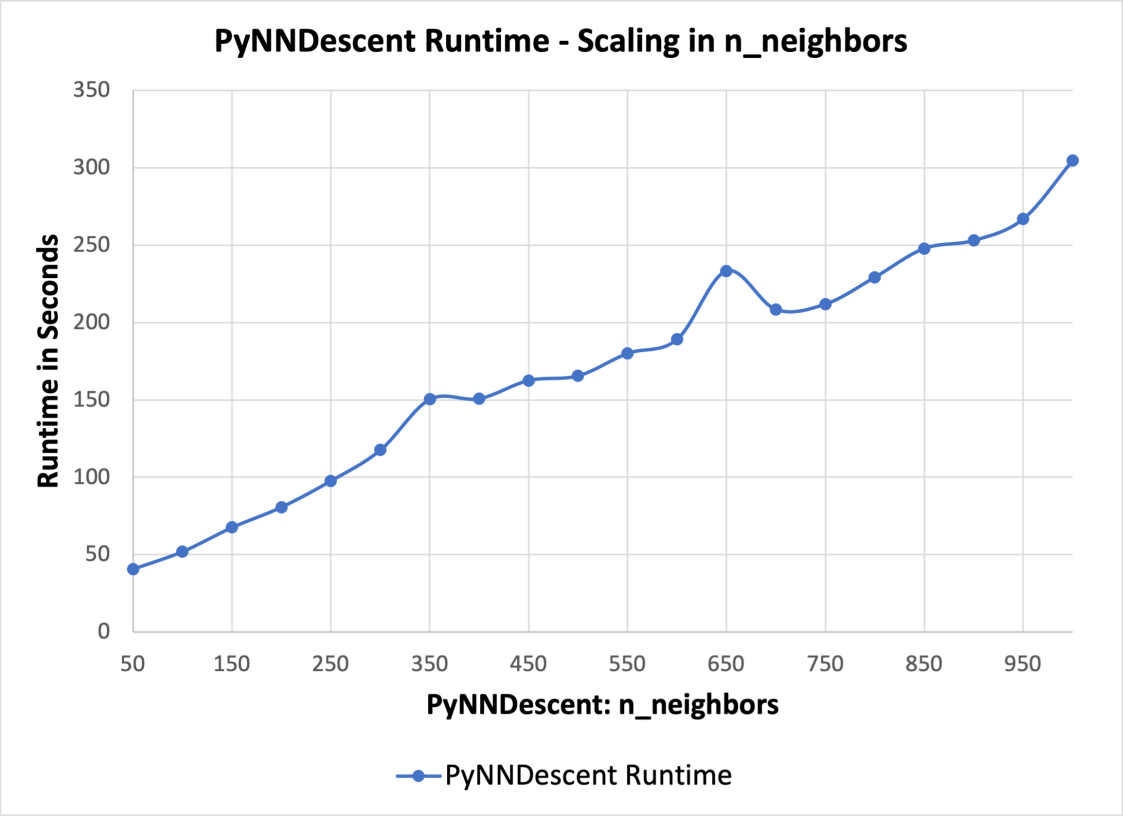


Figure 35 The runtime of the PyNNDescent when scaling the number of neighbors. Note that these numbers were generated on the laptop machine, as opposed to the other benchmarks shown in this chapter. Hence, this result needs to be interpreted qualitatively.

MNist Benchmark Comparison

Distance Backend	Runtime in Seconds
SciPy k-d Tree	1,551.20
Scikit Learn Ball-Tree	462.44
RDist	338.73
Scikit Learn RNT	99.26
PyNNDescent	375.22

Table 4 The runtime of different distance calculation backends in the MNist handwritten digits example. The PyNNDescent benchmark did produce a competitive runtime with regards to the k-d tree and the scikit Learn ball-tree. However both RDist and the RNT were faster.

Just as the other distance backends, the PyNNDescent implementation has been tested with the MNist Dataset. The results shown in table 4 show that it is not the fastest of the implementations, however does beat the Scipy k-d tree and the scikit Learn ball-tree. Further

performance might be possible with PyNNDescent when the restrictions for accuracy were lowered. However all other distance backends are non approximative, hence a high standard of numerical accuracy was requested, creating a level playing ground.

6.6. Impact of the k Parameter of optimize_parameters(...)

One parameter that was fixed for the benchmarks, is the k parameter in the optimize_parameters(...) function, which determines the index of the k^{th} closest neighbor which is chosen for the choice of the cut_off (amongst others). A more extensive description of the relevance of k in this function is explained in 5. It is obvious how a higher k would increase the cut_off parameter leading to a higher number of neighbors per row in the distance matrix.

Since different recommendations exist for the choice of k, it makes sense, to explore the implications of different k values. While it was left to its default (k = 25) for benchmarks shown in this thesis, some sources argue, that k should be chosen in a way, that $k = 2 \times d$ (where d is the dimensionality of the dataset). The intent behind this approach is to consider at least 2 points per dimension (i.e. to look right and left at least once for each dimension).
In order to gain an insight and visualise the connection between k and the cut_off (and some other implications), a few test runs have been conducted on the hypercube example. Values recorded were: numbers of non-zeros in the distance matrix, the runtime of the distance calculation and the cut_off value. The runs were using the RNT backend, with random seeds being fixed to 0 for the hypercube and optimize_parameters(...) respectively.

Higher Dimensions

One example was chosen with a higher dimensionality of 2,000 and 10,000 samples. K was sampled from 10 to 4,000 (reaching the $2 \times d$ recommendation).

k	10	25	50	125	250	500	1,000	2,000	4,000
Non Zeros	4,973,670	7,812,294	11,518,008	18,710,970	26,615,598	36,755,308	51,082,424	51,082,424	51,082,424
Runtime [s]	4.44	4.63	5.16	6.08	7.01	8.33	10.04	10.00	10.15
cut_off	14.4200	14.4670	14.5110	14.5750	14.6280	14.6870	14.7610	14.7610	14.7610
Average NNZ/Row	497	781	1,152	1,871	2,662	3,676	5,108	5,108	5,108
Sparsity	4.97%	7.81%	11.52%	18.71%	26.62%	36.76%	51.08%	51.08%	51.08%

Figure 36 The raw results from scaling k for a high dimensional dataset from k=10 to k=4,000. The default case is highlighted in blue while the k = 2 $\times$ d is coloured in green.

Both the plot shown in figure 37 and the raw data shown in figure 36 show, how after a k of 1,000, all values remain constant (the runtime within reason). Hence increasing the number of k above 1,000 did not change the values shown.

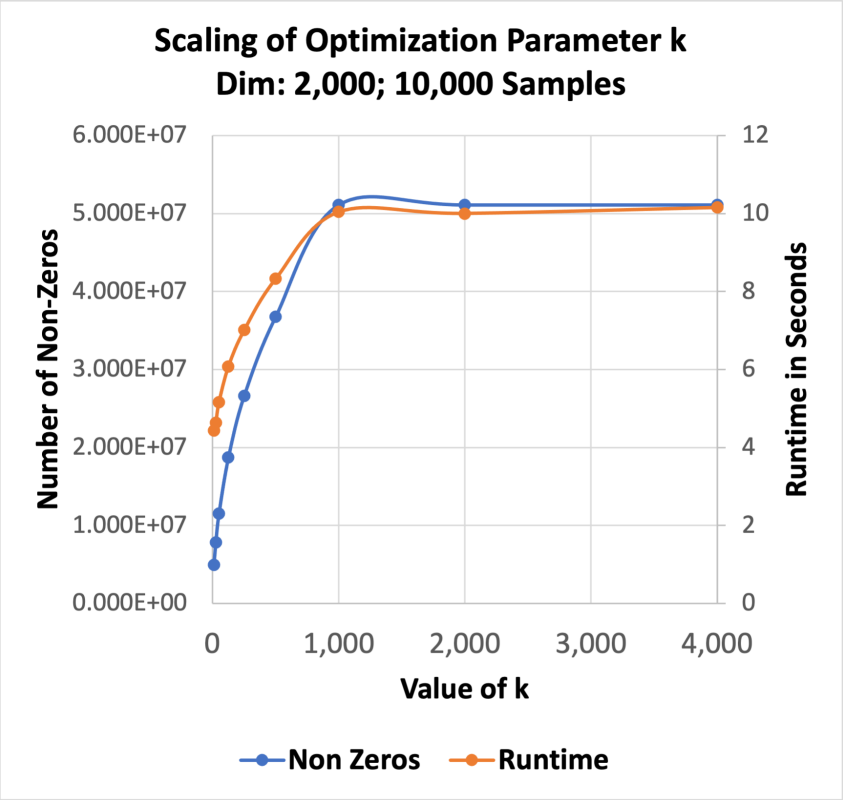


Figure 37 A plot of the amount of non-zero entries in the distance matrix and the runtime of the distance calculation over variable values of k.

Lower Dimensions

The same has been done with a lower dimensional example (d = 7). As a high number of dimensions causes all distances to approach a constant, a different behaviour is to be expected for a lower dimensional case.

k	2	4	8	14	25	28
Non Zeros	452,854	583,446	922,654	1,474,640	2,471,952	2,735,306
Runtime [s]	0.64	0.68	1.12	1.37	1.25	1.84
cut_off	0.265	0.276	0.297	0.32	0.349	0.355
Average NNZ/Row	6	8	13	21	35	39
Sparsity	0.11%	0.15%	0.23%	0.37%	0.62%	0.68%

56	112	225	500	1,000	2,000	4,000
5,667,034	12,067,332	24,492,326	51,722,750	92,192,914	92,192,914	92,192,914
2.71	4.17	5.48	9.91	16.56	17.17	16.21
0.402	0.46	0.525	0.609	0.691	0.691	0.691
81	172	350	739	1,317	1,317	1,317
1.42%	3.02%	6.12%	12.93%	23.05%	23.05%	23.05%

Figure 38 The raw results from scaling k for a low dimensional dataset from k=2 to k=4,000. The default case is highlighted in blue while the k = 2 $\times$ d is coloured in green.

As can be seen in figures 38 and 39, a qualitatively similar behaviour can be observed, how-

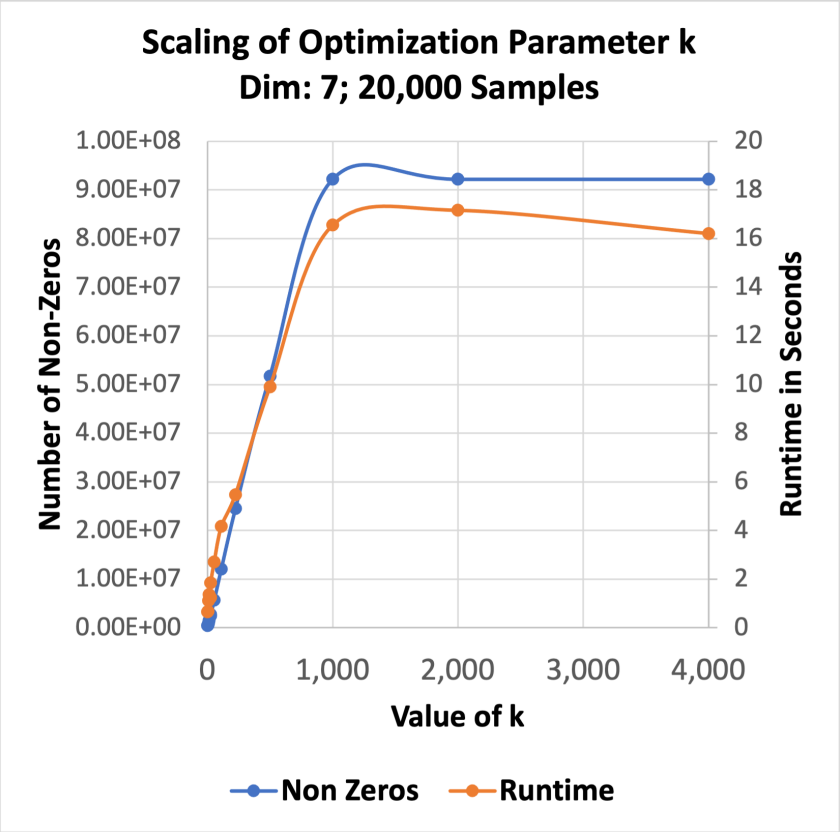


Figure 39 A plot of the amount of non-zero entries in the distance matrix and the runtime of the distance calculation over variable values of k.

ever with some slight deviations. Again, the numbers do not change after k reaches 1,000, but the cut_off values are noticeably different. While higher dimensional cut_off ranged from 14.2 to 14.7, it stays between 0.2 and 0.7 for the lower dimensional case. Interestingly, the same range of 0.5 can be seen between the two cut_off pairs.

Figure 40 compares the number of non zero entries in the distance matrix for both cases that were shown. Notice how the lines intersect before k = 256. Up to this point, the higher dimensional case produces more non-zero entries in the distance matrix. However a k value bigger than 256 causes the lower dimensional case to generate more non zero entries. While the two curves are qualitatively similar, you can see a difference in the number of non-zeros for a given k between the two cases.

You can also see how different the $2 \times$ d rule of thumb differs between the two cases. While it promotes a 50% sparsity in the distance matrix for the high dimensional case, the low dimensional case comes with a sparsity of 0.37%.
However, it has to be said, that the same principle holds true for the fixed default of 25.

Finally, the results once again state, that the implications for the choice of k again greatly

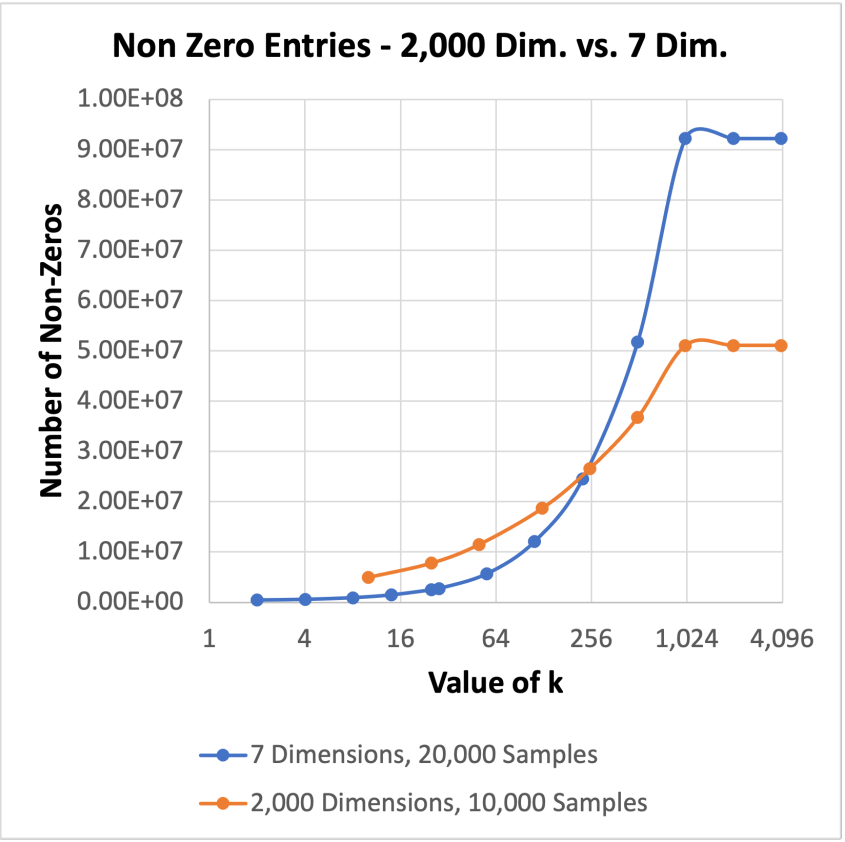


Figure 40 A logarithmic comparison between the two cases (7 dimensions, 10,000 samples and 2,000 dimensions, 20,000 samples). You can see how the orange curve intersects the blue one just before k = 256.

depend on the structure of the dataset itself. Hence, different densities and distribution of the data are expected to have an impact on how the k value affects the result for a certain dataset.

7. Project Recap

As mentioned previously, this project had a rather agile approach, with no specific predefined outcome. In order to do it justice, an overview will be given, which chronologically describes challenges, opportunities, findings, lessons learned, etc. which were found along the way. On one hand, this gives an insight for researchers working on a similar topic or approach, while on the other hand makes it clearer why some paths were preferred over others.

Starting with the IDP (as mentioned in 3) that occurred before the thesis ([13]), a SLEPc eigensolver was implemented into datafold, which provided good results while granting a great scalability for cluster deployments. This IDP had a different starting point than this project, as it was clearly predefined which eigensolver was supposed to be used, knowing it was a performant backend. The benchmark case that was utilized in the IDP was mainly the hypercube example, which found its way over to this thesis. This example allowed us to easily scale its number datapoints, which was the relevant scaling factor, as the eigensolver is applied after the distance calculations, therefor it receives a square matrix with the number of rows and columns equalling the number of datapoints. During the benchmarking on a cluster, we noticed that certain parts of the code were causing a high increase in the overall runtime. These functions were suspected to be optimize_parameters(...) and the distance calculations. Note that we were benchmarking scaling only in the number of datapoints, up to 3.2 million (in 7 dimensions). It would later turn out, that this was a rather special use case, which would not generally be a good representative.

Following this IDP, the datafold developers wanted to further optimize the capabilities of the framework, in form of the code that is executed before the eigensolver. At this point this thesis was started, based on the observations made before. The goal at the beginning was to find efficient solutions for bottlenecks for use on both cluster and local machines.

As is necessary for any work on an existing framework, at first, the existing code, which was to become relevant, was studied. Having worked on the eigensolver before, we would not come across the codebase prior to the eigensolver part very much. As the hypercube example was still the go-to, the first baseline benchmarking and profiling runs were conducted on it, confirming the previous observations, that the main focuses were going to lie on the optimize_parameters(...) function and the distance calculations.
Scaling up only the numbers of sample points, the performance of the distance calculation-runtime scaled decently, but the runtime of optimize_parameters(...) would quickly account for more than 50% of the overall runtime. Hence it became the first function for which a more efficient solution was searched (see 4.1). The following literature and framework research

to this problem finally pointed towards the DASK framework which yielded promising results, which were later confirmed by the final benchmarks (see 5).
This solution fulfilled all of the goals that were aimed at, improving the runtime significantly for a wide variety of deployment cases. Overall, optimizing this part of the code required no real compromise or re-orientation in any form.

For further orientation regarding the distance calculation, a meeting was organized (see [18]) with researchers who work extensively with and on the datafold framework to gain some more insight. Up to this point, the main benchmark case was the hypercube in 7 dimensions, having performed baseline benchmarks with the existing distance backends and identifying the k-d tree as the fastest for this specific use case. Subsequently the literature and framework research up to this point was based on such a use case. At the mentioned meeting, it became clear, that the restriction to 7 dimensions was too shallow, as we were told, that real world use cases can have thousands of dimensions. Following a target of 2,000 dimensions, which was deemed a decent value, the benchmark suite, explained in 5.2.1 was set up. It was also mentioned, that the existing distance calculation backend RDist performs well on datasets, which fulfill the manifold assumption well (i.e. the s-curve, Swiss roll, etc.). Other cases however, like the hyper-cube can lead to an increased memory usage and can harm the overall performance. It was therefor decided to stick to the hypercube example for two reasons:

A) As the hypercube was used in many benchmarks, spanning multiple projects, a considerable collection of data was generated on its base. Sticking to this example would preserve comparability and continuity.

B) It is less promising to find an out of the box Python framework, which would outperform the C++ based and optimized RDist

This again opened the literature and framework research, now with a different use case in mind. It is fair to say, that it is important to oversee the whole scope of possible and likely use cases, as this can have major implications for the following and the work that has already been done.
Despite the extensive and time consuming search for alternative Python distance calculation frameworks, none were found that represent this exact problem at hand, that were not already used as an implemented backend. According to various postings on developer forums and framework developers themselves, the conclusion was found, that Python available distance calculations are represented best in form of the already implemented backends. At this point, it was decided to drop the goal of a cluster centered approach in favor of attempting to improve runtime on local machines. Hence, it lead to searching the current distance backends (mainly: scikit Learn) for improvements, one of which was found in form of the Radius Neighbor Transformer, which was then implemented and benchmarked. But the fact, that many solutions exist for approximate NN, testing such an approach was discussed to be worth to

try, which was finally done using PyNNDescent. To name few of the frameworks which have been taken a look at, but were rather quickly disregarded due to poor initial results, dask-distance and fastdist should be named. These unfortunately do not offer cut_off parameters and initially did not provide promising results, calculating all distances.

Further concepts have been discussed, such as using a local sensitive hashing framework to get index tuples of the points that fall within a certain proximity to each other and querying their distances, which were concluded to be rather non competitive as correspondence with the developers of the ANN benchmarks ([1]) brought up (see [4]). For more infos, see 8.2.

Another example is the deployment of the existing backends on multiple nodes using a MPI4PY program, which has been tested on local machines, but was disregarded when the decision was made to not focus on cluster deployments.
Some of these approaches could yield good results, however could not be further pursued anymore in this thesis. These will however be outlined in the following chapter 8.

As you can see, this project involved a non negligible amount of ambiguity and necessity to research, reorient and rethink approaches and trials. Thus, what is shown in this thesis are only those parts that were reckoned to have a sustainable and relevant impact on the project and are worth to be documented.

8. Outlook and Future Work

During the project it became apparent, that finding an implementation especially for cluster deployment and distributed memory application is rather difficult, as the fixed radius neighbor search is still an open problem and therefor still researched. Therefor we shifted our attention towards off cluster use cases. However to give a perspective, on how this problem might be somewhat mitigated, a small example should be given on how the given backends might be used on multi node systems.

8.1. Cluster deployment

In order to port existing solutions like the ones based on SciPy and scikit Learn, which are optimized for shared memory, local applications, one can use a simple MPI4PY program to distribute the work over multiple processes.
Since the distance calculations do not dependent on one another, you can have them happen in parallel on multiple MPI processes. This approach furthermore allows for a black-box approach, allowing easy swapping between the existing distance backends.

Figure 41 shows a section of this example program. The first three lines collect some general information about the MPI environment, like the communicator, rank of the instance and the overall number of processors in form of size. Assume the master rank 0 is given the dataset X and the cut_off parameter (as is implied in lines 8-9). Rank 0 also generates the indices tuples (start, end) which determine what rank is responsible for a specific range of indices of the dataset (here called: segments). I.e. a data set with 3,000 points is distributed over 3 processes with rank 0 calculating distances for points 0 to 999, rank 1: 1,000-1,999, rank 2: 2,000-2,999.
Since the whole dataset is required to calculate the correct distances for a respective segment, rank 0 firstly broadcasts X to all other processes, followed by the cut_off. Finally, the indices are scattered around the processes in line 21.
Line 24 starts the distance calculations for each rank. Note that in this case, as we do not want each rank to calculate all distances, we need to use a cdist(...) function. This way, we get the distances between the whole dataset X and the segment of X which the respective rank is assigned to.
Once finished, the calculated distance matrix is sent back to rank 0 using the gather command in line 26. The same rank is finally responsible to stack all the partial solutions to the final distance matrix and convert it into CSR format.

As this can use any cdist(...) method of the distance backends, which are optimized for shared

memory, but oftentimes with multiple cores in mind, you would probably want to deploy it in form of a hybrid MPI program, granting multiple threads to one MPI process. This way, you also help the fact, that the distance calculation can be memory intensive. Hence running i.e 64 MPI processes on one node could lead to an excessive memory usage. Running 16 MPI processes with 4 threads each would require 1/4 of the data structures to be built.
Note that this concept has only been tested in small scale on a local machine and yet needs to be assessed on a cluster.

```
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

if rank == 0:
    #Master
    #Assume rank 0 loads the dataset X and the cut_off
    X = get_data(...)
    cut_off = gat_cut_off(...)
  #Distribute the length of the dataset over the number of processors
    segment = get_segments(size, len(X))
else:
    #Worker
    #Prepare workers to receive data
    X = None
    segment = None
    cut_off = None
#Send data to ranks
X = comm.bcast(X, root=0)
cut_off = comm.bcast(cut_off, root=0)
segment = comm.scatter(segment, root=0)

#compute distance using an arbitrary distance backend.
distance = compute_distance(cut_off,X, X[segment[0]:segment[1]])
#collect solutions
distance = comm.gather(distance, root=0)

if rank == 0:
  #build final solution
    combined = scipy.sparse.vstack(distance).tocsr()
```

Figure 41 A section of the example MPI program with which the existing distance backends can be deployed over multiple nodes.

8.2. Local Sensitive Hashing

At the end of the project, a framework came into view, implementing so called *Local Sensitive Hashing*. This framework, *FALCONN* (FAst Lookups of Cosine and Other Nearest Neighbors), enables range based nearest neighbor searches for high dimensions. The implementation and assessment of this approach unfortunately was not possible in the remaining time of the project, nevertheless, the idea has been noted and put aside for future work.

The idea would be to pass the dataset to the FALCONN framework, which would then dis-

tribute the datapoints into a predefined number of buckets, considering local proximity as their similarity. Querying the indices of points that fall within a certain radius of one point would allow sub-linear query times (see [1]), leaving the distances to be calculated to the identified neighbors.

Personal correspondence with the ANN benchmarks developers, stating our use case, however uncovered, that Erik Bernhardsson (see [4]) doubted FALCONN's ability to deliver these results performantly: "[...] generally LSH is not very competitive" ([4]). This statement has discouraged further pursuit of this approach.

8.3. Further ANN Benchmark Candidates

Since successfully adding a kNN approach opened a new door for possible candidates, future work is intended, benchmarking and comparing further frameworks from the ANN benchmarks (see [24]).

9. Summary and Conclusion

optimize_parameters(...)

Optimizing the optimize_parameters(...) function yielded good results, with up to 5 times faster runtimes and vastly extending the number of points that can be handled. It has been found, that the implementation of the numpy.partition(...) method is a good implementation with great runtime classes for the problem. Replacing this method with one of similar function provided by DASK has therefor proven to be efficient, allowing almost twice as many datapoints in half of the time. Unfortunately current compatibility constraints of DASK do not allow support for SciPy's CSR matrices at this point in time. It has shown, that the number of datapoints is the main scaling factor for this function.
Overall the DASK framework appears to be a promising software library, which could provide datafold with further improvements over the future, especially once SciPy adds support that enables compatibility for scipy.sparse.csr matrices. Enabling efficient scaling for both cluster and local machines, fulfills the requirements perfectly. Being made for enabling less performant machines to process large dataset fits the use cases of datafold well.

Distance Calculations

Finding further Python frameworks that enable a more efficient distance calculation has proven to be difficult and finally brought us back to scikit Learn. Still, concentrating mostly on the hypercube benchmark, a use case which does not benefit the RDist backend, considerable improvements have been made by altering the implementation of the backends already installed, with some cases running around 10 times faster than the previously fastest: SklearnBalltreeDist backend. Especially letting the RNT from scikit Learn framework choose which algorithm to run held the key to faster distance calculation times, otherwise it resembles the SklearnBalltreeDist.
A whole new approach has been tested in form of the PyNNDescent approximate kNN framework. It has shown good runtimes in comparison to the original k-d tree and scikit Learn ball-tree, but was not as fast as the RNT.
Generally, the tested distance algorithms scaled worse in the number of samples than their dimensionality.
For the chosen benchmark scenario, the hypercube, the newly implemented Radius Neighbor Transformer is the backend of choice with the fastest runtimes.

Project Retrospective

As is usual for scrum projects, the final part is the project retrospective, in which I would like to put the overall approach of this project into perspective.
The nature of the project suited and served the original idea of exploring what is possible

and available, assessing the current status quo. Most of all, it provided a widespread insights into multiple fields of datafold itself, but also into the areas relevant for optimization, such as: selection algorithms, partitioning, fixed radius neighbor searches, approximate k nearest neighbors and multi dimensional data structures.

List of Acronyms

MPI Message Parsing Interface

IDP Interdisciplinary Project

VM Virtual Machine

HPC High Performance Computing

COO COOrdinate Form

CSR Compressed Sparse Row

RNT Radius Neighbors Transformer

aNN Approximate Nearest Neighbor(s)

kNN k Nearest Neighbor(s)

NN Nearest Neighbor(s)

LSH Locality-Sensitive Hashing

OS Operating System

Bibliography

[1] M. Aumüller, E. Bernhardsson, and A. J. Faithfull. ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms. *CoRR*, abs/1807.05614, 2018

[2] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Commun. ACM*, 18(9):509–517, Sep. 1975. https://doi.org/10.1145/361002.361007

[3] J. L. Bentley. *A Survey of Techniques for Fixed Radius near Neighbor Searching.* Technical report, Stanford, CA, USA, 1975. https://www.osti.gov/biblio/7368350

[4] E. Bernhardsson. Personal Correspondence (E-Mail via. Tobias Neckel), 2022. E-Mail correspondence

[5] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7(4):448–461, 1973. https://www.sciencedirect.com/science/article/pii/S0022000073800339

[6] Dask Development Team. SKLearn Source Code: scikit-learn/sklearn/neighbors/. date accessed: 2021-01-20, 2014-2018. https://docs.dask.org/en/latest/_modules/dask/array/reductions.html#topk

[7] Dask Development Team. *Dask: Library for dynamic task scheduling*, 2016. https://dask.org

[8] datafold contributors. Datafold Website. date accessed: 2022-01-12, 2022. https://datafold-dev.gitlab.io/datafold/index.html

[9] J. T. de Balsch. Searching for Fixed-Radius Near Neighbors with Cell Lists Algorithm in Julia Language. date accessed: 2021-01-27, 2021. https://jaantollander.com/post/searching-for-fixed-radius-near-neighbors-with-cell-lists-algorithm-in-julia-language/#introduction

[10] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.*, 3(3):209–226, sep 1977. https://doi.org/10.1145/355744.355745

[11] GeeksforGeeks. Quickselect Algorithm. date accessed: 2022-02-02, 2021. https://www.geeksforgeeks.org/quickselect-algorithm/

[12] GeeksforGeeks. Quickselect Algorithm. date accessed: 2022-02-09, 2022. https://www.geeksforgeeks.org/hoares-vs-lomuto-partition-scheme-quicksort/

[13] M. Grad and T. Raith. *Efficient Numerical Solvers for the Manifold Learning Framework datafold*. Technical report, Technical University of München, Garching, Munich, March 2021

[14] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. page 47–57, 1984. https://doi.org/10.1145/602259.602266

[15] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, jul 1961

[16] M. Hucker. Tree algorithms explained: Ball Tree Algorithm vs. KD Tree vs. Brute Force. date accessed: 2021-01-26, 2020. https://towardsdatascience.com/tree-algorithms-explained-ball-tree-algorithm-vs-kd-tree-vs-brute-force-9746debcd940

[17] R. R. U. S. Kevin Beyer, Jonathan Goldstein. When Is "Nearest Neighbor" Meaningful? In C. Beeri and P. Buneman (editors), *Database Theory — ICDT'99*, pages 217–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 1999. https://link.springer.com/chapter/10.1007%2F3-540-49257-7_15

[18] D. Lehmberg. Personal Correspondence (Meeting and E-Mail), 2021, 2022. Meeting held on: 2021-11-16

[19] D. Lehmberg, F. Dietrich, G. Köster, and H.-J. Bungartz. datafold: data-driven models for point clouds and time series on manifolds. *Journal of Open Source Software*, 5(51):2283, 2020

[20] L. McInnes. PyNNDescent. date accessed: 2021-01-23, 2020. https://pynndescent.readthedocs.io/en/latest/index.html

[21] D. R. Musser. Introspective sorting and selection algorithms. *Software: Practice and Experience*, 27(8):983–993, 1997

[22] S. M. Omohundro. *Five Balltree Construction Algorithms*. Technical report, International Computer Science Institute California, 1989. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.91.8209

[23] H. Samet and J. Gray. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann series in computer graphics and geometric modeling. Elsevier Science, 2006. https://books.google.de/books?id=v0-NRRKHG84C

[24] The ANN Developers. ANN Benchmark Git. date accessed: 2021-01-27, 2021. https://github.com/erikbern/ann-benchmarks

[25] The NumPy Developers. SKLearn Source Code: scikit-learn/sklearn/neighbors/. date accessed: 2021-01-20, 2008-2022. https://numpy.org/doc/stable/reference/generated/numpy.partition.html

[26] The PyNNDescent Developers. PyNNDescent Git. date accessed: 2021-01-27, 2021. https://github.com/lmcinnes/pynndescent

[27] The Scikit-learn developers. SKLearn Documentation: sklearn.neighbors.NearestNeighbors. date accessed: 2021-01-19, 2007-2021. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.NearestNeighbors.html#sklearn.neighbors.NearestNeighbors.radius_neighbors_graph

[28] The Scikit-learn developers. SKLearn Documentation: sklearn.neighbors.RadiusNeighborsTransformer. date accessed: 2021-01-30, 2007-2021. https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.RadiusNeighborsTransformer.html

[29] The Scikit-learn developers. SKLearn Source Code: scikit-learn/sklearn/neighbors/. date accessed: 2021-01-19, 2007-2021. https://github.com/scikit-learn/scikit-learn/tree/7e1e6d09bcc2eaeba98f7e737aac2ac782f0e5f1/sklearn/neighbors

[30] The SciPy community. SciPy Documentation: scipy.spatial.ckdtree.pyx. date accessed: 2021-01-19, 2008-2021. https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.cKDTree.html#scipy.spatial.cKDTree

[31] The SciPy community. SciPy Roadmap. date accessed: 2021-01-31, 2008-2021. https://docs.scipy.org/doc/scipy/dev/roadmap.html

[32] The SciPy community. SciPy Source Code: scipy.spatial.ckdtree.pyx. date accessed: 2021-01-19, 2008-2021. https://github.com/scipy/scipy/blob/maintenance/1.7.x/scipy/spatial/ckdtree.pyx

[33] A. A. Tokuç. k-Nearest Neighbors and High Dimensional Data. date accessed: 2021-01-27, 2021. https://www.baeldung.com/cs/k-nearest-neighbors

[34] Wikipedia, The Free Encyclopedia. K-D Tree. date accessed: 2022-02-06, 2022. https://en.wikipedia.org/wiki/K-d_tree