pynndescent

Release 0.5.0

Apr 20, 2023

User Guide / Tutorial:

PyNNDescent is a Python nearest neighbor descent for approximate nearest neighbors. It provides a python implementation of Nearest Neighbor Descent for k-neighbor-graph construction and approximate nearest neighbor search, as per the paper:

Dong, Wei, Charikar Moses, and Kai Li. *"Efficient k-nearest neighbor graph construction for generic similarity measures."* Proceedings of the 20th international conference on World wide web. ACM, 2011.

This library supplements that approach with the use of random projection trees for initialisation. This can be particularly useful for the metrics that are amenable to such approaches (euclidean, minkowski, angular, cosine, etc.). Graph diversification is also performed, pruning the longest edges of any triangles in the graph.

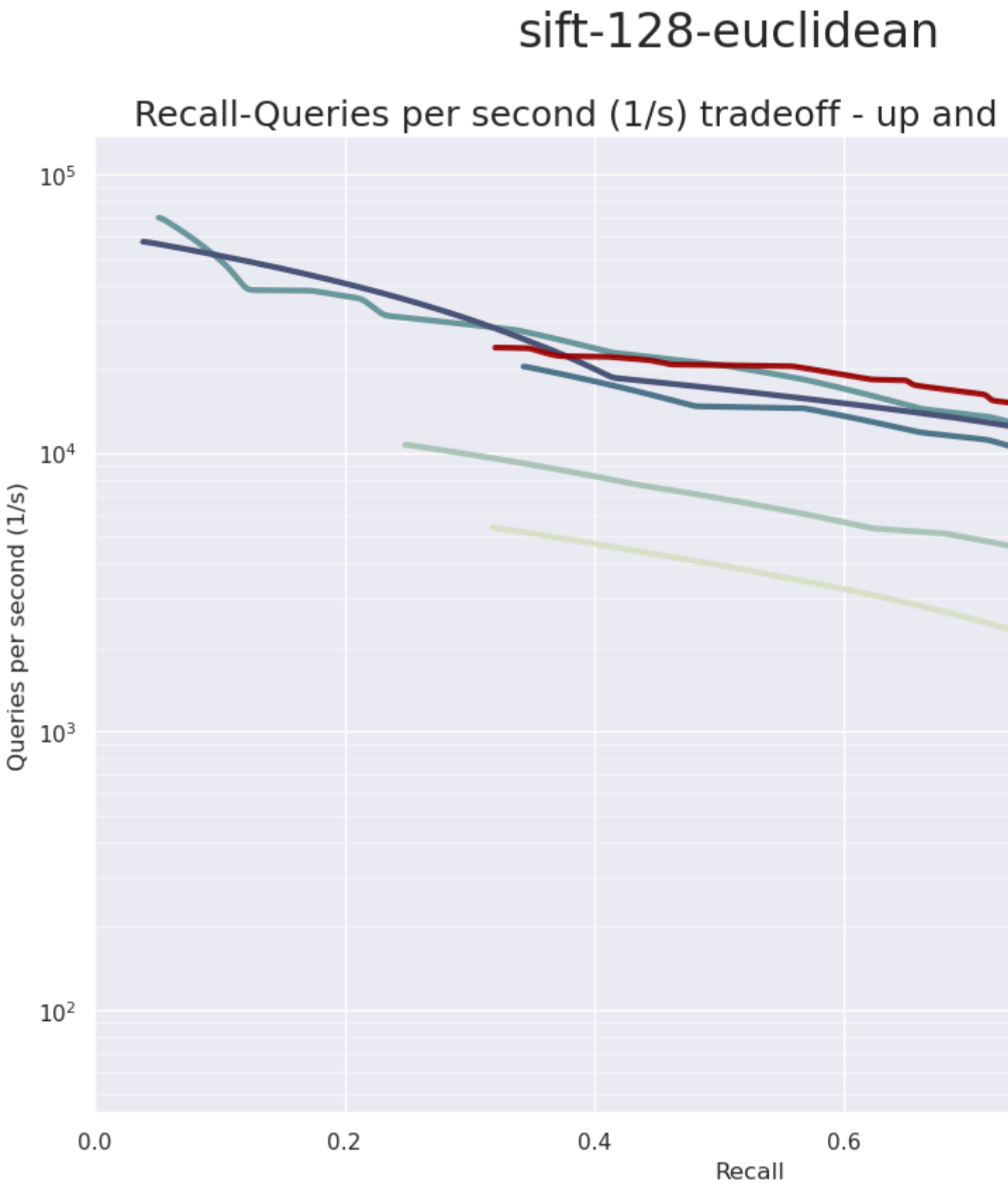
Currently this library targets relatively high accuracy (80%-100% accuracy rate) approximate nearest neighbor searches.

CHAPTER 1

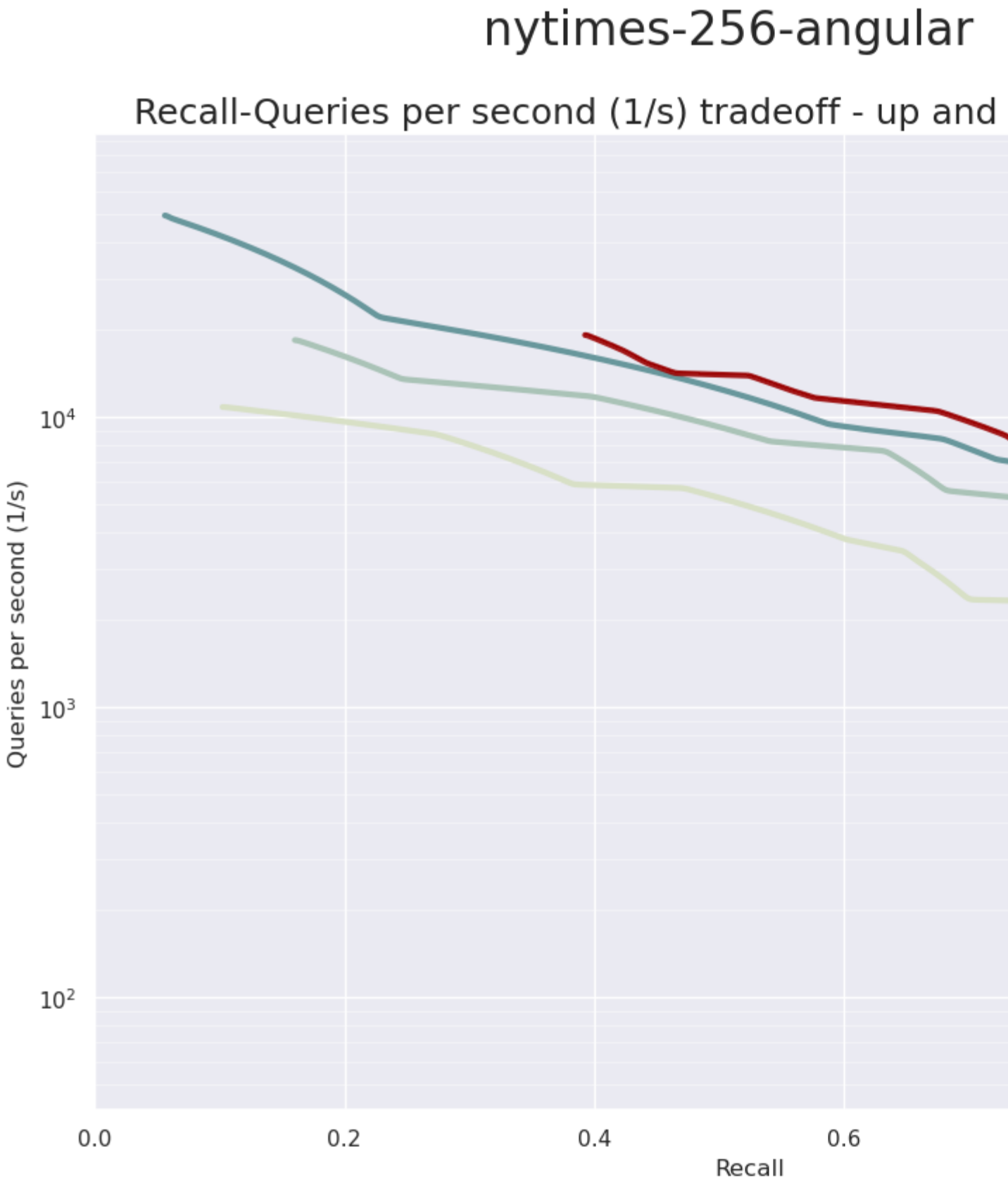
Why use PyNNDescent?

PyNNDescent provides fast approximate nearest neighbor queries. The ann-benchmarks system puts it solidly in the mix of top performing ANN libraries:

SIFT-128 Euclidean


sift-128-euclidean
Recall-Queries per second (1/s) tradeoff - up and
Queries per second (1/s)
10^5
10^4
10^3
10^2
0.0
0.2
0.4
0.6
Recall

NYTimes-256 Angular


nytimes-256-angular
Recall-Queries per second (1/s) tradeoff - up and
Queries per second (1/s)
10^4
10^3
10^2
0.0
0.2
0.4
0.6
Recall

While PyNNDescent is among fastest ANN library, it is also both easy to install (pip and conda installable) with no platform or compilation issues, and is very flexible, supporting a wide variety of distance metrics by default:

Minkowski style metrics

- euclidean
- manhattan
- chebyshev
- minkowski

Miscellaneous spatial metrics

- canberra
- braycurtis
- haversine

Normalized spatial metrics

- mahalanobis
- wminkowski
- seuclidean

Angular and correlation metrics

- cosine
- dot
- correlation
- spearmanr
- tsss
- true_angular

Probability metrics

- hellinger
- wasserstein

Metrics for binary data

- hamming
- jaccard
- dice
- russelrao
- kulsinski
- rogerstanimoto
- sokalmichener
- sokalsneath
- yule

and also custom user defined distance metrics while still retaining performance.

PyNNDescent also integrates well with Scikit-learn, including providing support for the KNeighborTransformer as a drop in replacement for algorithms that make use of nearest neighbor computations.

CHAPTER 2

Installing

PyNNDescent is designed to be easy to install being a pure python module with relatively light requirements:

- numpy
- scipy
- scikit-learn >= 0.22
- numba >= 0.51

all of which should be pip or conda installable. The easiest way to install should be via conda:

```
conda install -c conda-forge pynndescent
```

or via pip:

```
pip install pynndescent
```

2.1 How to use PyNNDescent

PyNNDescent is a library that provides fast approximate nearest neighbor search. It is designed to be as flexible as possible for python users. That includes a wealth of pre-defined distance measures, the ability to use custom user-defined distance measures, as well as handling of sparse matrix inputs and more. Let's walk through how you can use PyNNDescent for approximate nearest neighbor search. First let's load the library, and some tools to get some suitable data.

[1]:
```
import pynndescent
import numpy as np
import h5py
from urllib.request import urlretrieve
import os
```

The ann-benchmarks website, maintained by Erik Bernhardsson, Martin Aumueller and Alex Faitfull, provides a comprehensive suite of benchmarking for approximate nearest neighbor libraries and algorithms. For our purposes the

important fact is that this includes keeping a variety of datasets, of varying levels of size and difficulty, for trying out nearest neighbor search on. To make things easy we'll write a short function that can fetch the (pre-prepared) dataset from ann-benchmarks and load it into train and test numpy arrays.

[2]:
```
def get_ann_benchmark_data(dataset_name):
    if not os.path.exists(f"{dataset_name}.hdf5"):
        print(f"Dataset {dataset_name} is not cached; downloading now ...")
        urlretrieve(f"http://ann-benchmarks.com/{dataset_name}.hdf5", f"{dataset_name}
↪.hdf5")
    hdf5_file = h5py.File(f"{dataset_name}.hdf5", "r")
    return np.array(hdf5_file['train']), np.array(hdf5_file['test']), hdf5_file.attrs[
↪'distance']
```

To start let's grab the fashion-mnist dataset for some initial experiments.

[3]:
```
fmnist_train, fmnist_test, _ = get_ann_benchmark_data('fashion-mnist-784-euclidean')
```

With the dataset now loaded let's build a search index for the training set. This is done using the `NNDescent` class, which we simply hand the training data to (we'll look at other parameter options later). It will take a little time to build the index, so let's keep track using the `%time` magic in jupyter.

[4]:
```
%%time
index = pynndescent.NNDescent(fmnist_train)
```

```
CPU times: user 48.5 s, sys: 1.07 s, total: 49.5 s
Wall time: 27.7 s
```

Over ten seconds! That seems slow. How much data did we index, and how high dimensional is it?

[5]:
```
fmnist_train.shape
```

[5]:
```
(60000, 784)
```

Okay, so sixty-thousand samples living in a seven-hundred and eighty-four dimensional space – that is a reasonable amount of data. We already had the data however, so why did we spend all that time building an index on top of it? It allows us to query that data to find the points closest to new, previously unseen data points. This is a surprisingly common problem – "find the other things that look like this" – that crops up in everything from recommendation systems (e.g. what are some other songs or artists that are like this song?), to classification and regression (using a KNN-classifier and friends), to clustering (and density based clustering algorithm makes heavy use of near neighbor searches to determine density).

So, given that we have built an index on the training data, we can use that index to find the nearest neighbors from the training set to each sample in the test set. Let's do that now for the first 10 samples of the test set, again, keeping track of the time:

[6]:
```
%%time
neighbors = index.query(fmnist_test[:10])
```

```
CPU times: user 23.4 s, sys: 377 ms, total: 23.8 s
Wall time: 18.3 s
```

That was also slow! What happened? The first time the index is queried it does some book-keeping (that may not be required for some other tasks as we'll see below). There is also time for numba to JIT compile many routines in the background (you might find, for instance, that the very first run of index building might take longer than you expect, but subsequence runs are much faster). You can force this ahead of time by calling the `prepare` method. Let's rebuild the index from scratch including the `prepare` step to see how long that takes.

[7]:
```
%%time
index = pynndescent.NNDescent(fmnist_train)
index.prepare()
```

```
CPU times: user 47.3 s, sys: 743 ms, total: 48.1 s
Wall time: 20 s
```

That took longer, but how long do the queries take now? Let's query the entire test set.

[8]:
```
%%time
neighbors = index.query(fmnist_test)
```

```
CPU times: user 742 ms, sys: 17.6 ms, total: 759 ms
Wall time: 758 ms
```

That's more like the sort of performance we might have been hoping for. And just to demonstrate that the prepare step is getting called on the first query, we'll start from scratch again.

[9]:
```
%%time
index = pynndescent.NNDescent(fmnist_train)
neighbors = index.query(fmnist_test[:10])
```

```
CPU times: user 47.1 s, sys: 673 ms, total: 47.8 s
Wall time: 19.4 s
```

[10]:
```
%%time
neighbors = index.query(fmnist_test)
```

```
CPU times: user 785 ms, sys: 21.3 ms, total: 806 ms
Wall time: 821 ms
```

Again, we get the sort of performance we might hope for by constructing a fast index over the data (and the numba JIT compiler is stating to warm up a little). There is a catch however – at the start we said **approximate** nearest neighbor search. We might have gotten some neighbors data returned, but how good is it? Being fast doesn't matter if you return random results. To check we'll have to find the true nearest neighbors of the test set. Given that we have to search through sixty-thousand points for each of the ten-thousand test samples a pure brute force approach is going to be too expensive. Instead we can use kd-trees from scikit-learn.

[11]:
```
from sklearn.neighbors import KDTree
```

The `KDTree` is scikit-learn has a very similar interface. You hand the tree constructor the training data, and then you can query it. In this case there is no prepare step required – all the work is done at construction time since there aren't intermediate results that might be useful. We can time how long the `KDTree` takes to do its exact nearest neighbor computation to get a gauge on what PyNNDescent is buying us before we get to the issue of comparing the results of the exact and approximate algorithms.

[12]:
```
%%time
tree_index = KDTree(fmnist_train)
```

```
CPU times: user 15.2 s, sys: 177 ms, total: 15.3 s
Wall time: 15.6 s
```

A faster build time for the tree itself. The question is how well the tree index works when querying new points…

[13]:
```
%%time
tree_neighbors = tree_index.query(fmnist_test, k=10)
```

```
CPU times: user 10min 19s, sys: 3.24 s, total: 10min 22s
Wall time: 11min 12s
```

Ouch! We can see now what PyNNDescent is buying us in terms of querying speed (though, to be fair, this is high dimensional data that is a worst case scenario for kd-trees). We did have to give up some accuracy to get that speed-up however. The real question is how much accuracy did we lose? How approximate are our nearest neighbors?

We can write a short function to take some approximate neighbors and the true neighbors and output the proportion of neighbors that the apprxomation got correct for each and every query point (i.e. each point of the test set).

[14]:
```
def accuracy_per_query_point(approx_neighbors, true_neighbors):
    approx_indices = approx_neighbors[0]
    true_indices = true_neighbors[1]
    result = np.zeros(approx_indices.shape[0])
    for i in range(approx_indices.shape[0]):
        n_correct = np.intersect1d(approx_indices[i], true_indices[i]).shape[0]
        result[i] = n_correct / true_indices.shape[1]
    return result
```

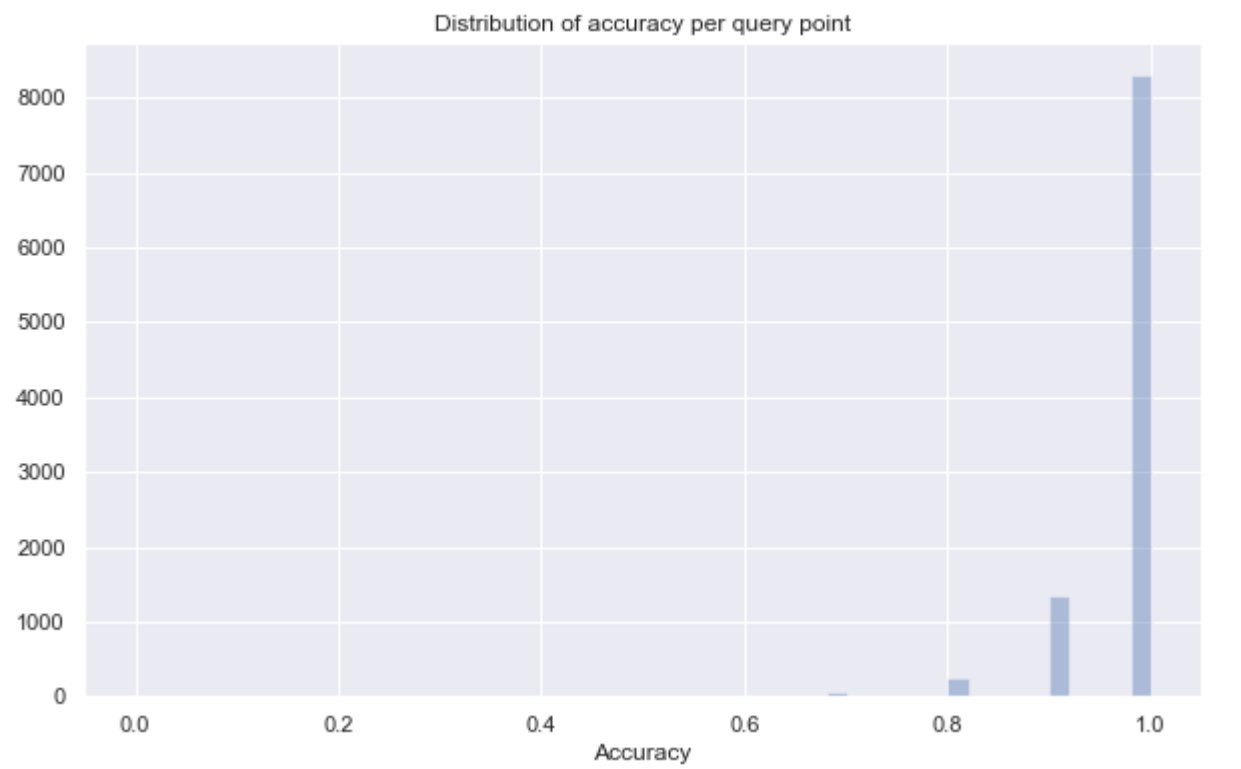
[15]:
```
accuracy_stats = accuracy_per_query_point(neighbors, tree_neighbors)
```

[16]:
```
import seaborn as sns
import matplotlib.pyplot as plt

sns.set(rc={"figure.figsize":(10,6)})
```

[17]:
```
sns.distplot(accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(accuracy_stats)}")
```

```
Average accuracy of 0.9781
```



So we are getting 100% accuracy for the vast majority of query points, but there are a thousand or so that get one

neighbor wrong, and a very few that get two or three wrong. Even then we are likely getting the eleventh and twelfth nearest neighbors in those cases. So in general we have a very high accuracy given how much faster the query is.

2.1.1 Query parameters

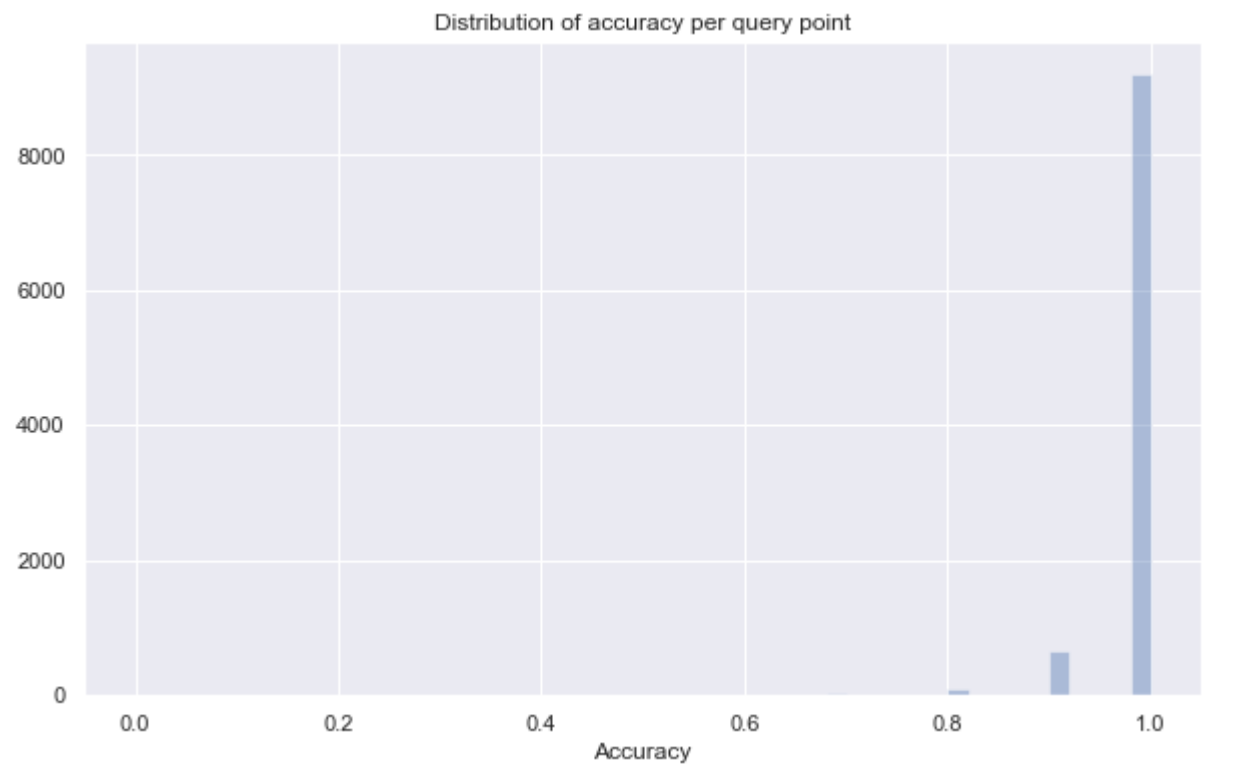
What if we want to be more accurate? There are a few ways to go about this. The most obvious is to simply ask for more neighbors, and then the top 10 will be more accurate. We can do this by specifying `k` in the query (just as we did for the KDTree – it is just that PyNNDescent defaults to 10 neighbors).

[18]:
```
%%time
more_neighbors = index.query(fmnist_test, k=15)
```

```
CPU times: user 1.1 s, sys: 486 ms, total: 1.59 s
Wall time: 1.99 s
```

[19]:
```
more_accuracy_stats = accuracy_per_query_point(more_neighbors, tree_neighbors)
sns.distplot(more_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(more_accuracy_stats)}")
```

```
Average accuracy of 0.99021
```



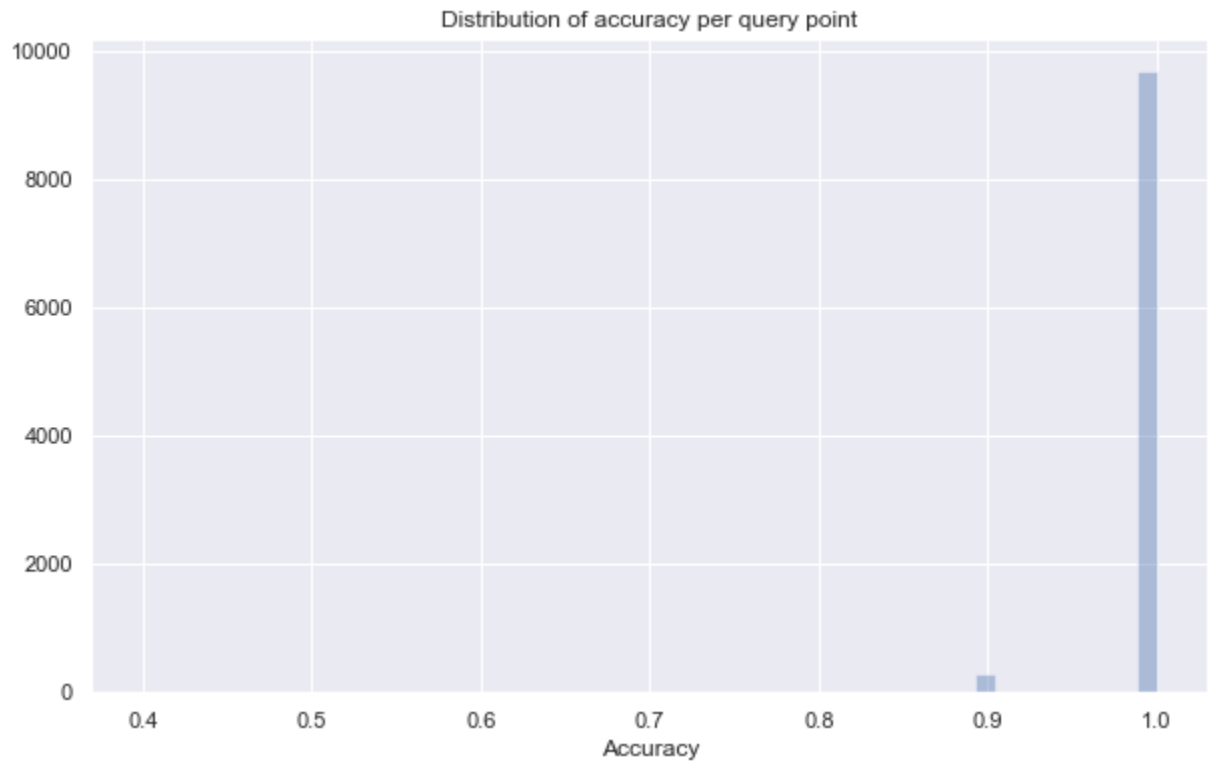
An alternative approach is to tolerate a little more backtracking in the approximate search. This is controlled by the parameter `epsilon`. The larger the value the more backtracking the algorithm will tolerate, and the more accurate it will be (at the cost of greater search time). The default value is `0.1` and for euclidean distance going to `0.2` or even `0.3` might make sense. You can, of course, also turn it down to `0.0` and do *no* backtracking for even faster search (but reduced accuracy). Let's try both options; greater accuracy first.

[20]:
```
%%time
better_neighbors = index.query(fmnist_test, epsilon=0.2)
```

```
CPU times: user 1.35 s, sys: 18.2 ms, total: 1.37 s
Wall time: 1.39 s
```

[21]:
```
better_accuracy_stats = accuracy_per_query_point(better_neighbors, tree_neighbors)
sns.distplot(better_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(better_accuracy_stats)}")
```

```
Average accuracy of 0.9965
```



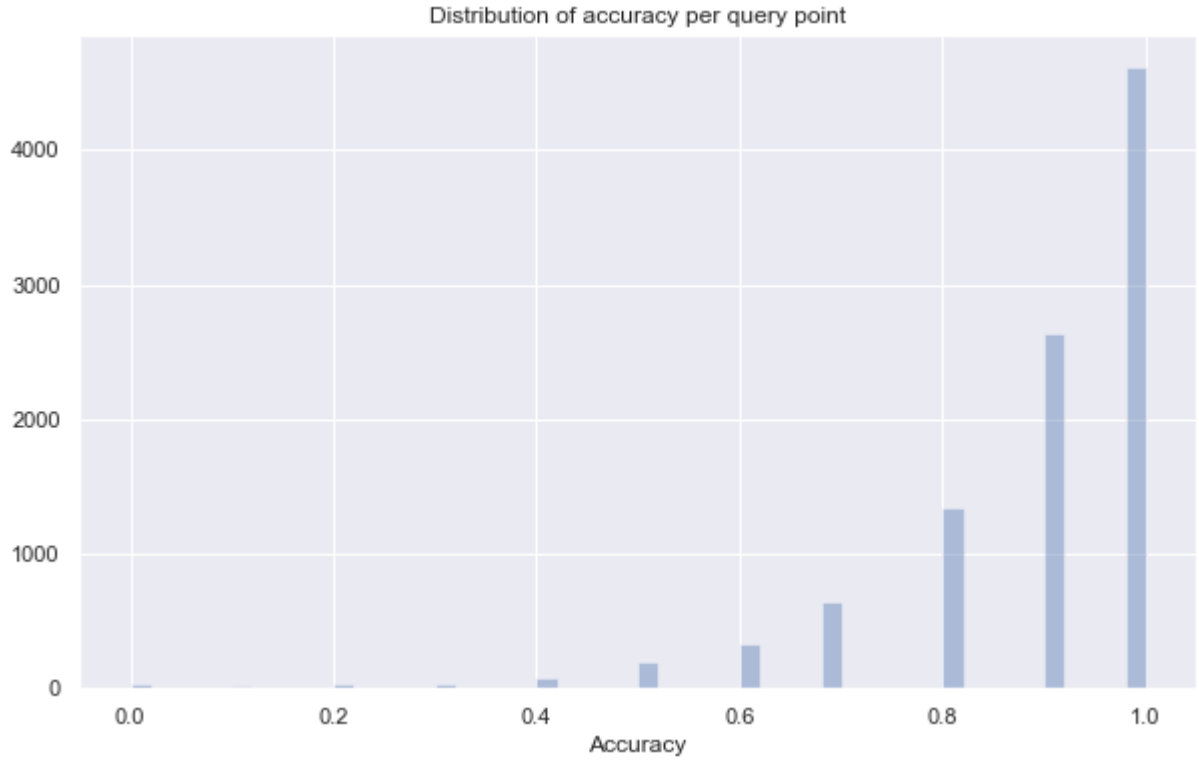
With `epsilon=0.2` we see a great improvement in accuracy. Everything but a very small number of query points get all 10 neighbors correct! We took a little more time, but still vastly less than the kd-tree. We also were significantly more accurate than simply taking more neighbors for very little extra cost in search time. What if we go the other way and turn `epsilon` down to zero?

[22]:
```
%%time
worse_neighbors = index.query(fmnist_test, epsilon=0.0)
```

```
CPU times: user 456 ms, sys: 6.8 ms, total: 462 ms
Wall time: 471 ms
```

[23]:
```
worse_accuracy_stats = accuracy_per_query_point(worse_neighbors, tree_neighbors)
sns.distplot(worse_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(worse_accuracy_stats)}")
```

```
Average accuracy of 0.88776
```



The accuracy has dropped quite a bit now, but if we only need to be "close" then this might be good enough, and it is definitely faster. This provides an easy trade-off between accuracy and query time.

2.1.2 Index parameters

It is often the case that we know what we really need is fast queries – as fast as possible – and we are willing to sacrifice accuracy to get there. We know we are going to do a lot of queries, so it would be better if we could set up the index itself to target faster querying rather than just hoping the `epsilon` parameter will get things fast enough.

Alternatively we might really want very accurate queries – ideally exact queries, but they are too expensive – and be prepared to put in more work on the index to make queries more accurate but still fast.

These options can be dealt with via various parameters that are passed to the index constructor. There are many parameters that can be tweaked (check the docstring for full details), but in practice there are only a few we need to concern ourselves with. The primary ones of interest are

1. `n_neighbors`;
2. `diversify_prob`; and
3. `pruning_degree_multiplier`.

We'll look at each in turn to get a quick understanding of what they do. First up is `n_neighbors`. By default this is `30`; more neighbors leads to a more accurate (albeit slower) index, fewer neighbors leads to a faster index at the cost of accuracy. In general for a high accuracy (on the order or 90%+) index you might want an `n_neighbors` value in the range of fifty to one-hundred (a lot will depend on other issues, such as the dataset itself, and the metric used – angular metrics generally need higher `n_neighbors` values). For fast indexes you can set it more in the range of five to twenty (again, dependent on the dataset itself and the metric; a little experimentation may be required). Let's try a couple of values with the fashion-mnist dataset and see how it works.

[24]:
```
%%time
accurate_index = pynndescent.NNDescent(fmnist_train, n_neighbors=50)
accurate_index.prepare()
```

```
CPU times: user 1min 21s, sys: 1.77 s, total: 1min 23s
Wall time: 35.1 s
```

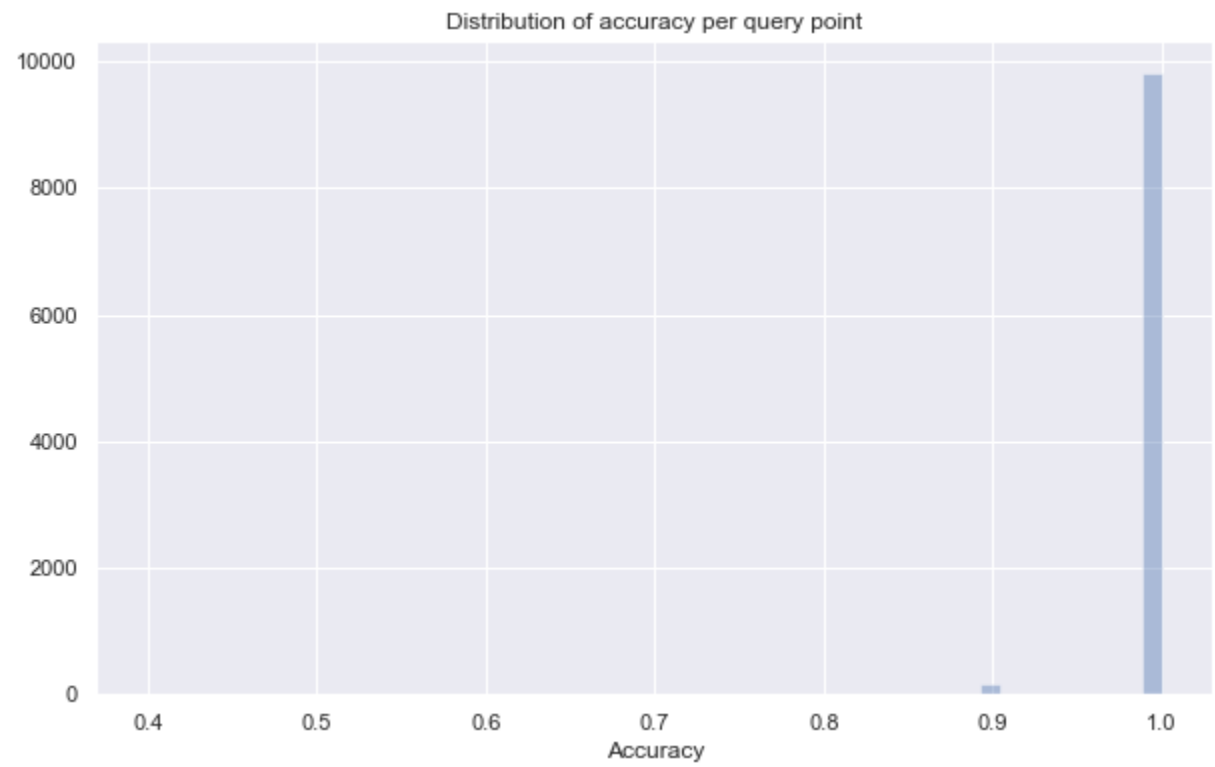
It takes a little longer to construct the index. Querying will also be a little slower.

[25]:
```
%%time
accurate_neighbors = accurate_index.query(fmnist_test, epsilon=0.2)
```

```
CPU times: user 1.39 s, sys: 20.2 ms, total: 1.41 s
Wall time: 1.42 s
```

[26]:
```
acc_index_accuracy_stats = accuracy_per_query_point(accurate_neighbors, tree_
↪neighbors)
sns.distplot(acc_index_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(acc_index_accuracy_stats)}")
```

```
Average accuracy of 0.9980899999999999
```



But we see that we are now essentially just about perfect with our accuracy – only a few query points have any neighbors at all that aren't true nearest neighbors (and they are almost certainly very close).

Now we can go the other way, and try to make a very very fast index (supposing we want to make an awful lot of queries at a very high throughput rate). We can decrease `n_neighbors`.

[27]:
```
%%time
fast_index = pynndescent.NNDescent(fmnist_train, n_neighbors=5)
fast_index.prepare()
```

```
/Users/leland/anaconda3/envs/numba51/lib/python3.8/site-packages/scipy/sparse/_index.
↪py:124: SparseEfficiencyWarning: Changing the sparsity structure of a csr_matrix is␣
↪expensive. lil_matrix is more efficient.
  self._set_arrayXarray(i, j, x)
```

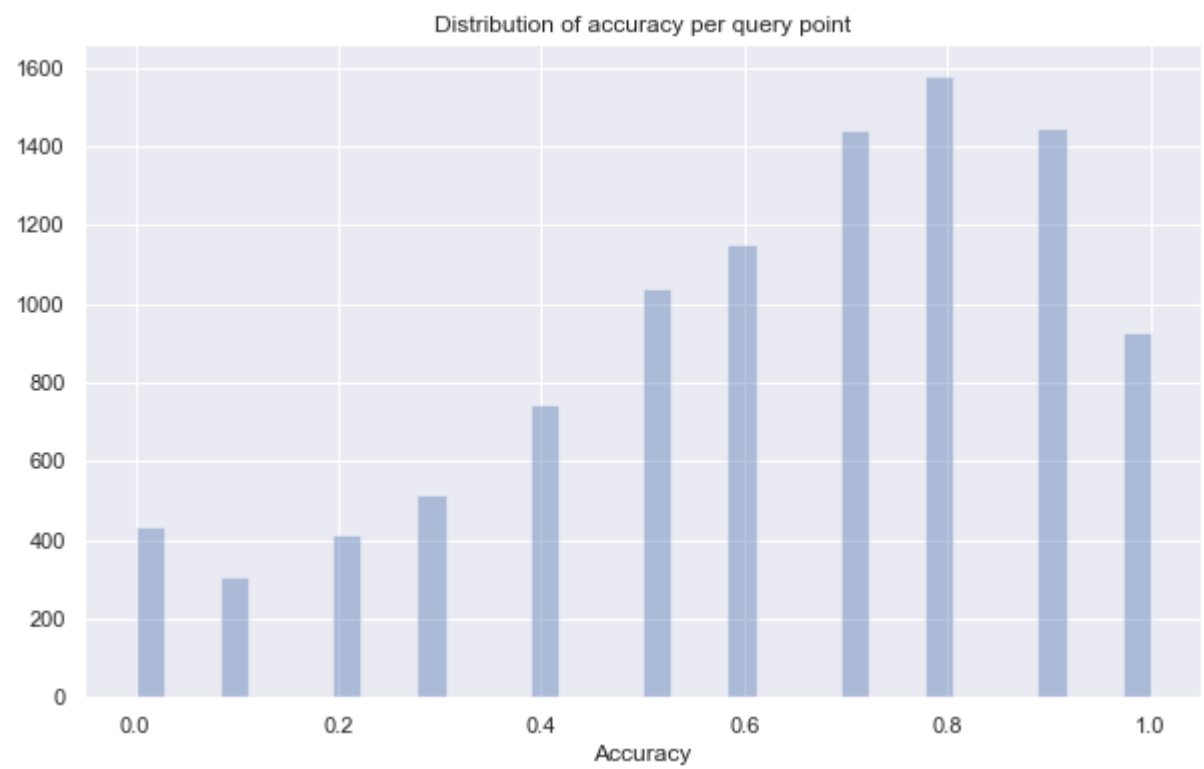
```
CPU times: user 23.5 s, sys: 974 ms, total: 24.5 s
Wall time: 9.15 s
```

[28]:
```
%%time
fast_neighbors = fast_index.query(fmnist_test, epsilon=0.0)
```

```
CPU times: user 290 ms, sys: 13.8 ms, total: 304 ms
Wall time: 305 ms
```

[29]:
```
fast_index_accuracy_stats = accuracy_per_query_point(fast_neighbors, tree_neighbors)
sns.distplot(fast_index_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(fast_index_accuracy_stats)}")
```

```
Average accuracy of 0.6279
```



The results are not that accurate – there are some query points that got no neighbors right (although it likely still got close-by points). However is speed is of the essence we have gotten under the time we were previously capable of with an `epsilon` of `0.0`. Depending on your needs, and the nature of your dataset you can play with `n_neighbors` to find the trade-off between speed and accuracy that best suits you.

The next step is `diversify_prob`. When PyNNDescent is constructing it's graph based index it tries to "diversify"

the edges, pruning away some that are largely redundant. The value of `diversify_prob` is the probability that an edge identified as redundant will get pruned. The more edges that get pruned away the less accurate the index is, but with fewer edges searching becomes much faster. If you want as accurate an index as you can get then turn `diversify_prob` to `0.0`. For speed you should obviously use `1.0` and prune away as many edges as you can. The default value is, in fact, `1.0` since it is usually worth it to edges identified by the diversify step. Let's try setting it to `0.0` and see how that effects speed and accuracy.

[30]:
```
%%time
accurate_index = pynndescent.NNDescent(fmnist_train, n_neighbors=50, diversify_prob=0.
↪0)
accurate_index.prepare()
```

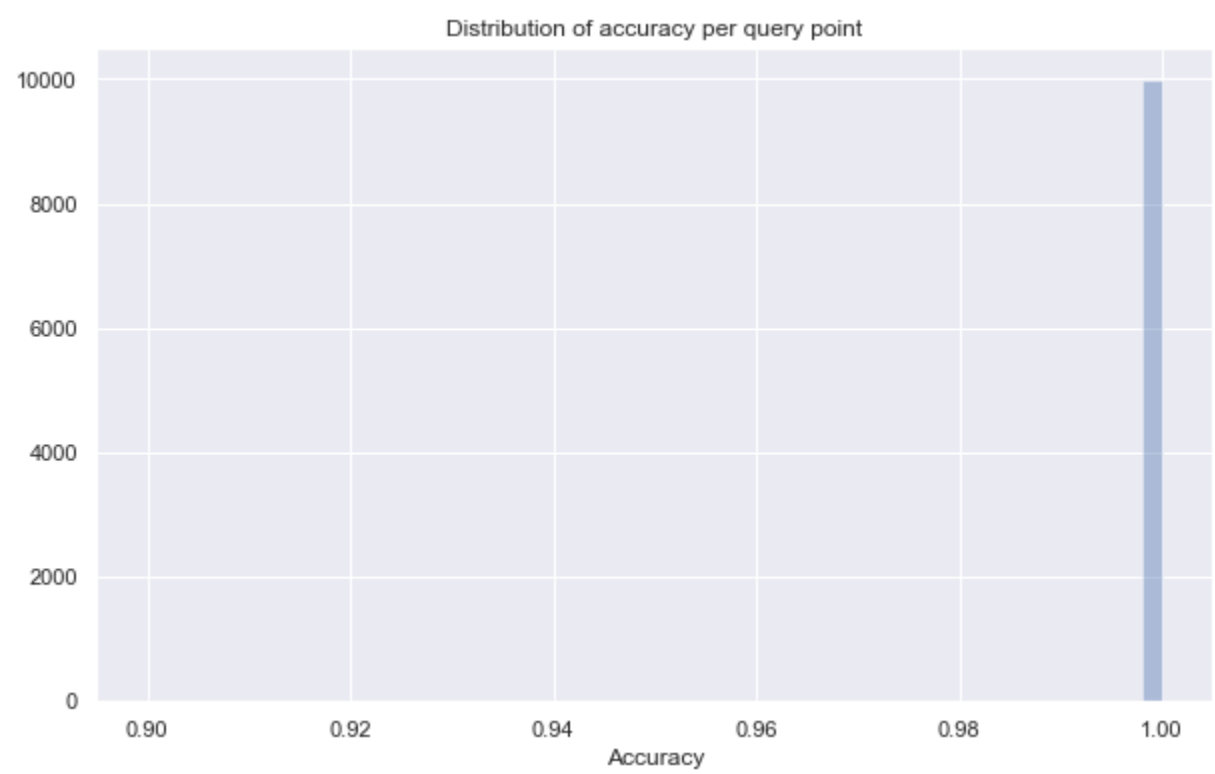
```
CPU times: user 1min 47s, sys: 1.07 s, total: 1min 48s
Wall time: 40.6 s
```

[31]:
```
%%time
accurate_neighbors = accurate_index.query(fmnist_test, epsilon=0.2)
```

```
CPU times: user 3.04 s, sys: 37.6 ms, total: 3.07 s
Wall time: 3.12 s
```

[32]:
```
acc_index_accuracy_stats = accuracy_per_query_point(accurate_neighbors, tree_
↪neighbors)
sns.distplot(acc_index_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(acc_index_accuracy_stats)}")
```

```
Average accuracy of 0.9999299999999999
```



At this point out of ten-thousand query points, each with 10 neighbors, we are getting a grand total of eight wrong – that's eight out of a hundred-thousand! Of course our query time is up to several seconds now, but that still beats the

many minutes of the kd-trees by a wide margin.

We still have one parameter left however – the `pruning_degree_multplier`. This gives the multiple of the number of neighbors many edges that any given vertex in the search graph is allowed to have. So if `n_neighbors` is 30 and the `pruning_degree_multiplier` is the default `1.5` then any vertex in the graph can have at most forty-five edges connected to it. This prevents hubs (which form in high dimensional spaces) causing the search stage to blow out in complexity. It can be tuned, however, to trade off between speed and accuracy. Higher multiples result in more accurate graphs with more edges that take longer to search. Low multiples (including less than one!) result in graphs with many fewer edges that are fast to search, but may get stuck in local minima or fail to find the true nearest neighbors. Let's play with that dial for a moment…

[33]:
```
%%time
accurate_index = pynndescent.NNDescent(
    fmnist_train,
    n_neighbors=50,
    diversify_prob=0.0,
    pruning_degree_multiplier=3.0
)
accurate_index.prepare()
```

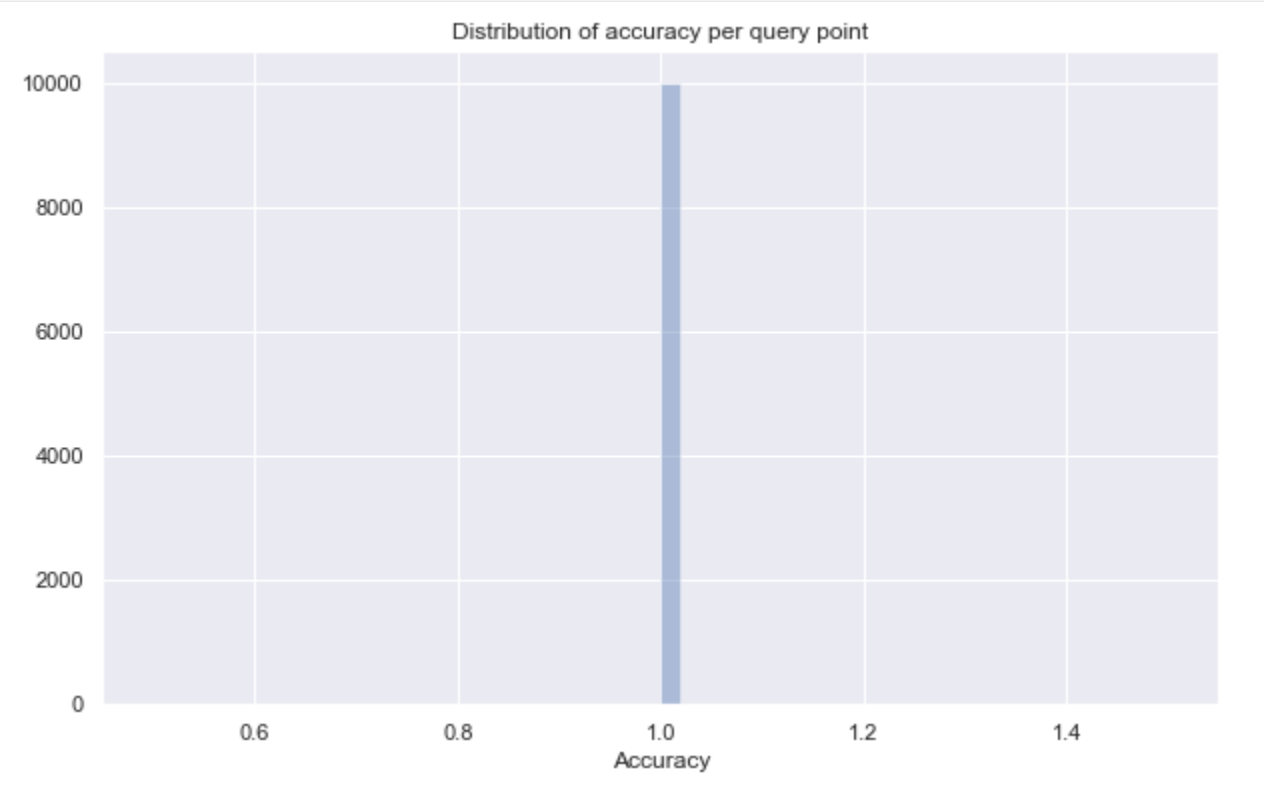
```
CPU times: user 1min 47s, sys: 1.35 s, total: 1min 48s
Wall time: 41.2 s
```

[34]:
```
%%time
accurate_neighbors = accurate_index.query(fmnist_test, epsilon=0.2)
```

```
CPU times: user 3.65 s, sys: 69.9 ms, total: 3.72 s
Wall time: 3.71 s
```

[35]:
```
acc_index_accuracy_stats = accuracy_per_query_point(accurate_neighbors, tree_
↪neighbors)
sns.distplot(acc_index_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(acc_index_accuracy_stats)}")
```

```
Average accuracy of 1.0
```



By turning up the `pruning_degree_multiplier` we have achieved perfect accuracy – and still at a very tiny fraction of the query time required for the exact kd-tree query. Of course on a different more complex dataset these same settings may not suffice. Ultimately what we lose is the *guarantee* that we will get the true nearest neighbors.

We can turn the dial the other way, however, and get very fast searching that is potentially still fairly accurate (compared to the very low `n_neighbors` version).

[36]:
```
%%time
fast_index = pynndescent.NNDescent(
    fmnist_train,
    n_neighbors=10,
    diversify_prob=1.0,
    pruning_degree_multiplier=0.5,
)
fast_index.prepare()
```

```
/Users/leland/anaconda3/envs/numba51/lib/python3.8/site-packages/scipy/sparse/_index.
↪py:124: SparseEfficiencyWarning: Changing the sparsity structure of a csr_matrix is␣
↪expensive. lil_matrix is more efficient.
  self._set_arrayXarray(i, j, x)
```

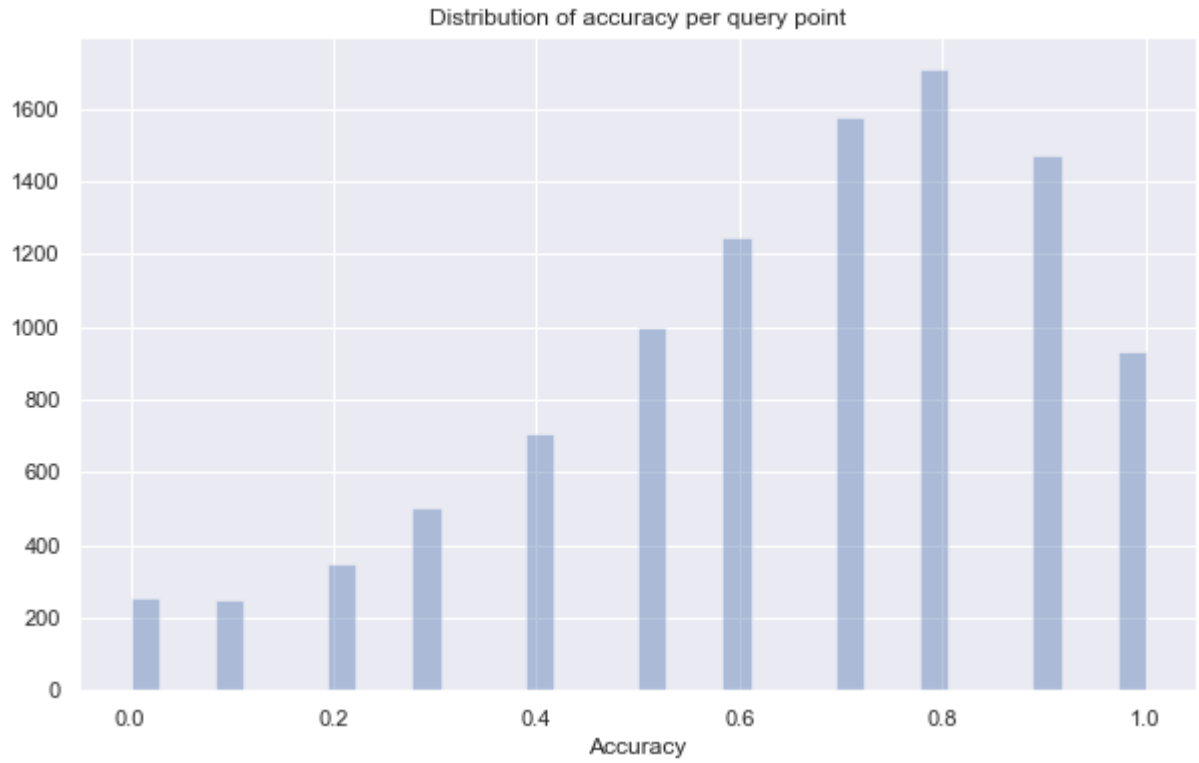
```
CPU times: user 27.2 s, sys: 1.14 s, total: 28.3 s
Wall time: 11.3 s
```

[37]:
```
%%time
fast_neighbors = fast_index.query(fmnist_test, epsilon=0.0)
```

```
CPU times: user 302 ms, sys: 16 ms, total: 318 ms
Wall time: 317 ms
```

[38]:
```
fast_index_accuracy_stats = accuracy_per_query_point(fast_neighbors, tree_neighbors)
sns.distplot(fast_index_accuracy_stats, kde=False)
plt.title("Distribution of accuracy per query point")
plt.xlabel("Accuracy")
print(f"Average accuracy of {np.mean(fast_index_accuracy_stats)}")
```

```
Average accuracy of 0.65031
```



In general this isn't a good idea, but it gives you an idea of the trade-offs that can be made.

2.1.3 Nearest neighbors of the training set

While querying for nearest neighbors of new points is a common use case, for many tasks, such as manifold learning or density based clustering, you want the nearest neighbors of all the points in the training set. You could build the index and then query for all the training points – but because PyNNDescent builds a nearest neighbor graph as part of its index you can actually get there faster by just building the index and extracting the results directly. This is, for example, what libraries like UMAP and OpenTSNE do.

This is why the `prepare` step is kept separate. It is extra overhead, preparing the index to be queried for new unseen data, that isn't required for the use case of simply extracting the nearest neighbors of the full training set. Thus if we just build, but don't prepare, the index:

[39]:
```
%%time
index = pynndescent.NNDescent(fmnist_train)
```

```
CPU times: user 36.5 s, sys: 375 ms, total: 36.9 s
Wall time: 14.8 s
```

We can then extract the nearest neighbors of each training sample by using the `neighbor_graph` attribute.

[40]:
```
index.neighbor_graph
```

[40]:
```
(array([[    0, 25719, 27655, ..., 38300,  4643,  7353],
        [    1, 42564, 37550, ..., 12169, 50358, 49552],
        [    2, 53513, 35424, ..., 16891, 29542, 40182],
        ...,
        [59997, 45348, 22272, ..., 58896, 53292, 11657],
        [59998, 17378,  8495, ...,  4389, 23426, 34912],
        [59999, 11912, 40600, ..., 31966, 45245, 58489]]),
 array([[   0.     , 1188.7826 , 1215.344  , ..., 1417.4388 , 1422.567  ,
         1424.7806 ],
        [   0.     , 1048.0482 , 1068.395  , ..., 1219.923  , 1222.1665 ,
         1225.5464 ],
        [   0.     ,  532.61993,  632.16534, ...,  885.45245,  886.3972 ,
          887.8125 ],
        ...,
        [   0.     , 1086.2596 , 1118.3738 , ..., 1303.3699 , 1306.9954 ,
         1308.5958 ],
        [   0.     ,  685.18243,  690.4209 , ...,  795.6589 ,  796.894  ,
          797.1349 ],
        [   0.     ,  884.41394,  886.4903 , ..., 1048.9224 , 1049.3502 ,
         1051.806  ]], dtype=float32))
```

The first array is `n_samples` by `n_neighbors` such that the `i`th row contains the `n_neighbors` nearest neighbors of the `i`th data point. The second array is the associated distances to each of those neighboring points. From the distance array you can see that each row is sorted, with the closest neighbors first, and the furthest last.

If you are simply wanting to extract this sort of data for further processing this is the fastest way to do it, and since you only need to run the index build and not actually query for any data, this can often be faster than some other approximate nearest neighbor techniques that need to be queried after being constructed.

2.2 PyNNDescent with different metrics

In the initial tutorial we looked at how to get PyNNDescent running on your data, and how to query the indexes it builds. Implicit in all of that was the measure of distance used to determine what counts as the "nearest" neighbors. By default PyNNDescent uses the euclidean metric (because that is what people generally expect when they talk about distance). This is not the only way to measure distance however, and is often not the right choice for very high dimensional data for example. Let's look at how to use PyNNDescent with other metrics.

First we'll need some libraries, and some test data. As before we will use ann-benchmarks for data, so we will reuse the data download function from the previous tutorial.

[1]:
```
import pynndescent
import numpy as np
import h5py
from urllib.request import urlretrieve
import os
```

[2]:
```
def get_ann_benchmark_data(dataset_name):
    if not os.path.exists(f"{dataset_name}.hdf5"):
        print(f"Dataset {dataset_name} is not cached; downloading now ...")
        urlretrieve(f"http://ann-benchmarks.com/{dataset_name}.hdf5", f"{dataset_name}
↪.hdf5")
    hdf5_file = h5py.File(f"{dataset_name}.hdf5", "r")
    return np.array(hdf5_file['train']), np.array(hdf5_file['test']), hdf5_file.attrs[
↪'distance']
```

2.2.1 Built in metrics

Let's grab some data where euclidean distance doesn't make sense. We'll use the NY-Times dataset, which is a TF-IDF matrix of data generated from NY-Times news stories. The particulars are less important here, but what matters is that the most sensible way to measure distance on this data is with an angular metric, such as cosine distance.

[3]:
```
nytimes_train, nytimes_test, distance = get_ann_benchmark_data('nytimes-256-angular')
nytimes_train.shape
```

[3]:
```
(290000, 256)
```

Now that we have the data we can check the distance measure suggested by ann-benchmarks

[4]:
```
distance
```

[4]:
```
'angular'
```

So an angular measure of distance – cosine distance will suffice. How do we manage to get PyNNDescent working with cosine distance (which isn't even a real metric! it violates the triangle inequality) instead of standard euclidean?

[5]:
```
%%time
index = pynndescent.NNDescent(nytimes_train, metric="cosine")
index.prepare()
```

```
/Users/leland/anaconda3/envs/umap_0.5dev/lib/python3.8/site-packages/scipy/sparse/_
↪index.py:126: SparseEfficiencyWarning: Changing the sparsity structure of a csr_
↪matrix is expensive. lil_matrix is more efficient.
  self._set_arrayXarray(i, j, x)
```

```
CPU times: user 5min 29s, sys: 6.4 s, total: 5min 35s
Wall time: 2min 9s
```

That's right, it uses the scikit-learn standard of the `metric` keyword and accepts a string that names the metric. We can now query the index, and it will use that metric in the query as well.

[6]:
```
%%time
neighbors = index.query(nytimes_train)
```

```
CPU times: user 20.3 s, sys: 387 ms, total: 20.7 s
Wall time: 21.2 s
```

```
/Users/leland/anaconda3/envs/umap_0.5dev/lib/python3.8/site-packages/pynndescent/
↪pynndescent_.py:1628: RuntimeWarning: invalid value encountered in correct_
↪alternative_cosine
  dists = self._distance_correction(dists)
```

It is worth noting at this point that these results will probably be a little sub-optimal since angular distances are harder to index, and as a result to get the same level accuracy in the nearest neighbor approximation we should be using a larger value than the default `30` for `n_neighbors`. Beyond that, however, nothing else changes from the tutorial earlier – except that we can't use kd-trees to learn the true neighbors, since they require distances that respect the triangle inequality.

How many metrics does PyNNDescent support out of the box? Quite a few actually:

[7]:
```
for dist in pynndescent.distances.named_distances:
    print(dist)
```

```
euclidean
l2
sqeuclidean
```

(continues on next page)

(continued from previous page)

```
manhattan
taxicab
l1
chebyshev
linfinity
linfty
linf
minkowski
seuclidean
standardised_euclidean
wminkowski
weighted_minkowski
mahalanobis
canberra
cosine
dot
correlation
hellinger
haversine
braycurtis
spearmanr
kantorovich
wasserstein
tsss
true_angular
hamming
jaccard
dice
matching
kulsinski
rogerstanimoto
russellrao
sokalsneath
sokalmichener
yule
```

Some of these are repeats or alternate names for the same metric, and some of these are fairly simple, but others, such as `spearmanr`, or `hellinger` are useful statistical measures not often implemented elsewhere, and others, such as `wasserstein` are complex and hard to compute metrics. Having all of these readily available in a fast approximate nearest neighbor library is one of PyNNDescent's strengths.

2.2.2 Custom metrics

We can go even further in terms of interesting metrics however. You can write your own custom metrics and hand them to PyNNDescent to use on your data. There, of course, a few caveats with this. Many nearest neighbor libraries allow for the possibility of user defined metrics. If you are using Python this often ends up coming in two flavours:

1. Write some C, C++ or Cython code and compile it against the library itself
2. Write a python distance function, but lose almost all performance

With PyNNDescent we get a different trade-off. Because we use Numba for just-in-time compiling of Python code instead of a C or C++ backend you don't need to do an offline compilation step and can instead have your custom Python distance function compiled and used on the fly. The cost for that is that the custom distance function you write must be a numba jitted function. This, in turn, means that you can only use Python functionality that is **'supported by numba <>'__**. That is still a fairly large amount of functionality, especially when we are talking about numerical

work, but it is a limit. It also means that you will need to import numba and decorate your custom distance function accordingly. Let's look at how to do that.

[8]:
```
import numba
```

Let's start by simply implementing euclidean distance where $d(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_i (\mathbf{x}_i - \mathbf{y}_i)^2}$. This is already implemented in PyNNDescent, but it is a simple distance measure that everyone knows and will serve to illustrate the process. First let's write the function – using numpy functionality this will be fairly short:

[9]:
```
def euclidean(x, y):
    return np.sqrt(np.sum((x - y)**2))
```

Now we need to get the function compiled so PyNNDescent can use it. That is actually as easy as adding a decorator to the top of the function telling numba that it should compile the function when it gets called.

[10]:
```
@numba.jit
def euclidean(x, y):
    return np.sqrt(np.sum((x - y)**2))
```

We can now pass this function directly to PyNNdescent as a metric and everything will "just work". We'll just train on the smaller test set since it will take a while.

[11]:
```
%%time
index = pynndescent.NNDescent(nytimes_test, metric=euclidean)
```

```
CPU times: user 21.5 s, sys: 220 ms, total: 21.7 s
Wall time: 12 s
```

This is a little slower than we might have expected, and that's because a great deal of the computation time is spent evaluating that metric. While numba will compile what we wrote we can make it a little faster if we look through the numba performance tips documentation. The two main things to note are that we can use explicit loops instead of numpy routines, and we can add arguments to the decorator such as `fastmath=True` to speed things up a little. Let's rewrite it:

[12]:
```
@numba.jit(fastmath=True)
def euclidean(x, y):
    result = 0.0
    for i in range(x.shape[0]):
        result += (x[i] - y[i])**2
    return np.sqrt(result)
```

[13]:
```
%%time
index = pynndescent.NNDescent(nytimes_test, metric=euclidean)
```

```
CPU times: user 12.3 s, sys: 116 ms, total: 12.4 s
Wall time: 8.53 s
```

That is faster! If we are really on the hunt for performance however, you might note that, for the purposes of finding nearest neighbors the exact values of the distance are not as important as the ordering on distances. In other words we could use the square of euclidean distance and we would get all the same neighbors (since the square root is a monotonic order preserving function of squared euclidean distance). That would, for example, save us a square root computation. We could do the square roots afterwards to just the distances to the nearest neighbors. Let's reproduce what PyNNDescent actually uses internally for euclidean distance:

[14]:
```
@numba.njit(
    [
        "f4(f4[::1],f4[::1])",
```

(continues on next page)

(continued from previous page)

```
        numba.types.float32(
            numba.types.Array(numba.types.float32, 1, "C", readonly=True),
            numba.types.Array(numba.types.float32, 1, "C", readonly=True),
        ),
    ],
    fastmath=True,
    locals={
        "result": numba.types.float32,
        "diff": numba.types.float32,
        "dim": numba.types.uint32,
        "i": numba.types.uint16,
    },
)
def squared_euclidean(x, y):
    r"""Squared euclidean distance.

    .. math::
        D(x, y) = \sum_i (x_i - y_i)^2
    """
    result = 0.0
    dim = x.shape[0]
    for i in range(dim):
        diff = x[i] - y[i]
        result += diff * diff

    return result
```

That is definitely more complicated! Most of it, however, is arguments to the decorator giving it extra typing information to let it squeeze out every drop of performance possible. By default numba will infer types, or even compile different versions for the different types it sees. With a little extra information, however, it can make smarter decisions and optimizations during compilation. Let's see how fast that goes:

[15]:
```
%%time
index = pynndescent.NNDescent(nytimes_test, metric=squared_euclidean)
```

```
CPU times: user 10.6 s, sys: 96.2 ms, total: 10.7 s
Wall time: 7.64 s
```

Definitely faster again – so there are significant gains to be had if you are willing to put in some work to write your function. Still, the naive approach we started with, just decorating the obvious implementation, did very well, so unless you desperately need top tier performance for your custom metric a straightforward approach will suffice. And for comparison here is the tailored C++ implementation that libraries like nmslib and hnswlib use:

```
static float
L2SqrSIMD16Ext(const void *pVect1v, const void *pVect2v, const void *qty_ptr) {
    float *pVect1 = (float *) pVect1v;
    float *pVect2 = (float *) pVect2v;
    size_t qty = *((size_t *) qty_ptr);
    float PORTABLE_ALIGN32 TmpRes[8];
    size_t qty16 = qty >> 4;

    const float *pEnd1 = pVect1 + (qty16 << 4);

    __m256 diff, v1, v2;
    __m256 sum = _mm256_set1_ps(0);
```

(continues on next page)

(continued from previous page)

```
    while (pVect1 < pEnd1) {
        v1 = _mm256_loadu_ps(pVect1);
        pVect1 += 8;
        v2 = _mm256_loadu_ps(pVect2);
        pVect2 += 8;
        diff = _mm256_sub_ps(v1, v2);
        sum = _mm256_add_ps(sum, _mm256_mul_ps(diff, diff));

        v1 = _mm256_loadu_ps(pVect1);
        pVect1 += 8;
        v2 = _mm256_loadu_ps(pVect2);
        pVect2 += 8;
        diff = _mm256_sub_ps(v1, v2);
        sum = _mm256_add_ps(sum, _mm256_mul_ps(diff, diff));
    }

    _mm256_store_ps(TmpRes, sum);
    return TmpRes[0] + TmpRes[1] + TmpRes[2] + TmpRes[3] + TmpRes[4] + TmpRes[5] +␣
↪TmpRes[6] + TmpRes[7];
}
```

Comparatively, the python code, even with its extra numba decorations, looks pretty straightforward. Notably (at last testing) the numba code and this C++ code (when suitably compiled with AVX flags etc.) have essentially indistinguishable performance. Numba is awfully good at finding optimizations for numerical code.

Beware of bounded distances

There is one remaining caveat on custom distance functions that is important. Many distances, such as cosine distance and jaccard distance are bounded: the values always fall in some fixed finite range (in these cases between 0 and 1). When querying new data points against an index PyNNDescent bounds the search by some multiple (1 + epsilon) of the most distant of the the top k neighbors found so far. This allows a limited amount of backtracking and avoids getting stuck in local minima. It does, however, not play well with bounded distances – a small but non-zero epsilon can end up failing to bound the search at all (suppose epsilon is 0.2 and the most distant of the the top k neighbors has cosine distance 0.8 for example). The trick to getting around this is the same trick described above when we decided not to bother taking the square root of the euclidean distance – we can apply transform to the distance values that preserves all ordering. This means that, for example, internally PyNNDescent uses the *negative log* of the cosine *similarity* instead of cosine distance (and converts the distance values when done). You will want to use a similar trick if your distance function has a strict finite upper bound.

2.3 Working with sparse data

Not all data conveniently fits in numpy arrays; sometimes a lot of the data entries are zero and we want to use a sparse data storage format. This is especially common for extremely high dimensional data (data with thousands, or even hundreds of thousands of dimensions). Such data is a lot harder to work with for many tasks, including nearest neighbor search. Let's see how we can work with sparse data like this in PyNNDescent.

First we'll need some data. For that let's use a standard NLP dataset that we can pull together with scikit-learn.

[9]:
```
import pynndescent
import sklearn.datasets
import sys
```

We need to fetch the train and test sets separately, but conveniently the data has already been converted from the text of newsgroup messages into TF-IDF matrices. This means that we have a feature column for each word in the vocabulary – though often the vocabulary is pruned a little.

[2]:
```
news_train = sklearn.datasets.fetch_20newsgroups_vectorized(subset='train')
news_test = sklearn.datasets.fetch_20newsgroups_vectorized(subset='test')
```

Now that we have the data let's see what it looks like:

[3]:
```
news_train.data
```

[3]:
```
<11314x130107 sparse matrix of type '<class 'numpy.float64'>'
        with 1787565 stored elements in Compressed Sparse Row format>
```

Not a numpy array! Instead it is a SciPy sparse matrix. It has 11314 rows (so not many data samples), but 130107 columns (a *lot* of features – as noted, one for each word in the vocabulary). Despite that large size there are only 17876565 non-zero entries. The trick with sparse matrices is that they only store information about those entries that aren't zero – they need to keep track of where they are, and what the value is, but they can ignore all the zero entries. If this were a raw numpy array with all those zeros in place it would have . . .

[7]:
```
11314 * 130107
```

[7]:
```
1472030598
```

. . . a lot of entries. To store all of that in memory would require (at 4 bytes per entry) . . .

[8]:
```
(11314 * 130107 * 4) / 1024**3
```

[8]:
```
5.48374130576849
```

. . . almost 5.5 GB! That is possible, but likely impractical on a laptop. And this is for a case with a small number of data samples. With more samples the size would grow enormous very quickly indeed. Instead we have an object that uses . . .

[13]:
```
(
    news_train.data.data.nbytes
     + news_train.data.indices.size
     + news_train.data.indptr.nbytes
) / 1024**2
```

[13]:
```
15.385956764221191
```

. . . only 15 MB. You will also note that to extract that information required poking at some of the internal attributes of the sparse matrix (`data`, `indices`, and `indptr`). This is the downside of the sparse format – they are more complicated to work with. It is certainly the case that many tools are simply not able to deal with these sparse structures at all, and the data would need to be cast to a numpy array and take up that full amount of memory.

Fortunately PyNNDescent is built to work with sparse matrix data. To see that in practice let's hand the sparse matrix directly to `NNDescent` and watch it work.

[4]:
```
%%time
index = pynndescent.NNDescent(news_train.data, metric="cosine")
```

```
CPU times: user 7min 3s, sys: 1.77 s, total: 7min 5s
Wall time: 2min 24s
```

You will note that this is a much longer index construction time than we would normally expect with only around eleven thousand data points – there is overhead in working with sparse matrices that makes it slower. That, combined with the fact that the data has over a hundred and thirty-thousand dimensions means this is a computationally intensive

task. Still it is likely better than working with the full 5.5 GB numpy array, and certainly better when dealing with larger sparse matrices where there is simply no way to instantiate a numpy array large enough to hold the data.

We can query the index – but we have to use the same sparse matrix structure (we can't query with numpy arrays for am index built with sparse data). Fortunately the test data is already in that format so we can simply perform the query:

[5]:
```
%%time
neighbors = index.query(news_test.data)
```

```
CPU times: user 1min 25s, sys: 1.5 s, total: 1min 26s
Wall time: 1min 26s
```

And that's it! Everything works essentially transparently with sparse data – it is just slower. Still, slower is a lot better than not working at all.

[6]:
```
neighbors
```

[6]:
```
(array([[ 1635,  4487,  9220, ..., 10071,  9572,  1793],
        [ 8648,   567,  2123, ...,   783,  6031,  9275],
        [ 7345,  4852,  4674, ...,  1679,  7518,  4228],
        ...,
        [ 6137, 10518,  6469, ...,  6937, 11083,  6164],
        [ 2926,  2011,  1679, ...,  7229,  1635, 11270],
        [ 3215,  3665,  4899, ..., 10810,  9907,  9311]], dtype=int32),
 array([[0.34316891, 0.34799916, 0.35074973, ..., 0.35952544, 0.36456954,
         0.36472946],
        [0.25881797, 0.26937056, 0.28056568, ..., 0.29210907, 0.29795945,
         0.29900843],
        [0.41124642, 0.42213005, 0.4426493 , ..., 0.46922857, 0.4724912 ,
         0.47429329],
        ...,
        [0.21533132, 0.22482073, 0.24046886, ..., 0.26193857, 0.26805884,
         0.26866162],
        [0.19485909, 0.19515198, 0.19891578, ..., 0.20851403, 0.21159202,
         0.21265447],
        [0.43528455, 0.43638128, 0.4378109 , ..., 0.45176154, 0.452402  ,
         0.45243692]]))
```

One final caveat is that custom distance metrics for sparse data need to be able to work with sparse data and thus have a different function signature. In practice this is really something you only want to try if you are familiar with working with sparse data structures. If that's the case then you can look through `pynndescent.sparse.py` for examples of many common distance functions and it will quickly become clear what is required.

2.4 Working with Scikit-learn pipelines

Nearest neighbor search is a fundamental building block of many machine learning algorithms, including in supervised learning with kNN-classifiers and kNN-regressors, and unsupervised learning with manifold learning, and clustering. It would be useful to be able to bring the speed of PyNNDescent's approximate nearest neighbor search to bear on these problems without having to re-implement everything from scratch. Fortunately Scikit-learn has done most of the work for us with their KNeighborsTransformer, which provides a means to insert nearest neighbor computations into sklearn pipelines, and feed the results to many of their models that make use of nearest neighbor computations. It is worth reading through the documentation they have, because we are going to use PyNNDescent as a drop in replacement.

To make this as simple as possible PyNNDescent implements a class `PyNNDescentTransformer` that acts as a `KNeighborsTransformer` and can be dropped into all the same pipelines. Let's see an example of this working ...

[1]:
```
from sklearn.manifold import Isomap, TSNE
from sklearn.neighbors import KNeighborsTransformer
from pynndescent import PyNNDescentTransformer
from sklearn.pipeline import make_pipeline
from sklearn.datasets import fetch_openml
from sklearn.utils import shuffle

import seaborn as sns
```

As usual we will need some data to play with. In this case let's use a random subsample of MNIST digits.

[2]:
```
def load_mnist(n_samples):
    """Load MNIST, shuffle the data, and return only n_samples."""
    mnist = fetch_openml("mnist_784")
    X, y = shuffle(mnist.data, mnist.target, random_state=2)
    return X[:n_samples] / 255, y[:n_samples]
```

[3]:
```
data, target = load_mnist(10000)
```

Now we need to make a pipeline that feeds the nearest neighbor results into a downstream task. To demonstrate how this can work we'll try manifold learning. First we will try out Isomap and then t-SNE. In both cases we can provide a "precomputed" distance matrix, and if it is a sparse matrix (as output by `KNeighborsTransformer`) then any entry not explicitly provided as a non-zero element of the matrix will be ignored (or treated as an effectively infinite distance). To make the whole thing work we simple make an sklearn pipeline (and could easily include pre-processing steps such as categorical encoding, or data scaling and standardisation as earlier steps if we wished) that first uses the `KNeighborsTransformer` to process the raw data into a nearest neighbor graph, and then passes that on to either `Isomap` or `TSNE`. For comparison we'll drop in a `PyNNDescentTransformer` instead and see how that effects the results.

[4]:
```
sklearn_isomap = make_pipeline(
    KNeighborsTransformer(n_neighbors=15),
    Isomap(metric='precomputed')
)
pynnd_isomap = make_pipeline(
    PyNNDescentTransformer(n_neighbors=15),
    Isomap(metric='precomputed')
)
sklearn_tsne = make_pipeline(
    KNeighborsTransformer(n_neighbors=92),
    TSNE(metric='precomputed', random_state=42)
)
pynnd_tsne = make_pipeline(
    PyNNDescentTransformer(n_neighbors=92, early_termination_value=0.05),
    TSNE(metric='precomputed', random_state=42)
)
```

First let's try Isomap. The algorithm first constructs a k-nearest-neighbor graph (which our transformers will handle in the pipeline), then measures distances between points as path lengths in that graph. Finally it performs an eigendecomposition of the resulting distance matrix. We can do much to speed up the later two steps, which are still non-trivial, but hopefully we can get some speedup by substituting in the approximate nearest neighbor computation.

[5]:
```
%%time
sklearn_iso_map = sklearn_isomap.fit_transform(data)
```

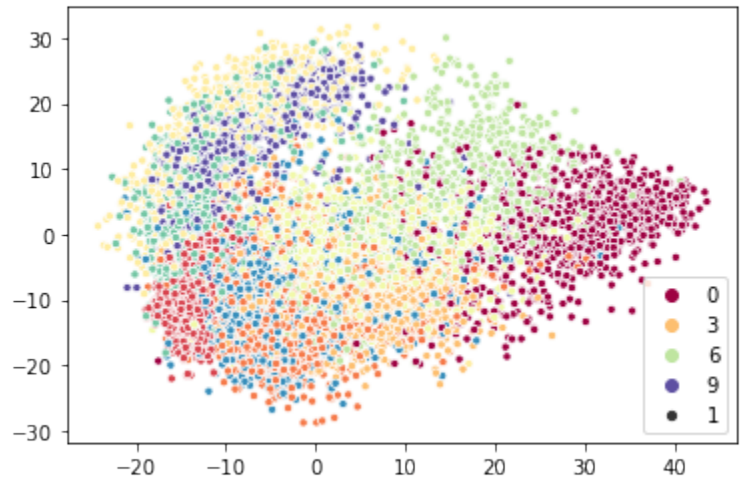
```
CPU times: user 2min 42s, sys: 1.76 s, total: 2min 44s
Wall time: 2min 43s
```

[6]:
```
%%time
pynnd_iso_map = pynnd_isomap.fit_transform(data)
```

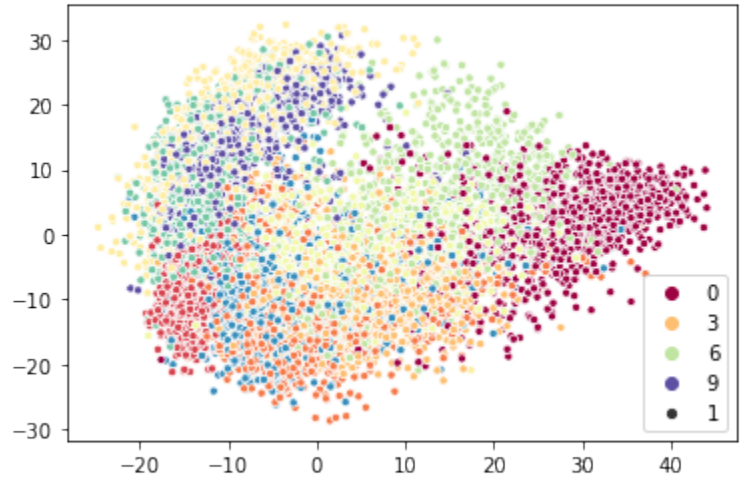
```
CPU times: user 1min 14s, sys: 2.85 s, total: 1min 17s
Wall time: 1min 17s
```

A two-times speedup is not bad, especially since we only accelerated one component of the full algorithm. It is quite good considering it was simply a matter of dropping a different class into a pipeline. More importantly as we scale to larger amounts of data the nearest neighbor search comes to dominate the over algorithm run-time, so we can expect to only get better speedups for more data. We can plot the results to ensure we are getting qualitatively the same thing.

[7]:
```
sns.scatterplot(x=sklearn_iso_map.T[0], y=sklearn_iso_map.T[1], hue=target, palette=
↪"Spectral", size=1);
```



[8]:
```
sns.scatterplot(x=pynnd_iso_map.T[0], y=pynnd_iso_map.T[1], hue=target, palette=
↪"Spectral", size=1);
```



Now let's try t-SNE. This algorithm requires nearest neighbors as a first step, and then the second major part, in terms of computation time, is the optimization of a layout of a modified k-neighbor graph. We can hope for some improvement in the first part, which usually accounts for around half the overall run-time for small data (and comes to consume a majority of the run-time for large datasets).

[9]:
```
%%time
sklearn_tsne_map = sklearn_tsne.fit_transform(data)
```

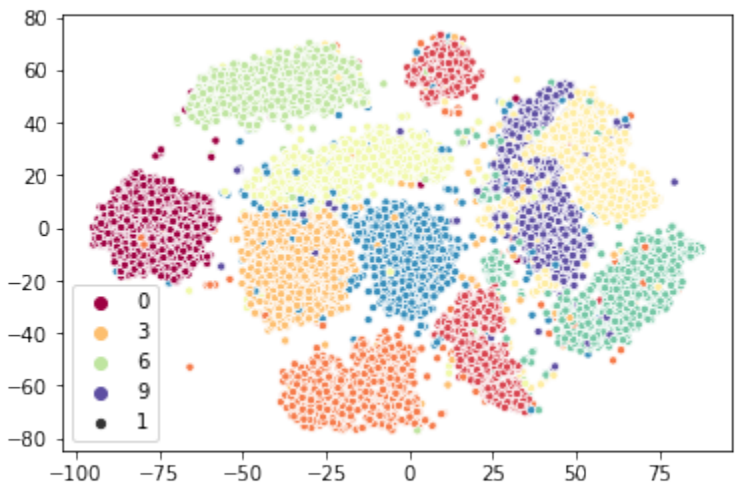
```
CPU times: user 7min 1s, sys: 3.72 s, total: 7min 5s
Wall time: 3min 8s
```

[10]:
```
%%time
pynnd_tsne_map = pynnd_tsne.fit_transform(data)
```

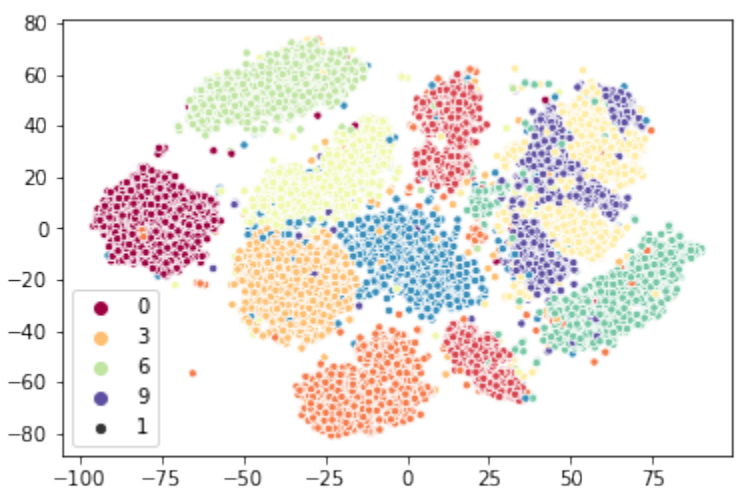
```
CPU times: user 5min 32s, sys: 3.7 s, total: 5min 35s
Wall time: 1min 29s
```

Again we have an approximate two-times speedup. Again this was achieved by simply substituting a different class into the pipeline (although in the case we tweaked the `early_termination_value` so it would stop *sooner*). Again we can look at the qualitative results and see that we are getting something very similar.

[11]:
```
sns.scatterplot(x=sklearn_tsne_map.T[0], y=sklearn_tsne_map.T[1], hue=target, palette=
↪"Spectral", size=1);
```



[12]:
```
sns.scatterplot(x=pynnd_tsne_map.T[0], y=pynnd_tsne_map.T[1], hue=target, palette=
↪"Spectral", size=1);
```



So the results, in both cases, look pretty good, and we did get a good speed-up. A question remains – how fast was

the nearest neighbor component, and how accurate was it? We can write a simple function to measure the neighbor accuracy: compute the average percentage intersection in the neighbor sets of each sample point. Then let's just run the transformers and compare the times as well as computing the actual percentage accuracy.

[13]:
```
import numba
import numpy as np

@numba.njit()
def arr_intersect(ar1, ar2):
    aux = np.sort(np.concatenate((ar1, ar2)))
    return aux[:-1][aux[:-1] == aux[1:]]

@numba.njit()
def neighbor_accuracy_numba(n1_indptr, n1_indices, n2_indptr, n2_indices):
    result = 0.0
    for i in range(n1_indptr.shape[0] - 1):
        indices1 = n1_indices[n1_indptr[i]:n1_indptr[i+1]]
        indices2 = n2_indices[n2_indptr[i]:n2_indptr[i+1]]
        n_correct = np.float64(arr_intersect(indices1, indices2).shape[0])
        result += n_correct / indices1.shape[0]
    return result / (n1_indptr.shape[0] - 1)

def neighbor_accuracy(neighbors1, neighbors2):
    return neighbor_accuracy_numba(
        neighbors1.indptr, neighbors1.indices, neighbors2.indptr, neighbors2.indices
    )
```

[14]:
```
%time true_neighbors = KNeighborsTransformer(n_neighbors=15).fit_transform(data)
%time pynnd_neighbors = PyNNDescentTransformer(n_neighbors=15).fit_transform(data)

print(f"Neighbor accuracy is {neighbor_accuracy(true_neighbors, pynnd_neighbors) *␣
↪100.0}%")
```

```
CPU times: user 1min 43s, sys: 177 ms, total: 1min 43s
Wall time: 1min 44s
CPU times: user 2.67 s, sys: 49 ms, total: 2.72 s
Wall time: 920 ms
Neighbor accuracy is 99.140625%
```

So for the Isomap case we went from taking over one and half minutes down to less then a second. While doing so we still achieved over 99% accuracy in the nearest neighbors. This seems like a good tradeoff.

By constrast t-SNE requires a much larger number of neighbors (approximately three times the desired perplexity value, which defaults to 30 in sklearn's implementation). This is a little more of a challenge so we might expect it to take longer.

[15]:
```
%time true_neighbors = KNeighborsTransformer(n_neighbors=92).fit_transform(data)
%time pynnd_neighbors = PyNNDescentTransformer(n_neighbors=92, early_termination_
↪value=0.05).fit_transform(data)

print(f"Neighbor accuracy is {neighbor_accuracy(true_neighbors, pynnd_neighbors) *␣
↪100.0}%")
```

```
CPU times: user 1min 42s, sys: 136 ms, total: 1min 42s
Wall time: 1min 43s
CPU times: user 25.2 s, sys: 545 ms, total: 25.7 s
Wall time: 7.37 s
Neighbor accuracy is 99.96612903225784%
```

We see that the `KNeighborsTransformer` takes the same amount of time for this – this is because it is making the choice, given the dataset size and dimensionality, to compute nearest neighbors by effectively computing the full distance matrix. That means regardless of how many neighbors we ask for it will take a largely constant amount of time.

In constrast we see that the `PyNNDescentTransformer` is having to work harder, taking almost eight seconds (still a lot better than one and half minutes!). The increased `early_termination_value` (the default is 0.001) stops the computation early, but even with this we are still getting over 99.9% accuracy! Certainly the minute and a half saved in computation time at this step is worth the drop of 0.033% accuracy in nearest neighbors. And these differences in computation time will only increase as dataset sizes get larger.

2.5 How PyNNDescent works

PyNNDescent uses neighbor graph based searching to efficiently find good candidates for the nearest neighbors of query points from a large training set. While the basic ideas turn out to be quite simple, the layers of refinements and tricks used to get the highest degree of efficiency complicate things a lot. With that in mind we will begin with the simple naive approach and gradually work out further refinements and details as we go. The core concept, upon which most everything else is based, is using a nearest neighbor graph to perform a search for approximate nearest neighbors.

2.5.1 Searching using a nearest neighbor graph

Suppose we are given a nearest neighbor graph (let's ignore, for now, how we can have obtained such a thing without the ability to find nearest neighbors already). By a graph, I simply mean a network – a set of nodes connected by edges. By a nearest neighbor graph I mean that the graph has been derived from data existing in some sort of space for which we have a quantitative notion of similarity, or dissimilarity, or distance. In particular the graph is derived by making each data point a node and connecting that data point, via edges, to the k most similar, least dissimilar, or closest other points. We would like to use this graph as an index structure for searching to find the nearest neighbors (points in the graph) of a new query point (potentially not a point in the graph) with respect to that notion of similarity of distance. How can we do this? We can do it via a kind of pruned breadth-first search of the graph, always keeping only the k best points we've found so far. The algorithm, therefore, looks something like this:

1. Choose a starting node in the graph (potentially randomly) as a candidate node
2. Look at all nodes connected by an edge to the best untried candidate node in the graph
3. Add all these nodes to our potential candidate pool
4. Sort the candidate pool by closeness / similarity to the query point
5. Truncate the pool to k best (as in closest to the query) candidates
6. Return to step 2, unless we have already tried all the candidates in the pool

We can see this play out visually on a small graph. For this case we will let $k = 2$ and have a 2-neighbor graph constructed by points in a plane. The query point will be drawn as an orange 'x'. We then search, exactly as described, and find ourselves steadily traversing the graph towards the query point.

While that did indeed find the points closest to the query point, it was hardly more efficient than a linear search of all the points – there are only two points we didn't try! This is ultimately about having such a small graph: while it is good for showing the algorithm clearly, the algorithm doesn't really scale down that well to this case. If we want to see it providing benefits we'll need a larger graph. That's not hard to generate, so let's see a larger example.

Now, with about as bad a start as we could hope to get, the algorithm efficiently traverses across the graph and find the neighbors of the orange query point, and this time we see that only a small fraction of the total number of points in the graph had to be tried. If we had been a little luckier with our start point we could have done significantly better again. We can scale up to even larger examples:

Again we had about as bad a starting point as possible, but again it very efficiently traverses the graph, steadily improving as it goes, until the algorithm eventually find the nearest neighbors of the orange query point. Notably the algorithm will continue to scale well as the number of points increases, and even as the dimension of the data increases – as long as the data itself has a low enough intrisic dimension.

There are, of course, several things that can be done to refine this algorithm to be a little more efficient, and more flexible at the same time. They include such things as keeping a hashtable of visited nodes to ensure you don't compute distances twice, and keeping a slightly larger priority queue of candidate nodes to expand – anything with (1+:math:*varepsilon*) of the kth best candidate found so far will suffice. This latter point allows us greater flexibility in search, allowing a degree of backtracking around local "optima" in the search. It also provides a parameter which we can tune to get a trade-off between the speed and accuracy that we desire: smaller epsilon gives a faster search, but less accuracy, while a large epsilon can give good accuracy but at the cost of a slower search.

All of this presumed, however, that we had a nearest neighbor graph to start with – the thing that we were using as our search index. The question remains: how are we to build that, since we could potentially just use the nearest neighbor search technique used to build the graph (which presumably has to be cheap) instead of the graph search described here. The answer, as it happens, is that we are going to use the graph search technique to build the graph itself – pulling ourselves up by our own bootstraps so to speak.

2.5.2 NNDescent for building neighbor graphs

A good way to get to grips with the NNDescent algorithm for constructing approximate k-neighbor graphs is to think of it in terms of the search algorithm we've already described. Suppose we had a decent, but not exact, k-neighbor graph. We could use that graph as our search index – it won't be great, and our search will be particularly approximate and often get stuck in local minima because of that, but it will still work a lot of the time. So, given that graph we could do a nearest neighbor search, using that graph as the index, for each and every point in the dataset. Because of the nature of the search we could end up finding new neighbors for a node that are closer than the ones in our not-so-good graph. We could use that information to update our graph – make it better by using these newly found closer neighbors. Having improved the graph, we can run the search again for each and every point in the dataset. Since the graph is better we will do an even better job of finding neighbors than before and potentially find some new neighbors that will let us update the graph again.

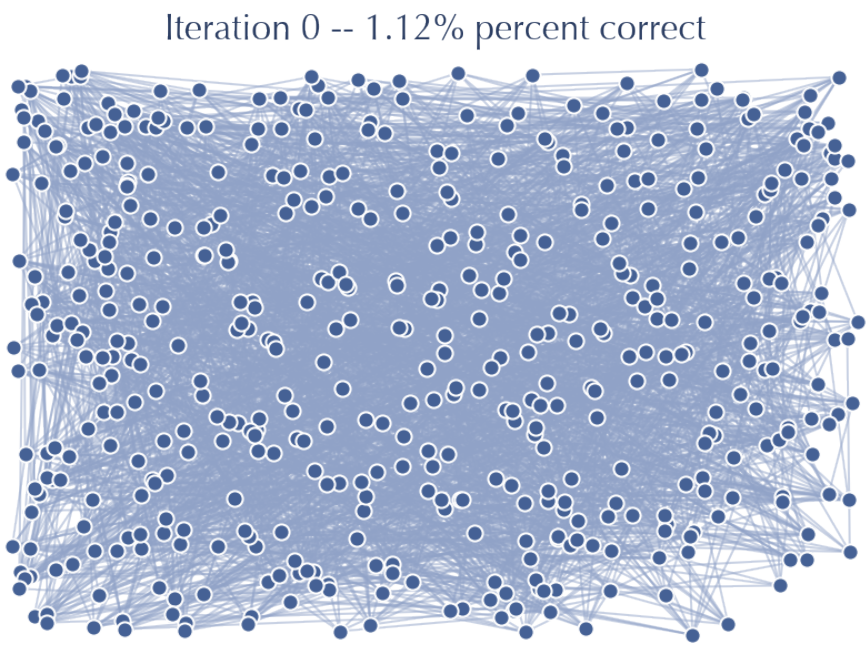
And that's actually the core of the algorithm. Start with a bad graph, use the nearest neighbor search on that graph to get better neighbors, and make a better graph. Rinse, repeat, and eventually we will end up with a good graph. In fact, the better the graph is, the faster and more accurate the search will run. So each iteration will go faster than the last, and we will steadily converge toward the true k-neighbor graph.

There are a couple things to take note of at this point: a better graph results in a better search, so making a better graph sooner will help the search; finding a good initial candidate node for the search was generically a problem (our animations all had particularly bad initial candidates), but we can always use the point itself as the initial candidate since it is guaranteed to be in the graph. So, given a good initial candidate (the node itself), and a desire to update the graph as soon as possible, it may not be worth running a full search to exhaustion. Instead we could run the search just far enough to potentially find some new neighbors that we haven't seen before. The first round of search, starting from the node itself, will only find the neighbors we already have. The second round, however, will step out to neighbors of neighbors (friends of friends if you will), and potentially find some nodes that are closer neighbors than the ones currently in the graph. That would be enough information to improve the graph – since we would have found new better neighbors. So the algorithm would now run:

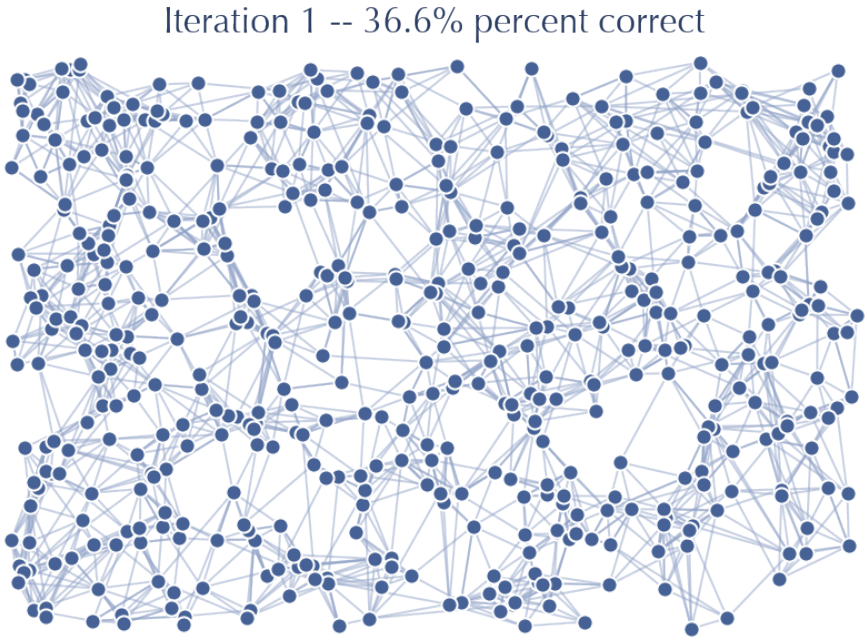
1. Start with a random graph (connect each node to k random nodes)
2. For each node:
3. Measure the distance from the node to the neighbors of its neighbors
4. If any are closer then update the graph accordingly, and keep only the k closest
5. If any updates were made to the graph then go back to step 2, otherwise stop

We can see this play out on some example data (the same data as the medium sized search example):

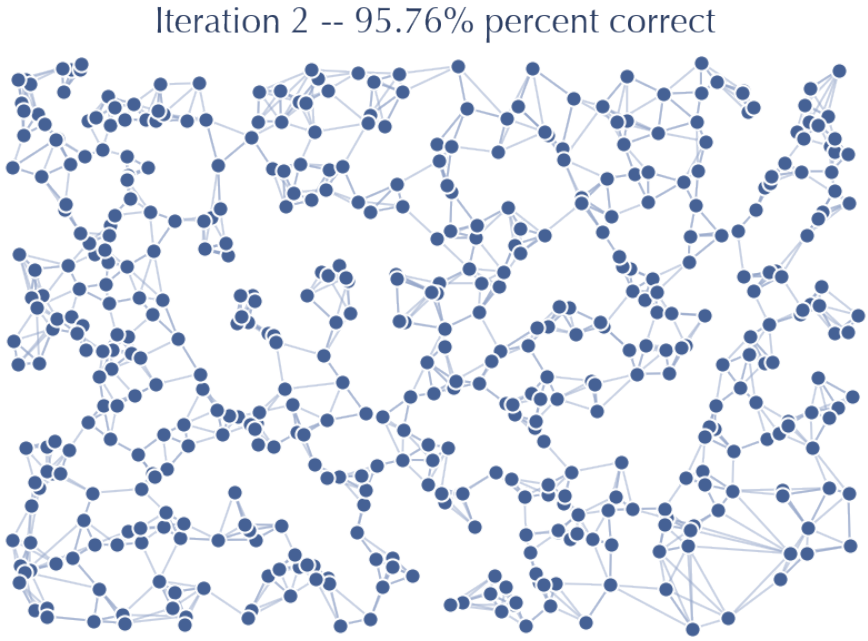
You can see the graph starts out looking nothing like a k-neighbors graph – just a mess of edges all over with no structure. Quickly, however, we update the graph, and the better the graph gets the better our updates go. After not very long at all the algorithm is simply refining an already very good graph with few, if any, updates at all actually occurring. We can see this most clearly by looking at things in terms of each round of iteration of the algorithm – consider the graph at the point where we've updated neighbors for every node and we are at step 3, considering whether to loop back to step 2 or not. At each such iteration we can both look at the state of the graph and measure how close we are to the true k-neighbor graph. We start with a random graph:



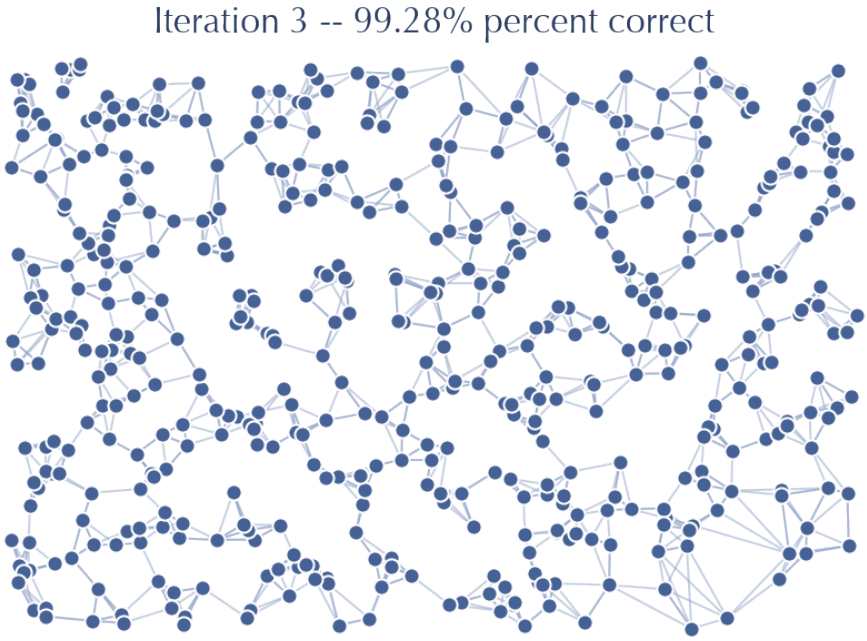
After we have touched each node once and updated the neighbors based on the friends of friends we get the following:



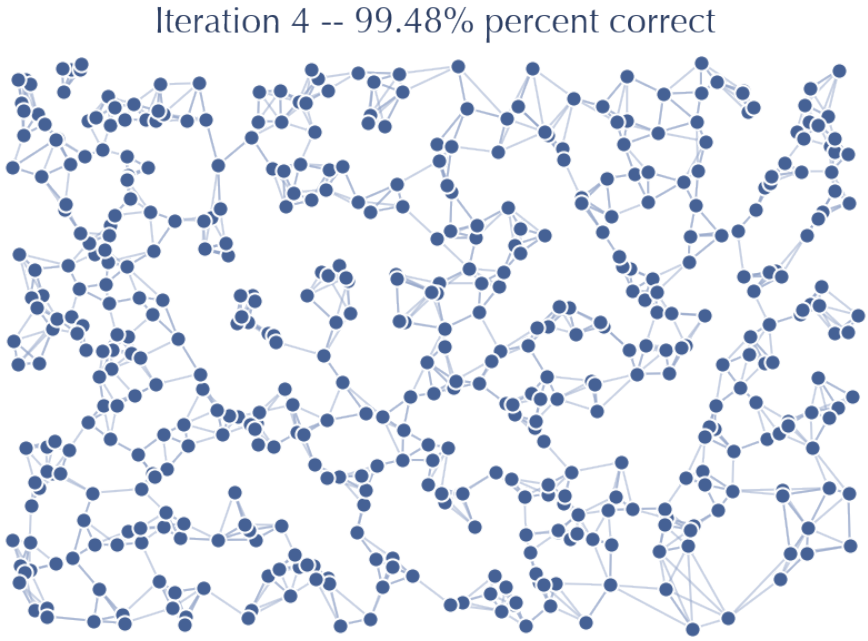
Already, after just one pass through, the graph looks a whole lot less random. And, indeed, we have already improved from the sort of accuracy of neighbors we would expect at random, to a graph that has 30% of the neighbors correct for each node (on average). That is a much better graph for doing searches on than the purely random graph, so we would expect to be able to do a lot better when we do another iteration performing updates based on friends of friends in this new much better graph.



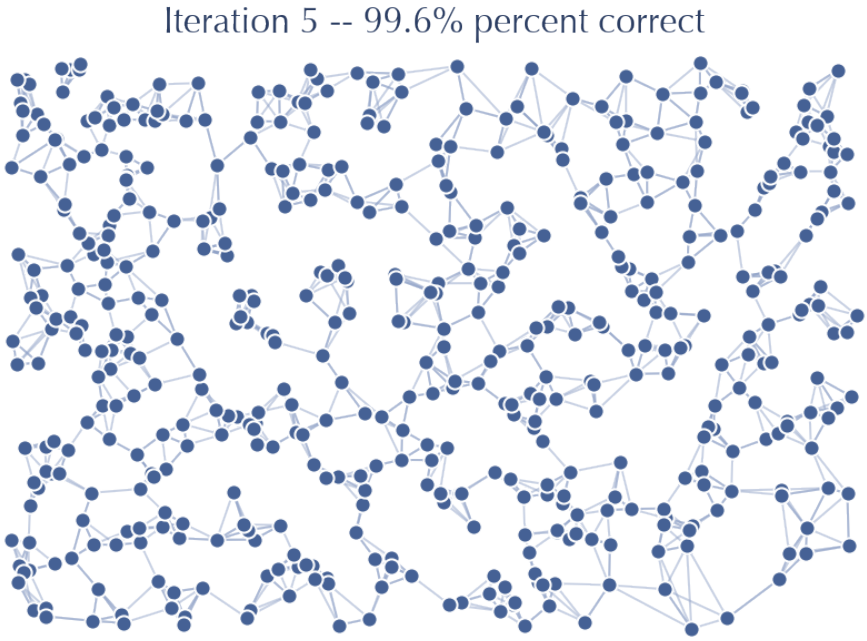
So now after just two passes we already have a graph that is 95% accurate! A search on this graph will be quite accurate, especially since we are guaranteed to be starting with a good candidate (the query itself). Running another pass should get a near perfect result.



The accuracy improved nicely, but you should note that visually there is very little difference between this graph and the previous graph. Very few edges actually changed in this iteration – the number of new neighbors, closer than any found so far, is very small. That suggests that from here on out we will get significantly diminishing returns.



Sure enough there is little improvement this time – the last few nodes for which we do not yet have the correct k neighbors are hard to get right with a search from this graph, and so it will be challenging to make significant further improvements. Another iteration will improve things further:

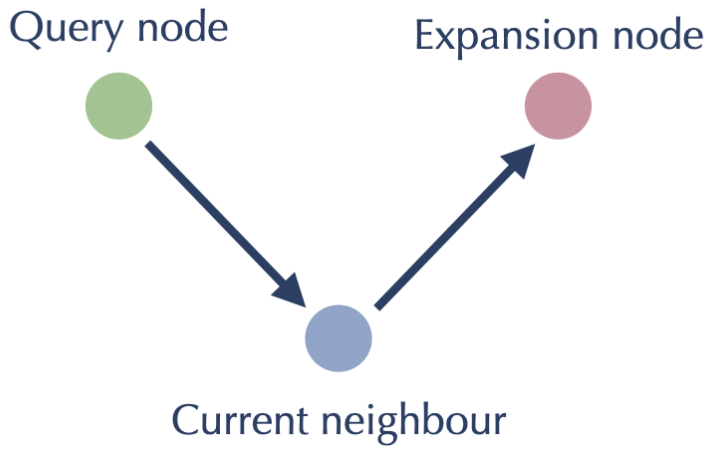


but this is where it stops. Further iterations fail to find any further improvements for the graph. We can't get to 100% accuracy in this case. Still, 99.6% accuracy is pretty good, and will be good enough enough to be an effective search index for new previously unseen queries.

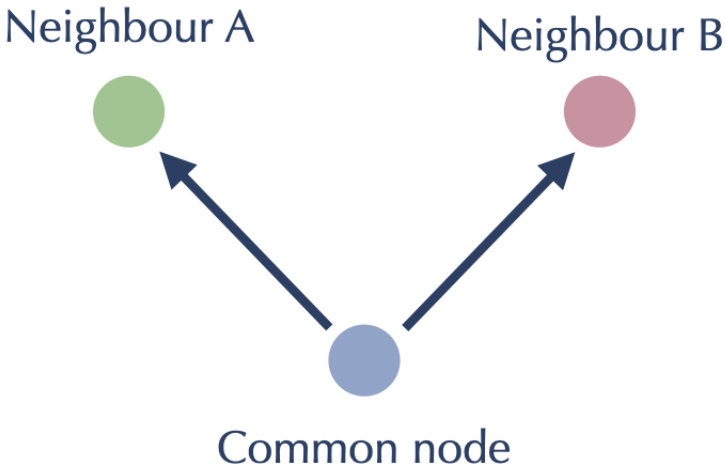
This provides us with the basic algorithm, which is already surprisingly efficient and, importantly, doesn't rely on specific properties of the dissimilarity (such as requiring the triangle inequality) beyond the requirement that "friends of friends" should be a good source of candidate neighbors. There are a number of computational considerations that can dramatically improve practical runtime performance however. Let's look at a few of the major ones, particularly since they involve looking at the algorithm in a different way and result in code that looks very dissimilar from what has been described so far, despite the fact that this is essentially what it is implementing.

The first major consideration is that the search should take place on an undirected graph. That means that when looking at neighbors of a node we need to consider not only the k neighbors that the node has edges to, but also all the other nodes that have our chosen node as one of their k neighbors, often called the "reverse nearest neighbors". While it is easy to keep track of the top k neighbors for each node, it is much harder to keep track of the reverse nearest neighbors for each node, and updating that information as the graph changes becomes challenging. For this reason (among others) it becomes beneficial to compute the combined set of neighbors and reverse nearest neighbors for each node based on the graph once at the start of an iteration and use that information while updating the graph in the background. In a sense we are holding the search graph fixed for the iteration, and then applying all the updates at the end.

The second major consideration is a useful inversion of how we look at the problem. Ultimately for each node we need to look at the neighbors of its neighbors, which involves two hops through the graph. From the point of view of the green node we might view it as looking something like this:



Since all our work will be on length two paths like this however, we can potentially flip our point of view around and center ourselves at the blue node. From the blue node's point of view the goal here is have the red and the green node consider adding each other to each others k nearest neighbor lists. This flipped viewpoint saves us from some doubled work (we just need to get green and red to talk to each other at once, rather than traversing the path once from green, and then much later the other way from red), but it also turns what was a graph walking algorithm into one that is entirely local to each node: having found the sets of neighbors and reverse neighbors for every node as described above, we simply need to do an all-pairs comparison between the nodes of each such set, having each possible pairing act as the red and green nodes attached to a common node:



Putting these two considerations together we arrive at an approach that first generates a set of "neighbors" (both neighbors and reverse neighbors) for each node, then generates graph updates by doing an all-pairs distance computation on each set. After the first step of computing the reverse neighbors the all-pairs computations can all run independently in parallel, saving off the updates from each neighbor set which can then be sorted and applied to update the graph. While this is executing the same notional algorithm, viewing it this way allows for easy parallelism options, and for further efficiency tweaks: keep track of which "neighbors" are new to a set and only worry about distance computations to nodes new to the set; restrict the number of elements in the set for any iteration; exit early when the number of updates falls below a (proportional) threshold; etc.

In the end we arrive at a parallelizable algorithm that can compute an approximate k-neighbor graph of a dataset extremely efficiently. Better still it does this, much as our search algorithm did, with few constraints: we require a

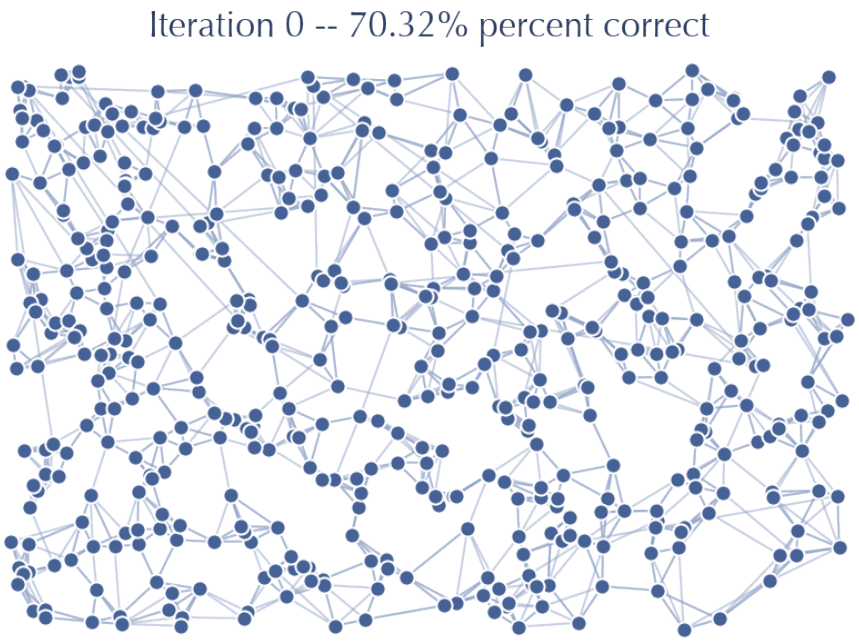
dissimilarity that has a "friend-of-a-friend" principle, and that the *intrinsic* dimension (as opposed to the apparent or ambient dimension) of the dataset is not too large. Is there any way we can speed this up at all?

2.5.3 Random projection trees for initialization

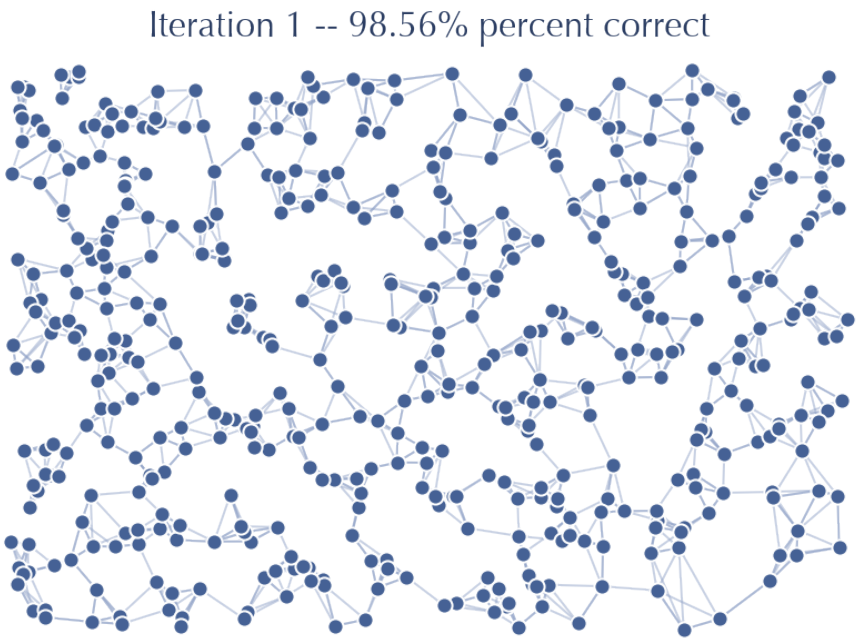
Random projection trees are an efficient approach to approximate nearest neighbor search for spatial data (i.e. data that has a vector space representation). The idea is simple enough: start by choosing a random hyperplane that splits the data in two (one can arrange a split by choosing two random points and taking the hyperplane to be the orthogonal hyperplane halfway between them – in either a euclidean or angular sense, depending on the distance metric); for each half choose a different random hyperplane to split that subset of the data; keep repeating this until each subset is at most some chosen size (usually called the "leaf size" of the tree). This simplistic approach manages to recursively partition up the data into conveniently sized buckets. The random hyperplanes serve two purposes: first by being at random orientations the resulting trees adapt well to high dimensional data (unlike other tree approaches such as kd-trees); second, by being randomized we can easily build many such trees that are all different. The problem with random projection trees for nearest neighbor search is that for a given query the nearest neighbors may be on the wrong side of one of those random splits, and if we just search the bucket of points in the leaf the query falls in we'll miss those nearest neighbors. The solution is to have many trees. Since each tree is random, each tree will make different mistakes in terms of the random splits. In aggregate, by searching through a whole forest of random projection trees, the odds of finding the true nearest neighbors becomes pretty good.

So why not just use this for approximate nearest neighbor search? To get adequate accuracy one either needs to use a very large leaf size (and brute force search through all the points in a leaf), or use a *lot* of trees. In practice this is often much slower than the graph based search described above. Still, trees are *very* cheap to build, and the search can be *very* fast for a single tree, so can we use them somehow? Recall that we started our graph off with a random initialization, but that the better our graph was the faster each iteration ran and the better the iteration was at finding good neighbors. Could we use random projection trees to find a good starting point for the graph? It doesn't have to be very good, just better than *random*. Better than random is not that hard to beat. So we can build a very small forest of random projection trees (small by the standard of the number of trees you would use if you wanted reasonable accuracy queries), and initialize our graph with it. How do we use the trees if we aren't using them on new query points? Each leaf node of the tree is a bucket for which we can do an all-pairs distance computation, and each node in the graph can get an initial k edges based on the results of the bucket it was in. This can all be done very quickly and entirely in parallel for a small number of trees. How well does it work?

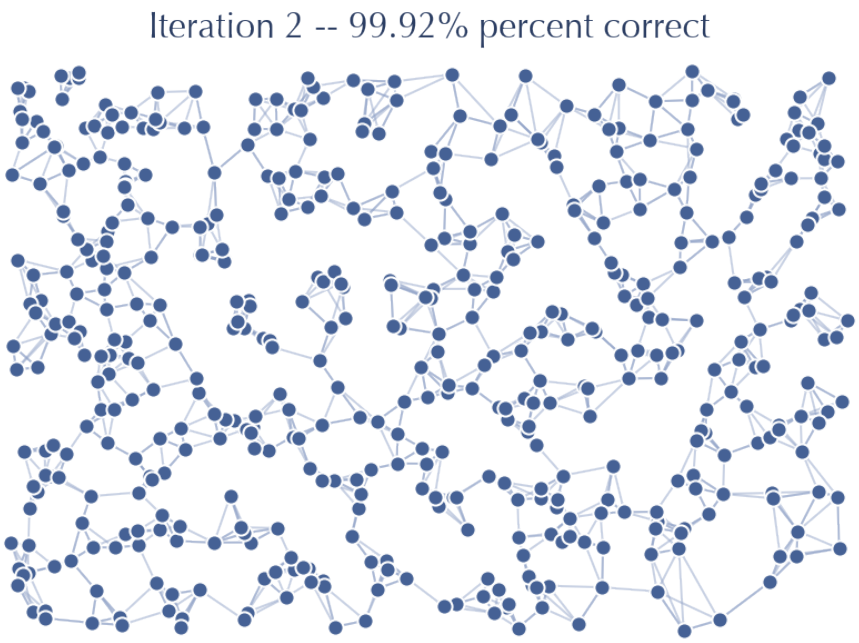
Let's got back to our example data we built the graph for in the last section. This time we'll use *just one* random projection tree to initialize things, and then proceed with nearest neighbor descent as before.



As you can see, even with only a single tree we get off to a good start; in this case 70% correct. For more difficult data in higher dimensions things won't go this well, but as we observed above, once you have a somewhat decent graph NNDescent proceeds very efficiently:



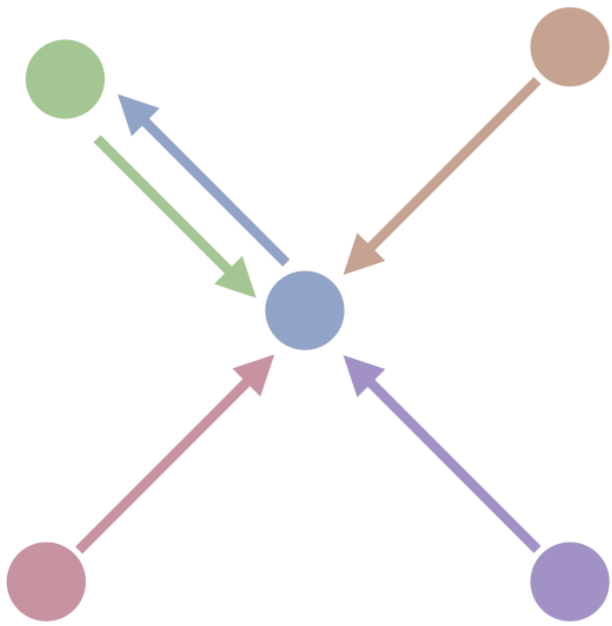
Since we started from a better graph after only one iteration of NNDescent we have already got to 98% accuracy. It takes only one more iteration of work:



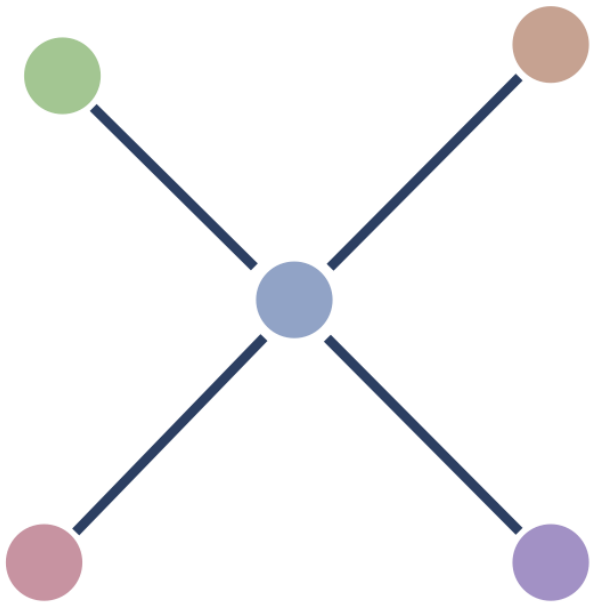
To get to a point where there are no further gains to be had. The real key is that the initialization can save us early, and very expensive, iterations. Saving a few of the early and most expensive iterations is a major gain. For this reason PyNNDescent uses a small forest of random projection trees to get a small head-start on getting a good k-neighbor graph for NNDescent to perform its iterative improvements on.

2.5.4 Refining the graph for faster searching

Now that we have means to build a k-neighbor graph efficiently, is there any way we can fine tune the graph to make our nearest neighbor searches run faster? One thing that slows down the search is adding *lots* of new nodes to the candidate pool, all of which then need to have the distance to a query point computed. If nodes have many edges attached then we end up with very large candidate pools, and search can slow significantly. While at first it seems a simple case of noting that since we built a k-neighbor graph each node has k edges, the difference is that search is conducted on an *undirected* graph, and this means that a node has not only edges to all the nodes that it thinks are its neighbors, but also edges to all the nodes that think it is their neighbor. For example, consider this 1-neighbor graph with five nodes – each node has a single edge to its nearest neighbor.



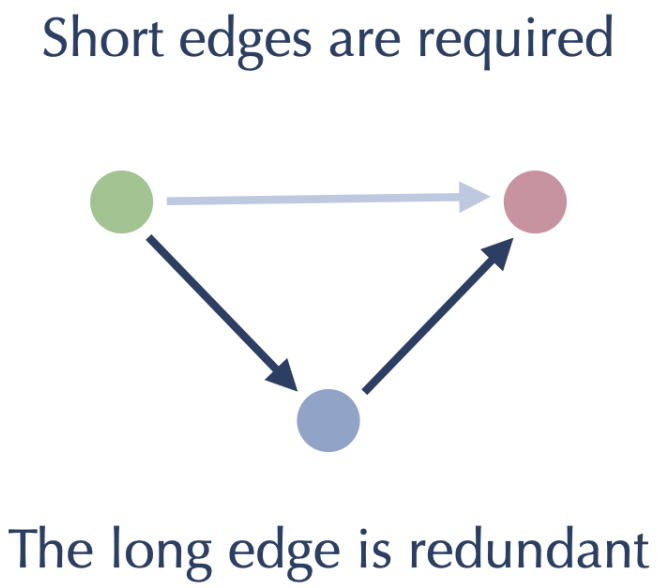
While each node has a single neighbor and thus a single out-going edge, if we convert to an *undirected* graph we end up with this situation:



We see that the central node has four edges, despite this being a 1-neighbor graph. This can be surprisingly common, especially for high dimensional data, and the resulting undirected version of the k-neighbor graph can have nodes with orders of magnitudes more than the k edges you might expect. Pruning edges out of the graph to reduce the number of edges any one node might have attached is going to be the best way to speed up search. The catch, of course, is that removing edges can make the search less accurate. Our goal for refining the graph, then, is to remove as many edges as possible while doing as little harm as possible to the accuracy of any searches performed on the resulting graph.

One immediate possibility simply use the directed graph and throw out all the reverse nearest neighbor edges that are induced in the undirected case. While this can be effective, it does reduce the accuracy of searches; many of those reverse nearest neighbor edges are, in fact, very helpful. We could, instead, take the directed graph plus some amount of reverse neighbor edges. In cases where a node has few of these edges we can keep them all, but for the cases of nodes, such as the central node in the example above, that have relatively many reverse neighbor edges attached, we can simply take the top few of them. This is, in fact, what PyNNDescent does using the `pruning_degree_multiplier` parameter. PyNNDescent will keep the top `pruning_degree_multiplier * n_neighbors` edges for each node, dropping any edges beyond that number.

Before we perform that kind of pruning, however, we can see if there are some other edges that we might also be able to drop with little impact on search quality. The answer is yes – specifically we can drop the long edges of triangles. What do we mean by this? Consider the case where we have a triangle formed of edges like this:

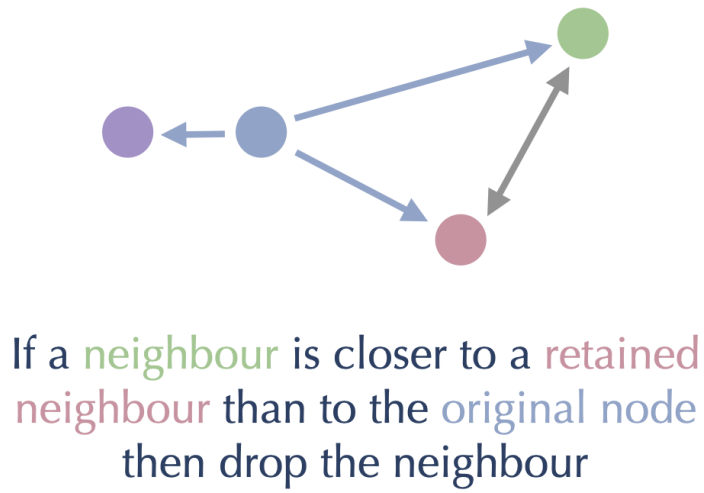


The short edges are needed to find near neighbors of each of the nodes to which they are attached. However if we follow our search algorithm we will not need the long edge since the search will traverse the two short edges and find the same result, just with one extra step / iteration. The expectation that this might slow down search doesn't play out in practice – the gains made from removing edges dominates the small extra costs of occasionally having to take extra steps in a search. We can put together an approach to removing these long edges of triangles as follows:

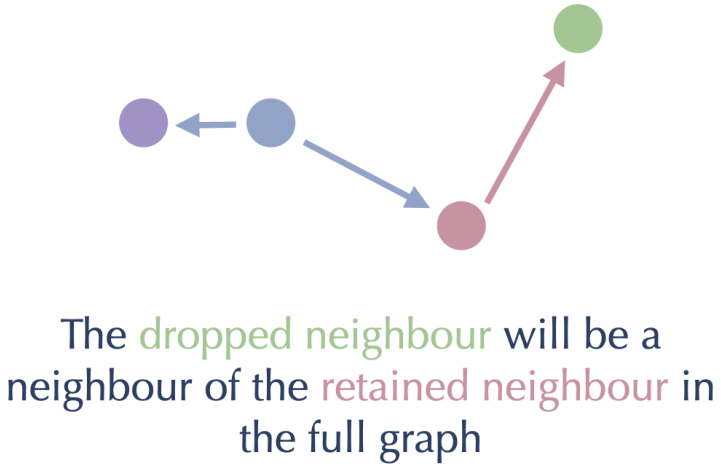
- For each node in the graph:

1. Retain the first nearest neighbor
2. For each other neighbor:
    1. If the neighbor is closer to the node than it is to any currently retained neighbor, then retain it as well
    2. If the neighbor is closer to a retained neighbor than the node, then drop it

This process ensures we keep the edges to the closest neighbors, and steadily prune out the long edges of any triangles that may occur. We can see this pictorially as follows: consider a node (in blue) with several neighbors, and suppose we have retained all the closer nodes and are considering whether to retain the green node.



Then, since green is closer to the red node (which we already retained) than it is to the blue node we are working on, we should drop the edge from blue to green. We can get to the green node from the blue node by traversing through the red node:



By preprocessing the graph in this way, first removing the long edges of triangles and then pruning back to a maximum fixed number of edges per node, we can significantly improve the search performance on the graph. Preprocessing like this is a non-trivial amount of computational work, so by default PyNNDescent doesn't perform that work unless specifically asked (a search query is taken to be such a request). This is because the user may be simply interested in the raw k-neighbor graph of the training data, and not in efficient querying later.

2.5.5 Putting it all together

We now have all the pieces in place. We can construct a k-neighbor graph using NNDescent, initialized using a small random projection tree forest. We can then refine the resulting graph to optimize it to search for near neighbors of new query points. For the search we can select a starting point using (the best) random projection tree, ensuring we start relatively near to neighbors of the query. We then the apply the graph search algorithm we started with to find the nearest neighbors of the query point. And that is how PyNNDescent works.

2.6 PyNNDescent Performance

How fast is PyNNDescent for approximate nearest neighbor search? How does it compare with other approximate nearest neighbor search algorithms and implementations? To answer these kinds of questions we'll make use of the ann-benchmarks suite of tools for benchmarking approximate nearest neighbor (ANN) search algorithms. The suite provides a wide array of datasets to benchmark on, and supports a wide array of ANN search libraries. Since the runtime of these benchmarks is quite large we'll be presenting results obtained earlier, and only for a selection of datasets and for the main state-of-the-art implementations. This page thus reflects the performance at a given point in time, and on a specific choice of benchmarking hardware. Implementations may (and likely will) improve, and different hardware will likely result in somewhat different performance characteristics amongst the implementations benchmarked here.

We chose the following implementations of ANN search based on their strong performance in ANN search benchmarks in general:

- Annoy (a tree based algorithm for comparison)
- HNSW from FAISS, Facebooks ANN library

- HNSW from nmslib, the reference implementation of the algorithm
- HNSW from hnswlib, a small spinoff library from nmslib
- ONNG from NGT, a more recent algorithm and implementaton with impressive performance
- PyNNDescent version 0.5

Not all the algorithms ran entirely successfully on all the datasets; where an algorithm gave spurious or unrepresentative results we have left it off rather the given benchmark.

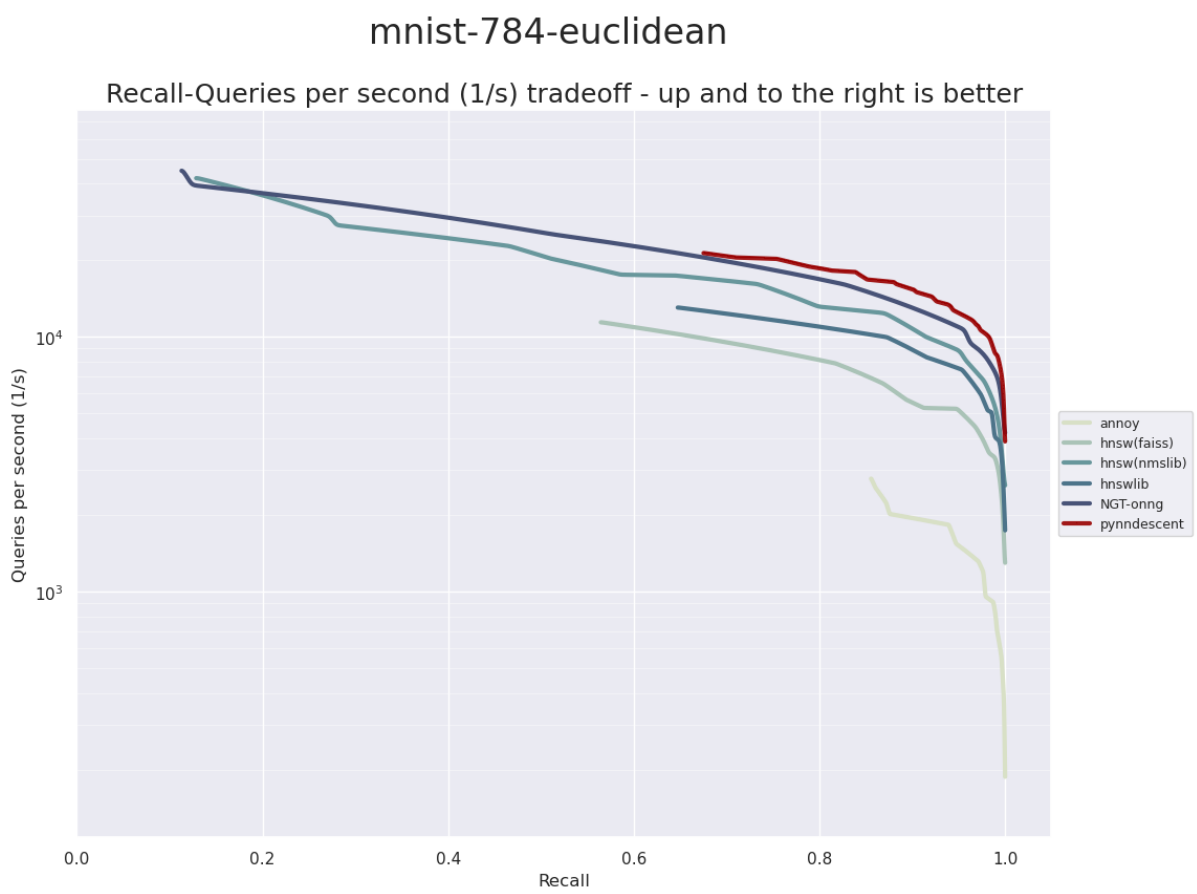
The ann-benchmark suite is designed to look at the trade-off in performance between search accuracy and search speed (or other performance statistic, such as index creation time, or index size). Since this is a trade-off that can often be tuned by appropriately adjusting parameters ann-benchmarks handles this by running a predefined (for each algorithm or implementation) range of parameters. It then finds the pareto frontier for the optimal speed / accuracy trade-off and presents this as a curve. The various implementations can then be compared in terms of the pareto frontier curves. The default choices of measure for ann-benchmarks puts recall (effective search accuracy) along the x-axis and queries-per-second (search speed) on the y-axis. Thus curves that are further up and / or more to the right are providing better speed and / or more accuracy.

To get a good overview of the relative performance characteristics of the different implementations we'll look at the speed / accuracy trade-off curves for a variety of datasets. This is because the dataset size, dimensionality, distribution and metric can all have non-trivial impacts on performance in various ways, and results for one dataset are not necessarily representative of how things will look for a different dataset. We will introduce each dataset in turn, and then look at the performance curves. To start with we'll consider datasets which use Euclidean distance.

2.6.1 Euclidean distance

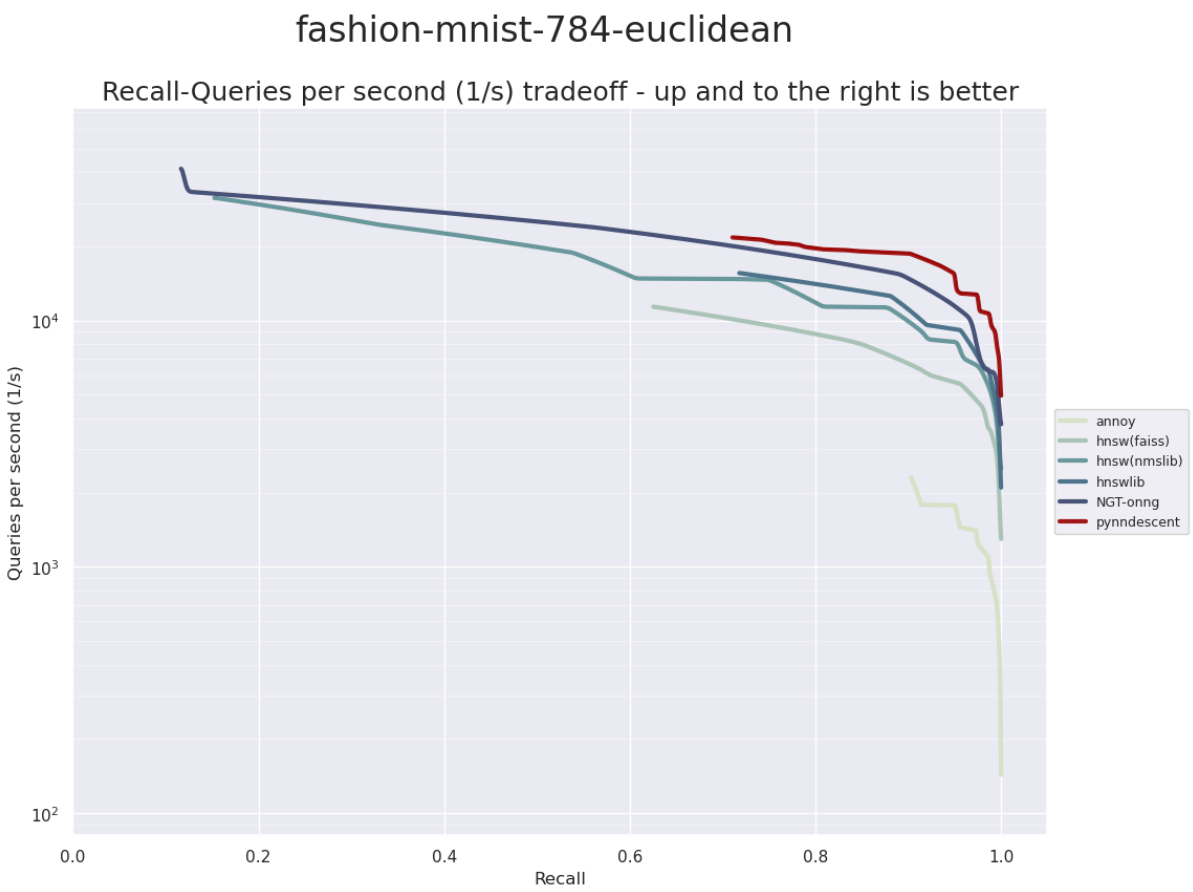
Euclidean distance is the usual notion of distance that we are familiar with in everyday life, just extended to arbitrary dimensions (instead of only two or three). It is defined as $d(\bar{x}, \bar{y}) = \sum_i (x_i - y_i)^2$ for vectors $\bar{x} = (x_1, x_2, \ldots, x_D)$ and $\bar{y} = (y_1, y_2, \ldots, y_D)$. It is widely used as a distance measure, but can have difficulties with high dimensional data in some cases.

The first dataset we will consider that uses Euclidean distance is the MNIST dataset. MNIST consists of grayscale images of handwritten digits (from 0 to 9). Each digit image is 28 by 28 pixels, which is usually unravelled into a single vectors of 784 dimensions. In total there are 70,000 images in the dataset, and ann-benchmarks uses the usual split into 60,000 training samples and 10,000 test samples. The ANN index is built on the training set, and then the test set is used as the query data for benchmarking.



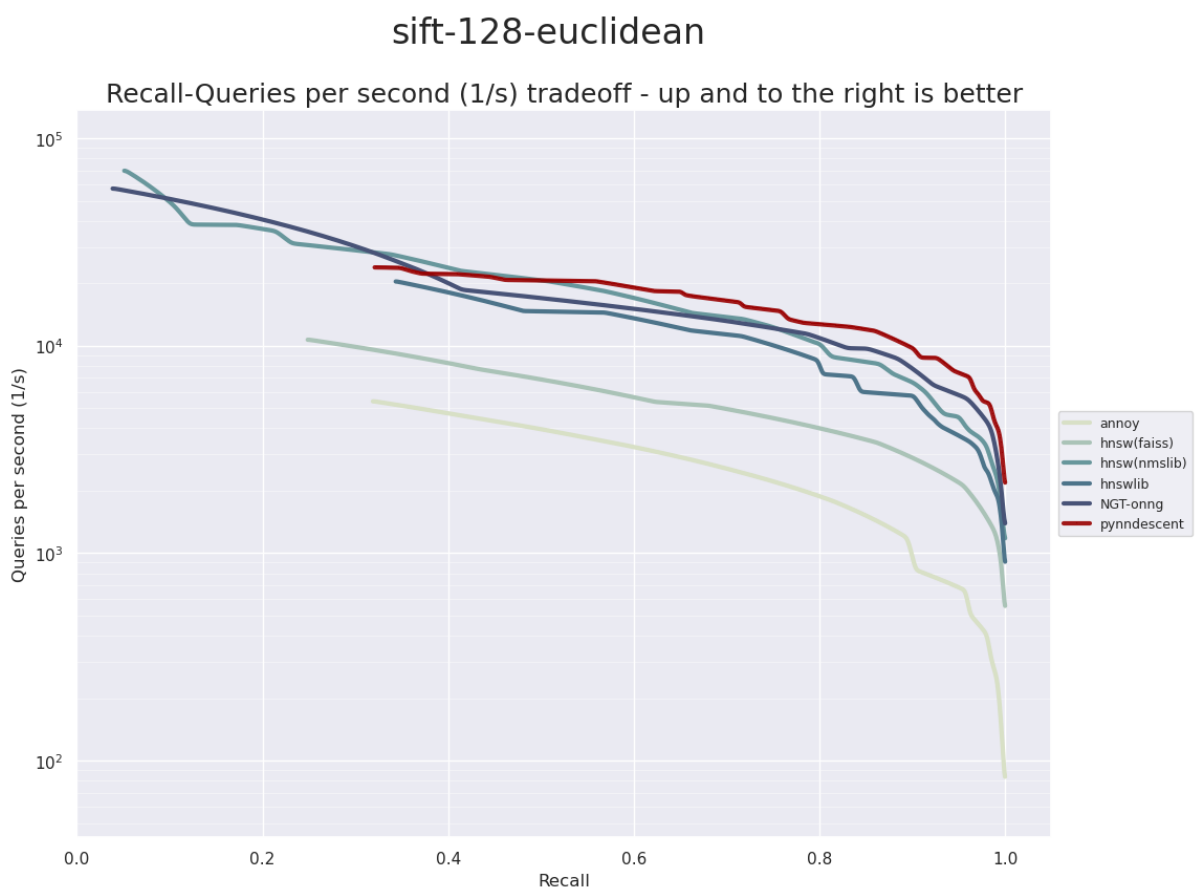
Remember that up and to the right is better. Also note that the y axis (queries per second) is plotted in *log scale* so each major grid step represents an order of magnitude performance difference. We can see that PyNNDescent performs very well here, outpacing the other ANN libraries in the high accuracy range. It is worth noting, however, that for lower accuracy queries it finishes essentially on par with ONNG, and unlike ONNG and nmslib's HNSW implementation, it does not extend to very high performance but low accuracy queries. If speed is absolutely paramount, and you only need to be in the vaguely right ballpark for accuracy then PyNNDescent may not be the right choice here.

Next up for dataset is Fashion-MNIST. This was a dataset designed to be a drop in replacement for MNIST, but meant to be more challenging for machine learning tasks. Instead of grayscale images of digits it is grayscale images of fashion items (dresses, shirts, pants, boots, sandas, handbags, etc.). Just like MNIST each image is 28 by 28 pixels resulting in 784-dimensional vectors. Also just like MNIST there are 70,000 total images, split into 60,000 training images and 10,000 test images.



Again we see a very similar result (although this should not entirely be a surprise given the similarity of the dataset in terms of the number of samples and dimensionality). PyNNDescent performs very well in the high accuracy regime, but does not scale to the very high performance but low accuracy ranges that ONNG and nmslib's HNSW manage. It is also worth noting the clear difference between the various graph based search algorithms and the tree based Annoy – while Annoy is a very impressive ANN search implementation it compares poorly to the graph based search techniques on these datasets.

Next up is the SIFT dataset. SIFT stands for Scale-Invariant Feature Transform and is a technique from compute vision for generating feature vectors from images. For ann-benchmarks this means that there exist some large databases of SIFT features from image datasets which can be used to test nearest neighbor search. In particular the SIFT dataset in ann-benchmarks is a dataset of one million SIFT vectors where each vector is 128-dimensional. This provides a good contrast to the earlier datasets which had relatively high dimensionality, but not an especially large number of samples. For ann-benchmarks the dataset is split into 990,000 training samples, and 10,000 test samples for querying with.



Again we see that PyNNDescent performs very well. This time, however, with the more challenging search problem presented by a training set this large, it does produce some lower accuracy searches and in those cases both ONG and nmslib's HNSW outperform it. It's also worth noting that in this lower dimensional dataset Annoy performs better, comparatively, than the previous datasets. Still, over the Euclidean distance datasets tested here PyNNDescent remains a clear winner for high accuracy queries. Let's move on to the angular distance based datasets.

2.6.2 Angular distance

Angular based distances measure the similarity of two vectors in terms of the angle they span – the greater the angle the larger the distance between the vectors. Thus two vectors of different length can be viewed as being very close as long as they are pointing in the same direction. Another way of looking at this is to imagine that the data is being projected onto a high dimensional sphere (by intersecting a ray in the vectors direction with a unit sphere), and distances are measured in terms of arcs around the sphere.

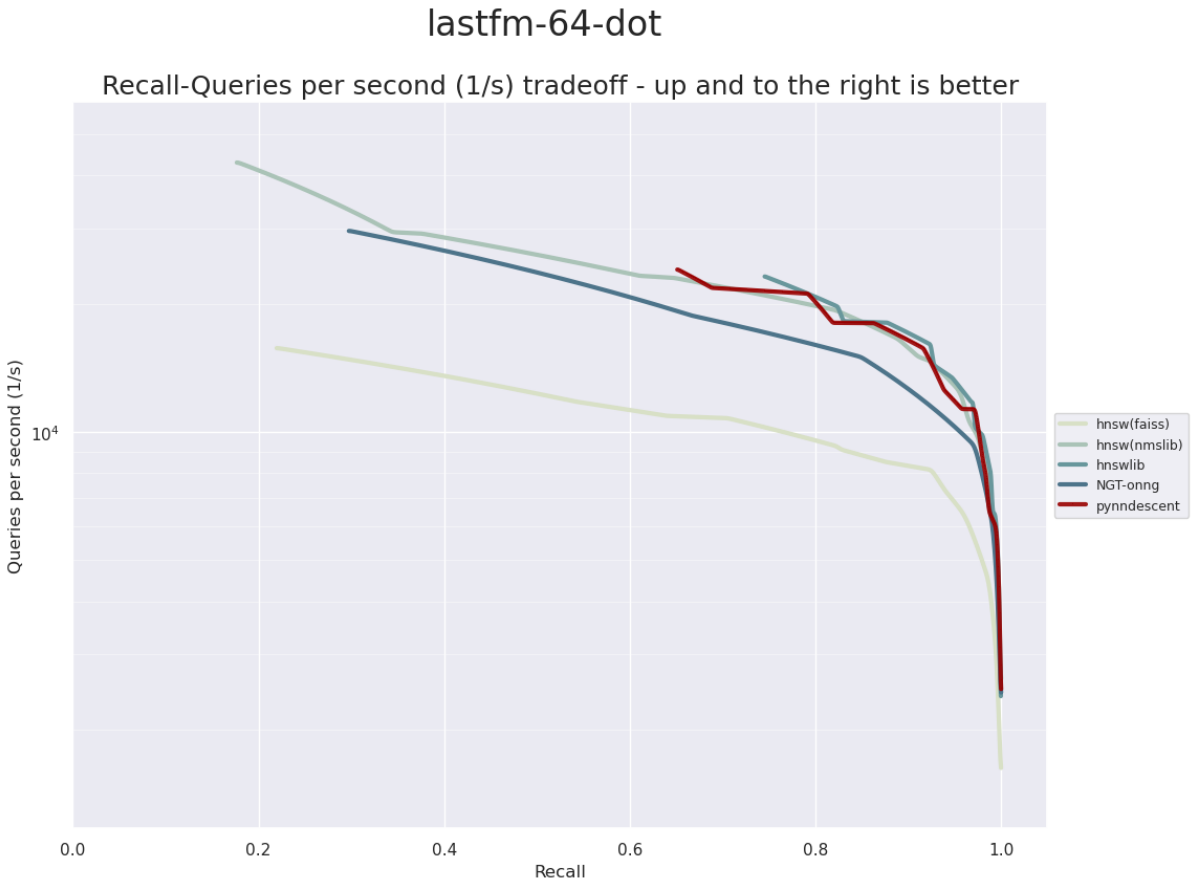
In practice the most commonly used angular distance is cosine distance, defined as

$$d(\bar{x}, \bar{y}) = 1 - \sum_i \frac{x_i y_i}{\|\bar{x}\|_2 \|\bar{y}\|_2}$$

where $\|\bar{x}\|_2$ denotes the ℓ^2 norm of $\bar{x}$. To see why this is a measure of angular distance note that $\sum_i x_i y_i$ is the euclidean dot product of $\bar{x}$ and $\bar{y}$ and that the euclidean dot product formula gives $\bar{x} \cdot \bar{y} = \|x\|_2 \|y\|_2 \cos\theta$ where θ is the angle between the vectors.

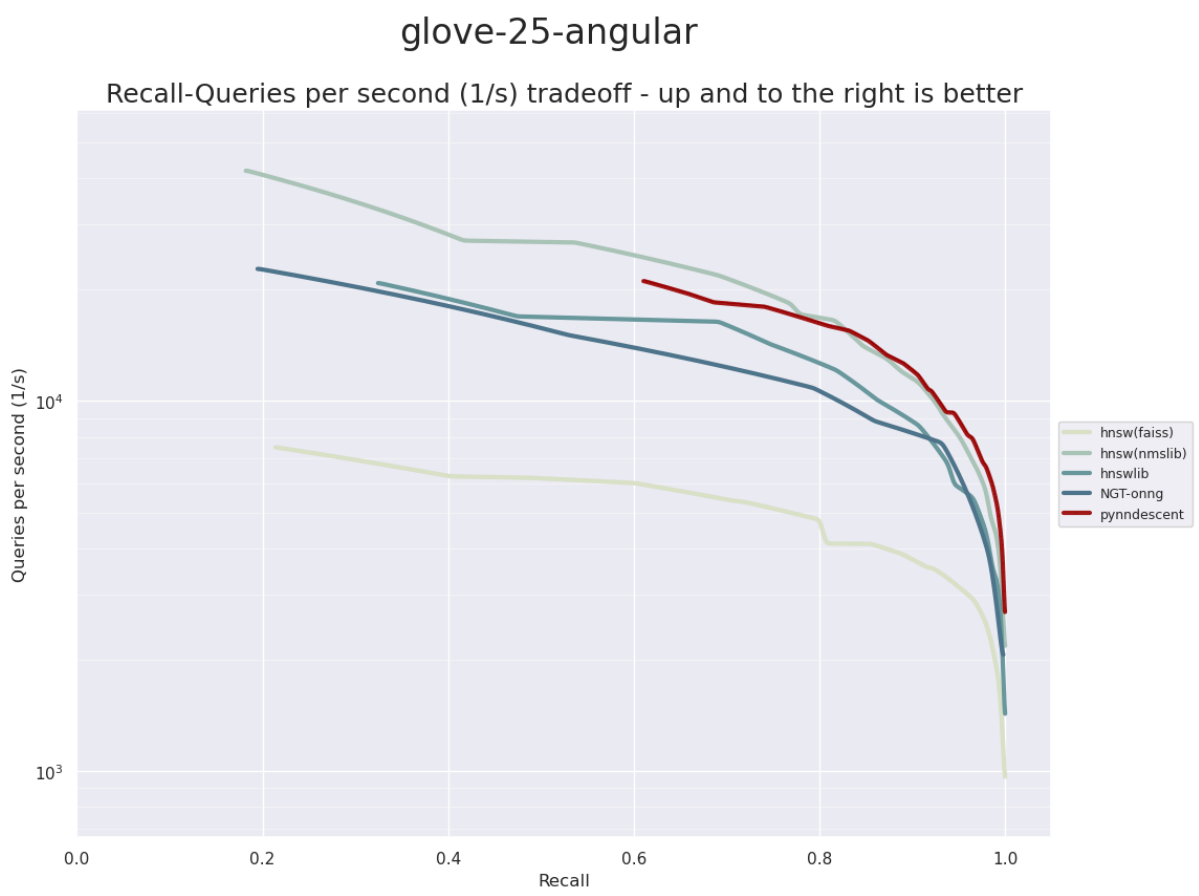
In the case where the vectors all have unit norm the cosine distance reduces to just one minus the dot product of the vectors – which is sometimes used as an angular distance measure. Indeed, that is the case for our first dataset, the LastFM dataset. This dataset is constructed of 64 factors in a recommendation system for the Last FM online music

service. It contains 292,385 training samples and 50,000 test samples. Compared to the other datasets explored so far this is considerably lower dimensional and the distance computation is simpler. Let's see what results we get.



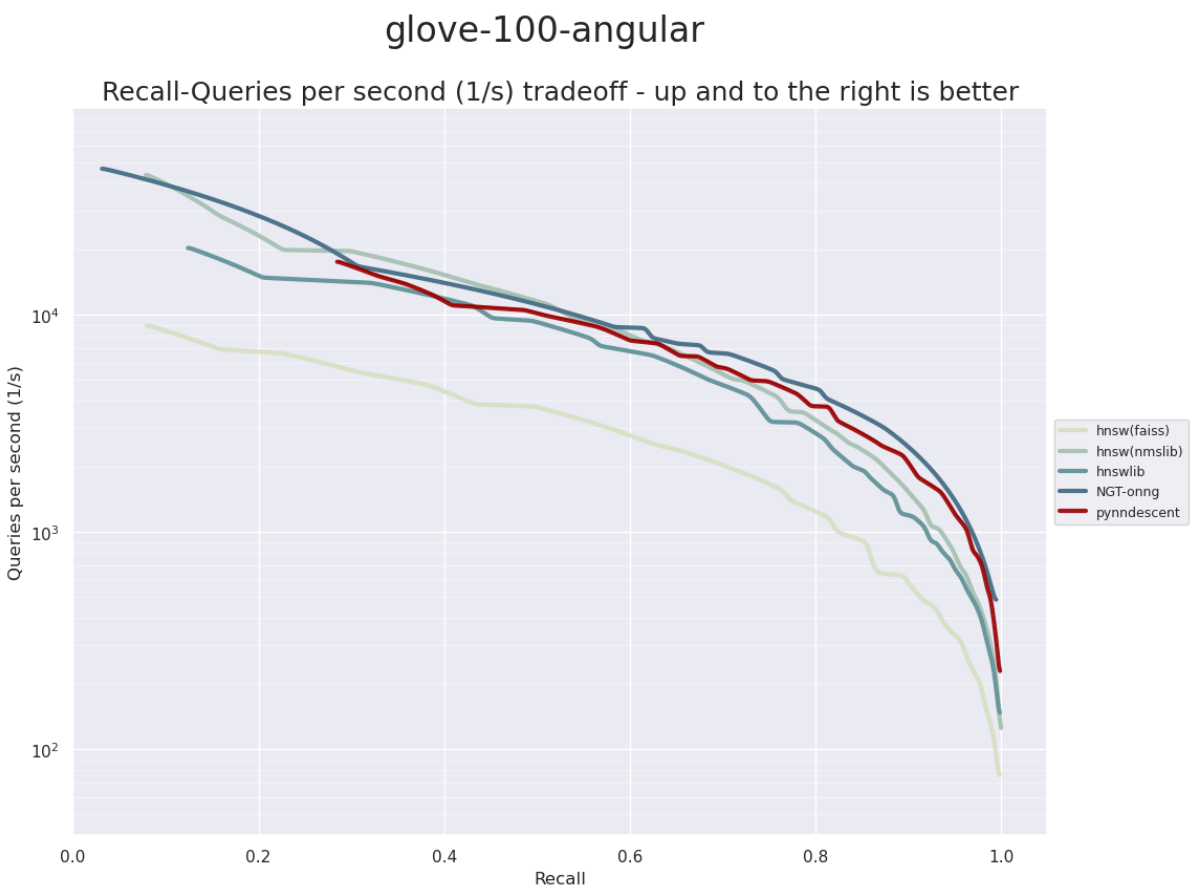
Here we see hnswlib and HNSW from nmslib performing extremely well – outpacing ONNG unlike we saw in the previous euclidean datasets. The HNSW implementation is FAISS is further behind. While PyNNDescent is not the fastest option on this dataset it is highly competitive with the two top performing HNSW implementations.

The next dataset is a GloVe dataset of word vectors. The GloVe datasets are generated from a word-word co-occurrence count matrix generated from vast collections of text. Each word that occurs (frequently enough) in the text will get a resulting vector, with the principle that words with similar meanings will be assigned vectors that are similar (in angular distance). The dimensionality of the generated vectors is an input to the GloVe algorithm. For the first of the the GloVe datasets we will be looking at the 25 dimensional vectors. Since GloVe vectors were trained useing a vast corpus there are over one million different words represented, and thus we have 1,183,514 training samples and 10,000 test samples to work with. This gives is a low dimensional but extremely large dataset to work with.



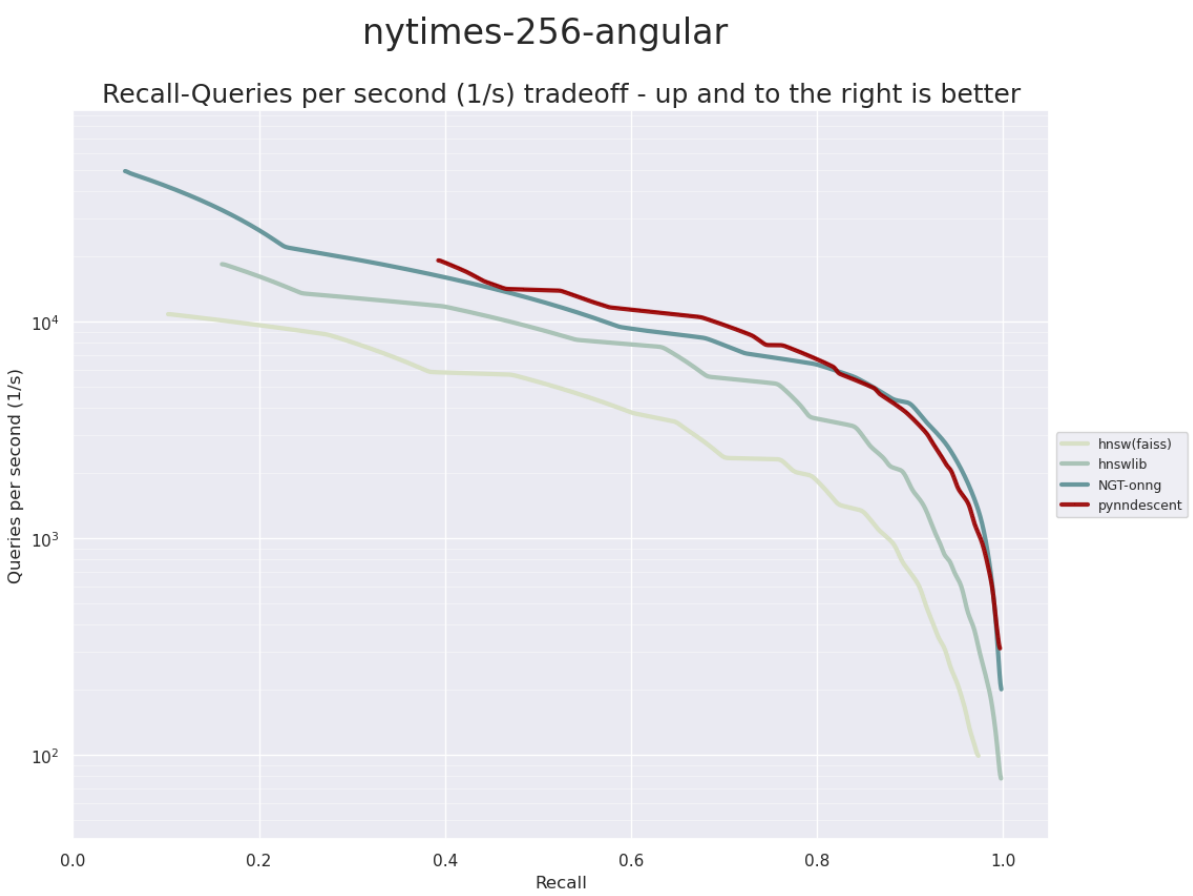
In this case PyNNDescent and hnswlib are the apparent winners – although PyNNDescent, similar to the earlier examples, performs less well once we get below abot 80% accuracy.

Next we'll move up to a higher dimensional version of GloVe vectors. These vectors were trained on the same underlying text dataset, so we have the same number of samples (both for train and test), but now we have 100 dimensional vectors. This makes the problem more challenging as the underlying distance computation is a little more expensive given the higher dimensionality.



This time it is ONNG that surges to the front of the pack. Relatively speaking PyNNDescent is not too far behind. This goes to show, however, how much performance can vary based on the exact nature of the dataset: while ONNG was a (relatively) poor performer on the 25-dimensional version of this data with hnswlib out in front, the roles are reversed for this 100-dimensional data.

The last dataset is the NY-Times dataset. This is data generated as dimension reduced (via PCA) TF-IDF vectors of New York Times articles. The resulting dataset has 290,000 training samples and 10,000 test samples in 256 dimensions. This is quite a challenging dataset, and all the algorithms have significantly lower query-per-second performance on this data.



Here we see that PyNNDescent and ONNG are the best performing implementations, particularly at the higher accuracy range (ONNG has a slight edge on PyNNDescent here).

This concludes our examination of performance for now. Having examined performance for many different datasets it is clear that the various algorithms and implementations vary in performance depending on the exact nature of the data. None the less, we hope that this has demonstrated that PyNNDescent has excellent performance characteristics across a wide variety of datasets, often performing better than many state-of-the-art implementations.

2.7 PyNNDescent API Guide

PyNNDescent has only two classes `NNDescent` and `PyNNDescentTransformer`.

2.7.1 PyNNDescent

class pynndescent.pynndescent_.**NNDescent**(*data*, *metric='euclidean'*, *metric_kwds=None*, *n_neighbors=30*, *n_trees=None*, *leaf_size=None*, *pruning_degree_multiplier=1.5*, *diversify_prob=1.0*, *n_search_trees=1*, *tree_init=True*, *init_graph=None*, *init_dist=None*, *random_state=None*, *low_memory=True*, *max_candidates=None*, *n_iters=None*, *delta=0.001*, *n_jobs=None*, *compressed=False*, *parallel_batch_queries=False*, *verbose=False*)

NNDescent for fast approximate nearest neighbor queries. NNDescent is very flexible and supports a wide variety of distances, including non-metric distances. NNDescent also scales well against high dimensional graph_data in many cases. This implementation provides a straightfoward interface, with access to some tuning parameters.

data: array of shape (n_samples, n_features) The training graph_data set to find nearest neighbors in.

metric: string or callable (optional, default='euclidean') The metric to use for computing nearest neighbors. If a callable is used it must be a numba njit compiled function. Supported metrics include:

- euclidean
- manhattan
- chebyshev
- minkowski
- canberra
- braycurtis
- mahalanobis
- wminkowski
- seuclidean
- cosine
- correlation
- haversine
- hamming
- jaccard
- dice
- russelrao
- kulsinski
- rogerstanimoto
- sokalmichener
- sokalsneath
- yule
- hellinger
- wasserstein-1d

Metrics that take arguments (such as minkowski, mahalanobis etc.) can have arguments passed via the metric_kwds dictionary. At this time care must be taken and dictionary elements must be ordered appropriately; this will hopefully be fixed in the future.

metric_kwds: dict (optional, default {}) Arguments to pass on to the metric, such as the `p` value for Minkowski distance.

n_neighbors: int (optional, default=30) The number of neighbors to use in k-neighbor graph graph_data structure used for fast approximate nearest neighbor search. Larger values will result in more accurate search results at the cost of computation time.

n_trees: int (optional, default=None) This implementation uses random projection forests for initializing the index build process. This parameter controls the number of trees in that forest. A larger number will result in more accurate neighbor computation at the cost of performance. The default of None means a value will be chosen based on the size of the graph_data.

leaf_size: int (optional, default=None) The maximum number of points in a leaf for the random projection trees. The default of None means a value will be chosen based on n_neighbors.

pruning_degree_multiplier: float (optional, default=1.5) How aggressively to prune the graph. Since the search graph is undirected (and thus includes nearest neighbors and reverse nearest neighbors) vertices can have very high degree – the graph will be pruned such that no vertex has degree greater than `pruning_degree_multiplier * n_neighbors`.

diversify_prob: float (optional, default=1.0) The search graph get "diversified" by removing potentially unnecessary edges. This controls the volume of edges removed. A value of 0.0 ensures that no edges get removed, and larger values result in significantly more aggressive edge removal. A value of 1.0 will prune all edges that it can.

n_search_trees: int (optional, default=1) The number of random projection trees to use in initializing searching or querying.

Deprecated since version 0.5.5.

tree_init: bool (optional, default=True) Whether to use random projection trees for initialization.

init_graph: np.ndarray (optional, default=None) 2D array of indices of candidate neighbours of the shape (data.shape[0], n_neighbours). If the j-th neighbour of the i-th instances is unknown, use init_graph[i, j] = -1

init_dist: np.ndarray (optional, default=None) 2D array with the same shape as init_graph, such that metric(data[i], data[init_graph[i, j]]) equals init_dist[i, j]

random_state: int, RandomState instance or None, optional (default: None) If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

algorithm: string (optional, default='standard') This implementation provides an alternative algorithm for construction of the k-neighbors graph used as a search index. The alternative algorithm can be fast for large `n_neighbors` values. The''alternative'' algorithm has been deprecated and is no longer available.

low_memory: boolean (optional, default=True) Whether to use a lower memory, but more computationally expensive approach to index construction.

max_candidates: int (optional, default=None) Internally each "self-join" keeps a maximum number of candidates (nearest neighbors and reverse nearest neighbors) to be considered. This value controls this aspect of the algorithm. Larger values will provide more accurate search results later, but potentially at non-negligible computation cost in building the index. Don't tweak this value unless you know what you're doing.

n_iters: int (optional, default=None) The maximum number of NN-descent iterations to perform. The NN-descent algorithm can abort early if limited progress is being made, so this only controls the worst case.

Don't tweak this value unless you know what you're doing. The default of None means a value will be chosen based on the size of the graph_data.

delta: float (optional, default=0.001) Controls the early abort due to limited progress. Larger values will result in earlier aborts, providing less accurate indexes, and less accurate searching. Don't tweak this value unless you know what you're doing.

n_jobs: int or None, optional (default=None) The number of parallel jobs to run for neighbors index construction. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors.

compressed: bool (optional, default=False) Whether to prune out data not needed for searching the index. This will result in a significantly smaller index, particularly useful for saving, but will remove information that might otherwise be useful.

parallel_batch_queries: bool (optional, default=False) Whether to use parallelism of batched queries. This can be useful for large batches of queries on multicore machines, but results in performance degradation for single queries, so is poor for streaming use.

verbose: bool (optional, default=False) Whether to print status graph_data during the computation.

query(*query_data*, *k=10*, *epsilon=0.1*)

Query the training graph_data for the k nearest neighbors

query_data: array-like, last dimension self.dim An array of points to query

k: integer (default = 10) The number of nearest neighbors to return

epsilon: float (optional, default=0.1) When searching for nearest neighbors of a query point this values controls the trade-off between accuracy and search cost. Larger values produce more accurate nearest neighbor results at larger computational cost for the search. Values should be in the range 0.0 to 0.5, but should probably not exceed 0.3 without good reason.

indices, distances: array (n_query_points, k), array (n_query_points, k) The first array, `indices`, provides the indices of the graph_data points in the training set that are the nearest neighbors of each query point. Thus `indices[i, j]` is the index into the training graph_data of the jth nearest neighbor of the ith query points.

Similarly `distances` provides the distances to the neighbors of the query points such that `distances[i, j]` is the distance from the ith query point to its jth nearest neighbor in the training graph_data.

update(*xs_fresh=None*, *xs_updated=None*, *updated_indices=None*)

Updates the index with a) fresh data (that is appended to the existing data), and b) data that was only updated (but should not be appended to the existing data).

Not applicable to sparse data yet.

xs_fresh: np.ndarray (optional, default=None) 2D array of the shape (n_fresh, dim) where dim is the dimension of the data from which we built self.

xs_updated: np.ndarray (optional, default=None) 2D array of the shape (n_updates, dim) where dim is the dimension of the data from which we built self.

updated_indices: array-like of size n_updates (optional, default=None) Something that is convertable to list of ints. If self is currently built from xs, then xs[update_indices[i]] will be replaced by xs_updated[i].

None

class pynndescent.pynndescent_.**PyNNDescentTransformer**(*n_neighbors=30, metric='euclidean', metric_kwds=None, n_trees=None, leaf_size=None, search_epsilon=0.1, pruning_degree_multiplier=1.5, diversify_prob=1.0, n_search_trees=1, tree_init=True, random_state=None, n_jobs=None, low_memory=True, max_candidates=None, n_iters=None, early_termination_value=0.001, parallel_batch_queries=False, verbose=False*)

PyNNDescentTransformer for fast approximate nearest neighbor transformer. It uses the NNDescent algorithm, and is thus very flexible and supports a wide variety of distances, including non-metric distances. NNDescent also scales well against high dimensional graph_data in many cases.

Transform X into a (weighted) graph of k nearest neighbors

The transformed graph_data is a sparse graph as returned by kneighbors_graph.

n_neighbors: int (optional, default=5) The number of neighbors to use in k-neighbor graph graph_data structure used for fast approximate nearest neighbor search. Larger values will result in more accurate search results at the cost of computation time.

metric: string or callable (optional, default='euclidean') The metric to use for computing nearest neighbors. If a callable is used it must be a numba njit compiled function. Supported metrics include:

- euclidean
- manhattan
- chebyshev
- minkowski
- canberra
- braycurtis
- mahalanobis
- wminkowski
- seuclidean
- cosine
- correlation
- haversine
- hamming
- jaccard
- dice

- russelrao
- kulsinski
- rogerstanimoto
- sokalmichener
- sokalsneath
- yule
- hellinger
- wasserstein-1d

Metrics that take arguments (such as minkowski, mahalanobis etc.) can have arguments passed via the metric_kwds dictionary. At this time care must be taken and dictionary elements must be ordered appropriately; this will hopefully be fixed in the future.

metric_kwds: dict (optional, default {}) Arguments to pass on to the metric, such as the `p` value for Minkowski distance.

n_trees: int (optional, default=None) This implementation uses random projection forests for initialization of searches. This parameter controls the number of trees in that forest. A larger number will result in more accurate neighbor computation at the cost of performance. The default of None means a value will be chosen based on the size of the graph_data.

leaf_size: int (optional, default=None) The maximum number of points in a leaf for the random projection trees. The default of None means a value will be chosen based on n_neighbors.

pruning_degree_multiplier: float (optional, default=1.5) How aggressively to prune the graph. Since the search graph is undirected (and thus includes nearest neighbors and reverse nearest neighbors) vertices can have very high degree – the graph will be pruned such that no vertex has degree greater than `pruning_degree_multiplier * n_neighbors`.

diversify_prob: float (optional, default=1.0) The search graph get "diversified" by removing potentially unnecessary edges. This controls the volume of edges removed. A value of 0.0 ensures that no edges get removed, and larger values result in significantly more aggressive edge removal. A value of 1.0 will prune all edges that it can.

n_search_trees: int (optional, default=1) The number of random projection trees to use in initializing searching or querying.

Deprecated since version 0.5.5.

search_epsilon: float (optional, default=0.1) When searching for nearest neighbors of a query point this values controls the trade-off between accuracy and search cost. Larger values produce more accurate nearest neighbor results at larger computational cost for the search. Values should be in the range 0.0 to 0.5, but should probably not exceed 0.3 without good reason.

tree_init: bool (optional, default=True) Whether to use random projection trees for initialization.

random_state: int, RandomState instance or None, optional (default: None) If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

n_jobs: int or None (optional, default=None) The maximum number of parallel threads to be run at a time. If none this will default to using all the cores available. Note that there is not perfect parallelism, so at several pints the algorithm will be single threaded.

low_memory: boolean (optional, default=False) Whether to use a lower memory, but more computationally expensive approach to index construction. This defaults to false as for most cases it speeds index con-

struction, but if you are having issues with excessive memory use for your dataset consider setting this to True.

max_candidates: int (optional, default=20) Internally each "self-join" keeps a maximum number of candidates (nearest neighbors and reverse nearest neighbors) to be considered. This value controls this aspect of the algorithm. Larger values will provide more accurate search results later, but potentially at non-negligible computation cost in building the index. Don't tweak this value unless you know what you're doing.

n_iters: int (optional, default=None) The maximum number of NN-descent iterations to perform. The NN-descent algorithm can abort early if limited progress is being made, so this only controls the worst case. Don't tweak this value unless you know what you're doing. The default of None means a value will be chosen based on the size of the graph_data.

early_termination_value: float (optional, default=0.001) Controls the early abort due to limited progress. Larger values will result in earlier aborts, providing less accurate indexes, and less accurate searching. Don't tweak this value unless you know what you're doing.

parallel_batch_queries: bool (optional, default=False) Whether to use parallelism of batched queries. This can be useful for large batches of queries on multicore machines, but results in performance degradation for single queries, so is poor for streaming use.

verbose: bool (optional, default=False) Whether to print status graph_data during the computation.

```
>>> from sklearn.manifold import Isomap
>>> from pynndescent import PyNNDescentTransformer
>>> from sklearn.pipeline import make_pipeline
>>> estimator = make_pipeline(
...     PyNNDescentTransformer(n_neighbors=5),
...     Isomap(neighbors_algorithm='precomputed'))
```

fit(*X*, *compress_index=True*)

Fit the PyNNDescent transformer to build KNN graphs with neighbors given by the dataset X.

X [array-like, shape (n_samples, n_features)] Sample graph_data

transformer [PyNNDescentTransformer] The trained transformer

fit_transform(*X*, *y=None*, ***fit_params*)

Fit to graph_data, then transform it.

Fits transformer to X and y with optional parameters fit_params and returns a transformed version of X.

X [numpy array of shape (n_samples, n_features)] Training set.

y : ignored

Xt [CSR sparse matrix, shape (n_samples, n_samples)] Xt[i, j] is assigned the weight of edge that connects i to j. Only the neighbors have an explicit value. The diagonal is always explicit.

transform(*X*, *y=None*)

Computes the (weighted) graph of Neighbors for points in X

X [array-like, shape (n_samples_transform, n_features)] Sample graph_data

Xt [CSR sparse matrix, shape (n_samples_transform, n_samples_fit)] Xt[i, j] is assigned the weight of edge that connects i to j. Only the neighbors have an explicit value.

A number of internal functions can also be accessed separately for more fine tuned work.

2.7.2 Distance Functions

`pynndescent.distances.`**`chebyshev`**

Chebyshev or l-infinity distance.

$$D(x, y) = \max_i |x_i - y_i|$$

`pynndescent.distances.`**`euclidean`**

Standard euclidean distance.

$$D(x, y) = sqrt \sum_i (x_i - y_i)^2$$

`pynndescent.distances.`**`manhattan`**

Manhattan, taxicab, or l1 distance.

$$D(x, y) = \sum_i |x_i - y_i|$$

`pynndescent.distances.`**`minkowski`**

Minkowski distance.

$$D(x, y) = \left(\sum_i |x_i - y_i|^p\right)^{\frac{1}{p}}$$

This is a general distance. For p=1 it is equivalent to manhattan distance, for p=2 it is Euclidean distance, and for p=infinity it is Chebyshev distance. In general it is better to use the more specialised functions for those distances.

`pynndescent.distances.`**`squared_euclidean`**

Squared euclidean distance.

$$D(x, y) = \sum_i (x_i - y_i)^2$$

`pynndescent.distances.`**`standardised_euclidean`**

Euclidean distance standardised against a vector of standard deviations per coordinate.

$$D(x, y) = \sqrt{\sum_i \frac{(x_i - y_i) **2}{v_i}}$$

`pynndescent.distances.`**`weighted_minkowski`**

A weighted version of Minkowski distance.

$$D(x, y) = \left(\sum_i w_i |x_i - y_i|^p\right)^{\frac{1}{p}}$$

If weights w_i are inverse standard deviations of graph_data in each dimension then this represented a standardised Minkowski distance (and is equivalent to standardised Euclidean distance for p=1).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index