Nearest neighbor search

Outline

Why nearest neighbour search?
Overview of KNN methods
Evaluating a KNN algorithm
Write your own KNN!

Many models resort to representation learning

my data -> vectors!

It becomes natural to look for similar items

Nearest neighbors!

Brute-force complexity is linear ☹

Compare the key item with all existing ones

We want to do better (i.e. sub-linear)

Many datasets today > million of items

About distances

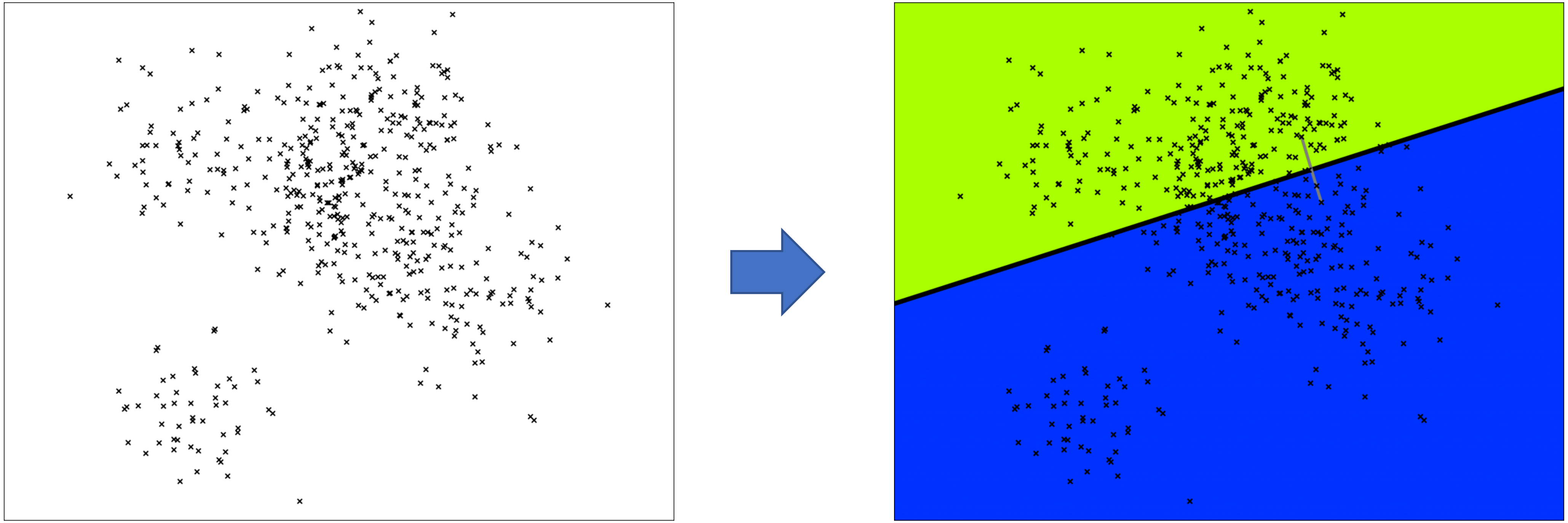
Measure	Meaning	Formula	Relationship to increasing similarity
Euclidean distance	Distance between ends of vectors	$\sqrt{(a_1-b_1)^2+(a_2-b_2)^2+\ldots+(a_N-b_N)^2}$	Decreases
Cosine	Cosine of angle θ between vectors	$\frac{a^T b}{\lvert a\rvert \cdot \lvert b\rvert}$	Increases
Dot Product	Cosine multiplied by lengths of both vectors	$a_1 b_1+a_2 b_2+\ldots+a_n b_n=\lvert a\rvert\lvert b\rvert cos(\theta)$	Increases. Also increases with length of vectors.

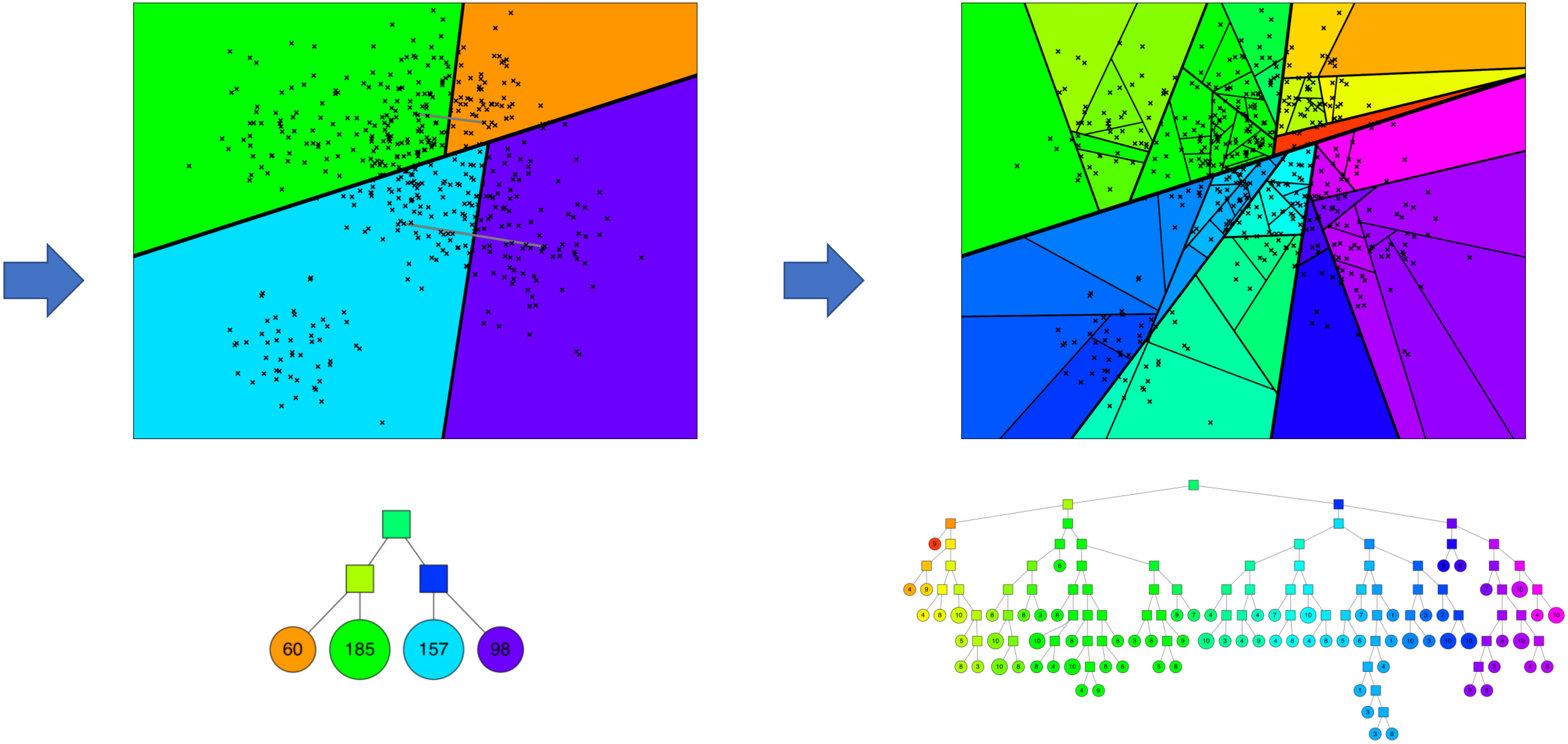
Overview of KNN methods

Tree based
Hash based
Graph based

Tree-based

Split the sub-space



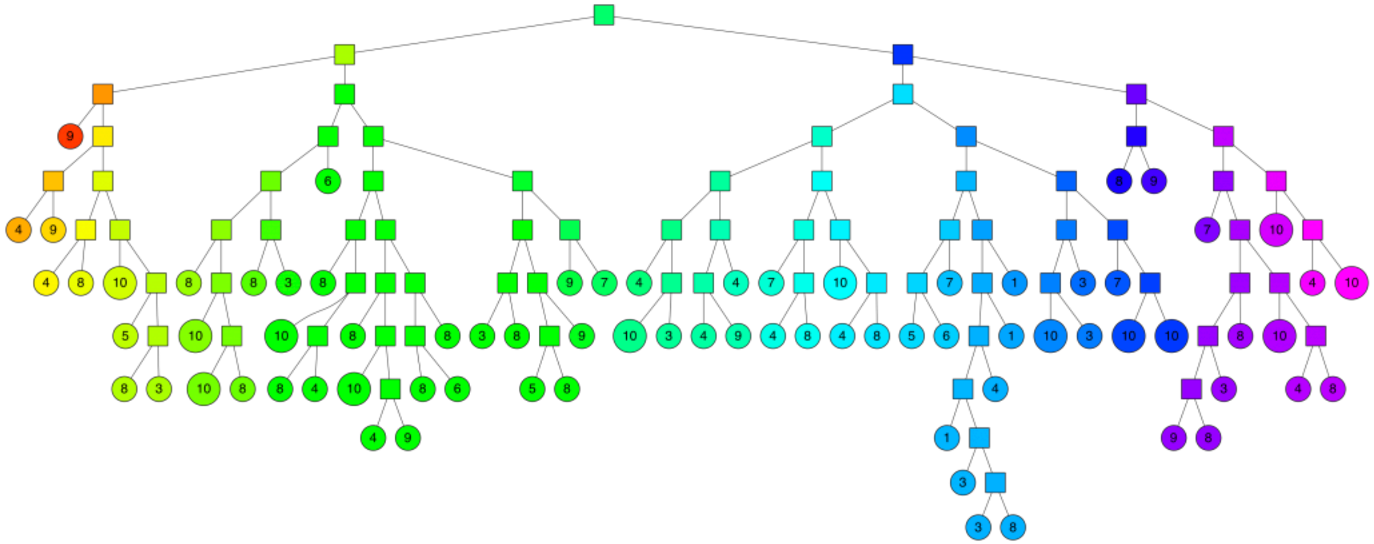

60
185
157
98

In practice

Build multiple trees
Split randomly

In practice

Search? Go down the tree!



Hash based

e.g. LSH (local sensitivity hashing)

Reminder: what is hashing?

Compress a dimension into a smaller dimensio

Reminder: what is hashing?

Typically, we want to minimize collisions.

Locality-sensitive hashing

Here we want to maximize collisions.

1. Pr(h(a) == h(b)) is high if a and b are near
2. Pr(h(a) == h(b)) is low if a and b are far
3. Time complexity to identify close objects is sub-linear.

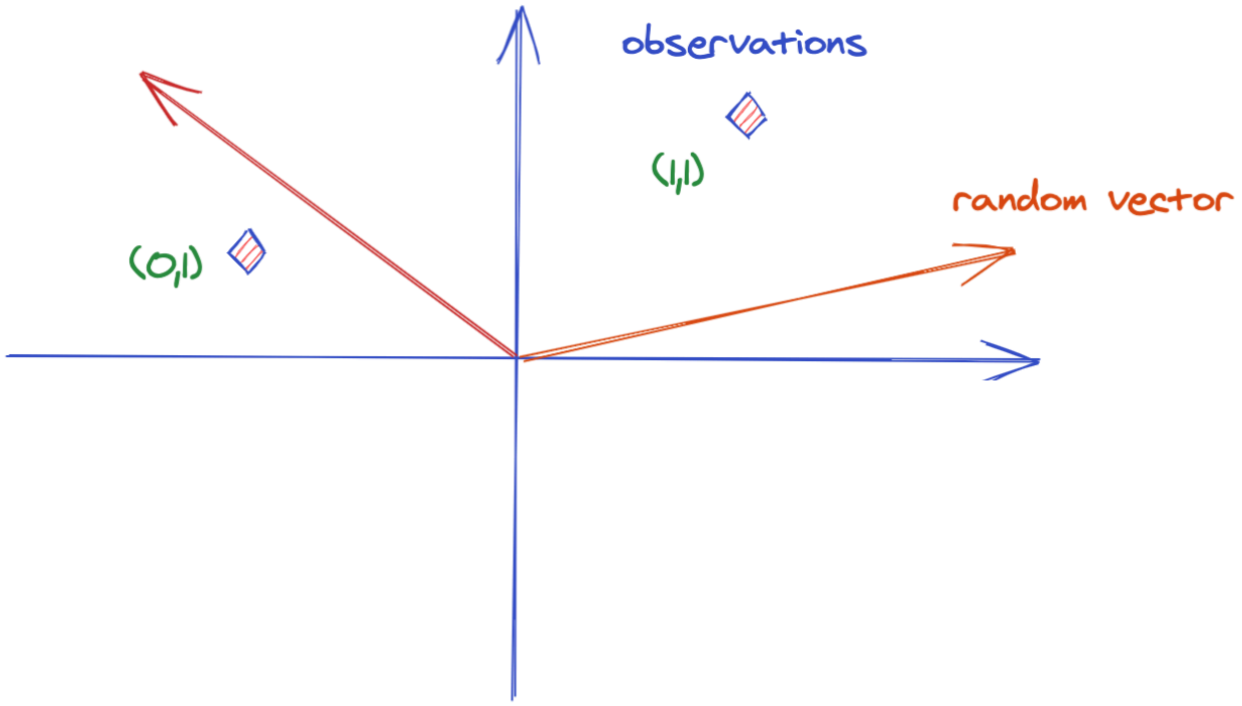
Locality-sensitive hashing

Example: Bit sampling

Random projections

If points in a vector space are of sufficiently high dimension, then they may be projected into a suitable lower-dimensional space in a way which approximately preserves the distances between the points.

Random projections



Random projections

$$\left[\quad Projected(P) \quad\right]_{k\times n} = \left[\quad Random(R) \quad\right]_{k\times d} \left[\quad Original(D) \quad\right]_{d\times n}$$

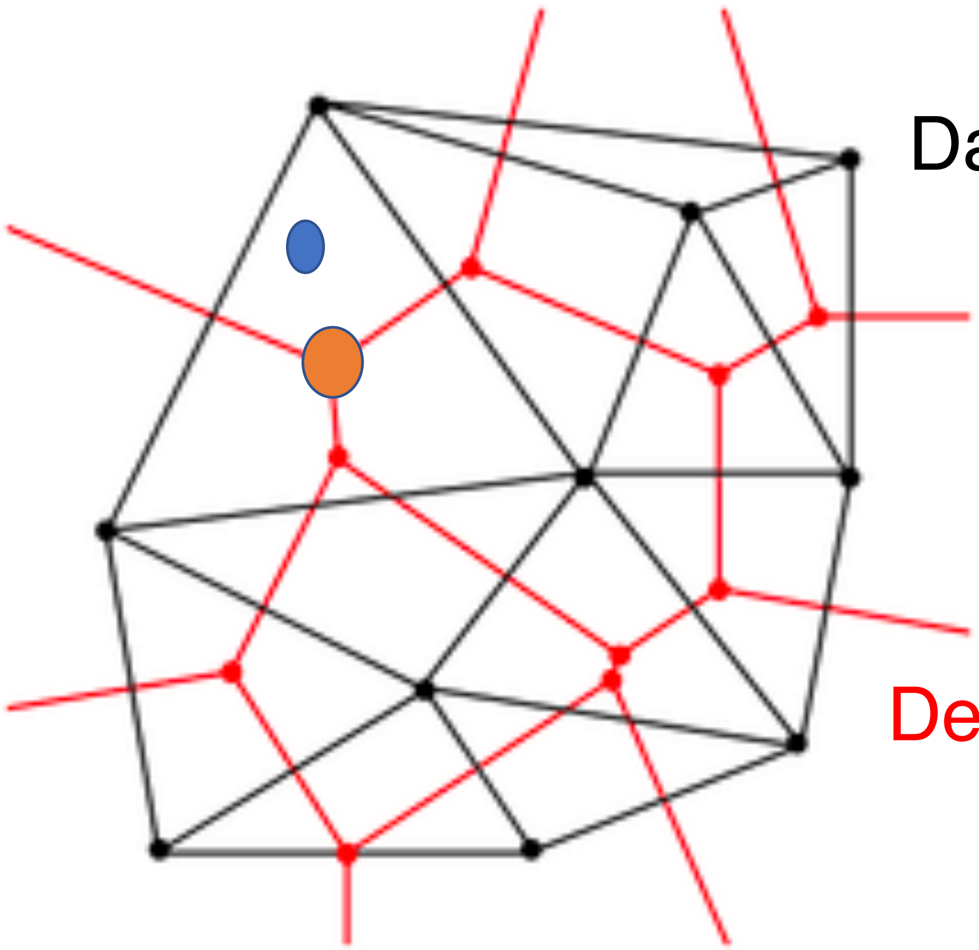
Choose hash table size and number of hash tabl

Random projections

Two vectors' bits match with probability proportional to the cosine of the angle between them.

Graph based

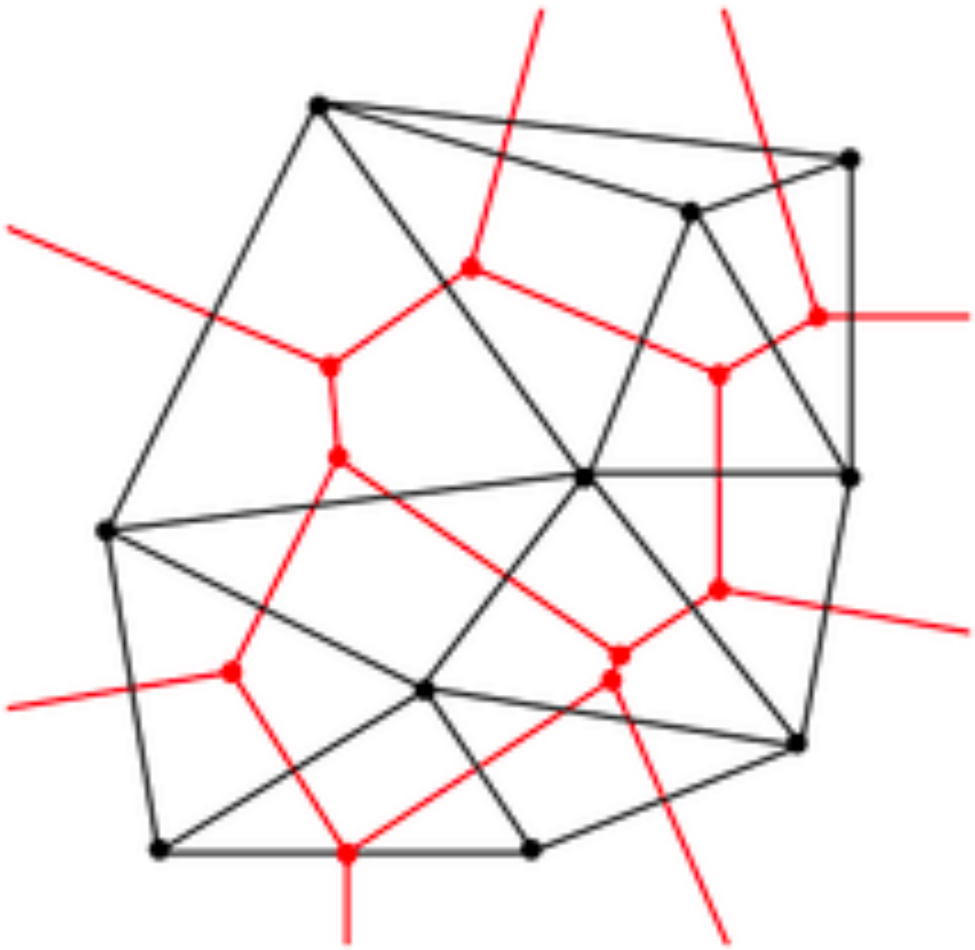
Delaunay triangulation guarantees that a greedy traversal yields the nearest neighbor.



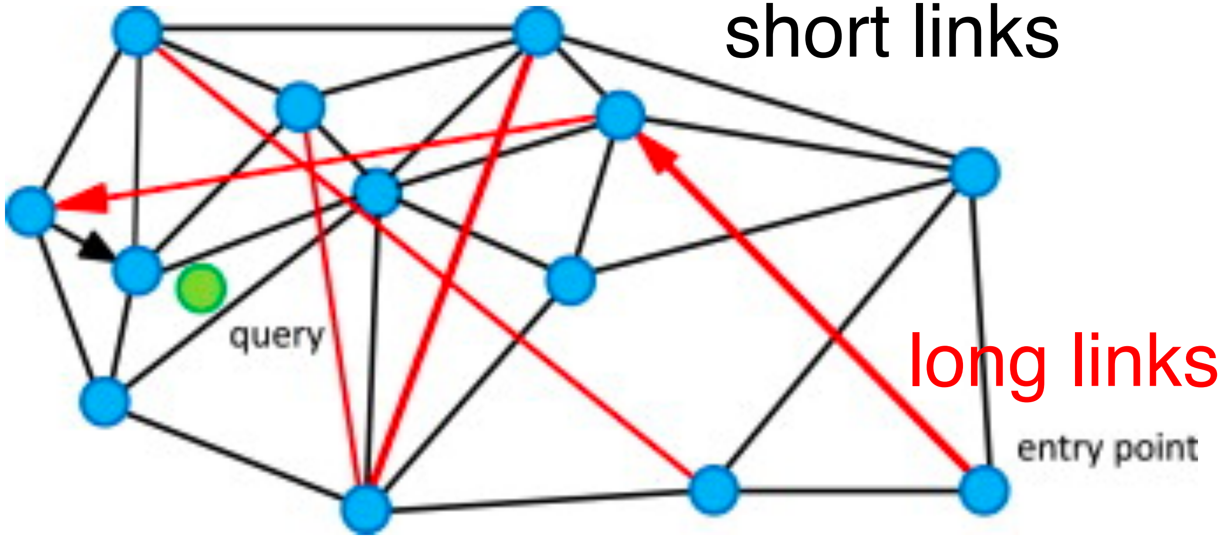
Graph based

But Delaunay triangulation is expensive ☹

$$O(n^{\lceil \frac{d}{2} \rceil})$$

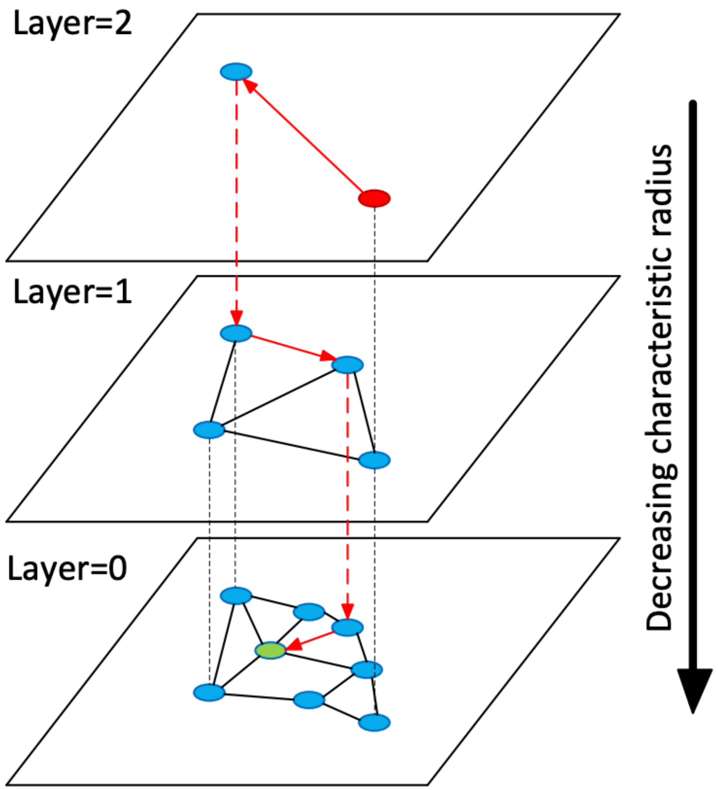


Navigable small worlds to the rescue



[Approximate nearest neighbor algorithm based on navigable small world graphs, Malkov, Yu and Ponomarenko, Alexander and Logvinov, Andrey and Krylov, Vladimir, Information Systems, 2013.]

Better: hierarchical navigable small worlds



hnsw

[Malkov, Yu A., and D. A. Yashunin. "Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs." TPAMI]

Recap

Tree based (annoy)
Hash based (lsh)
Graph based (hnsw)

But which one is best?

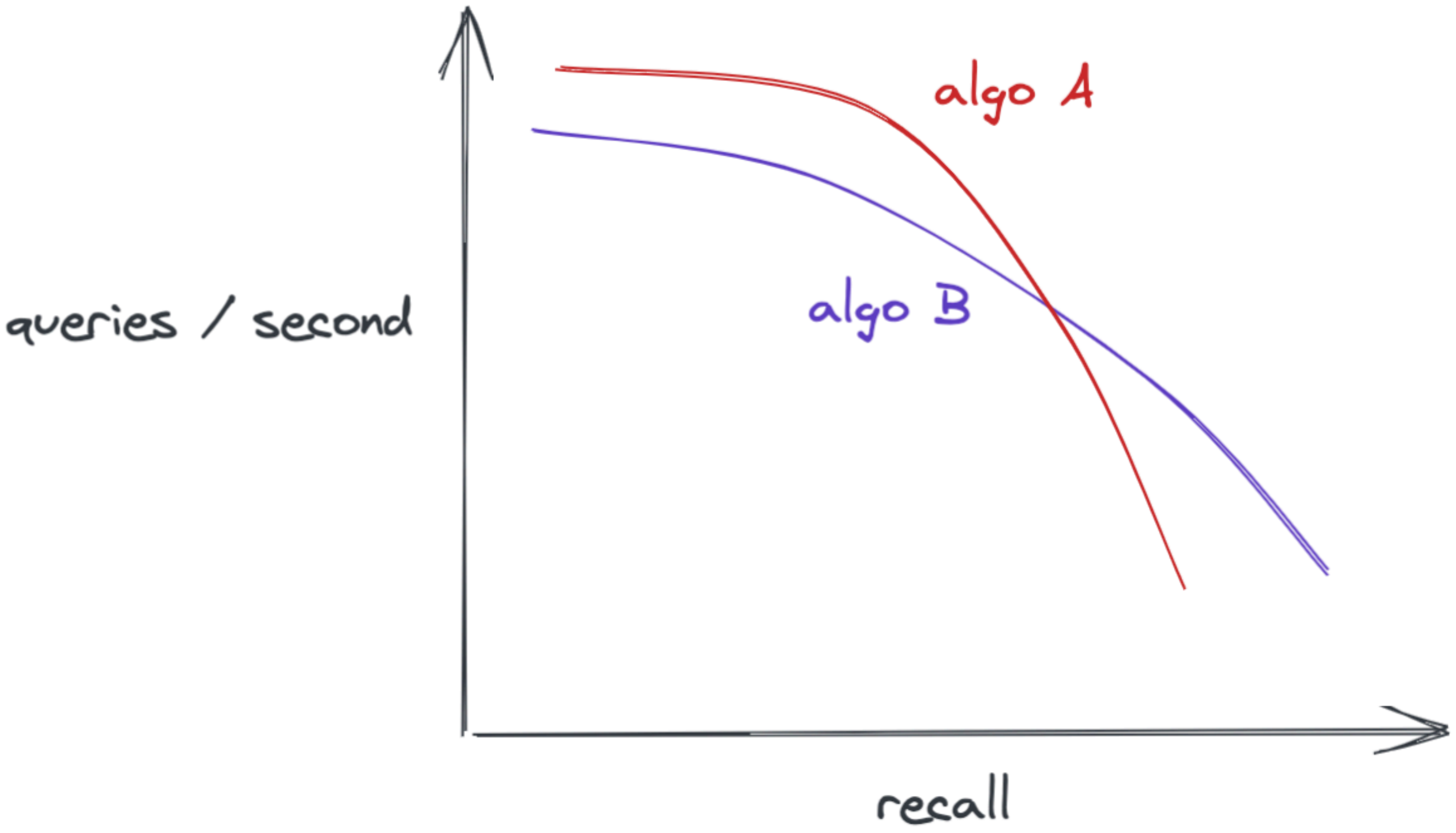
Evaluating a knn algorithm

1. Algorithmic performance
2. Computing performance

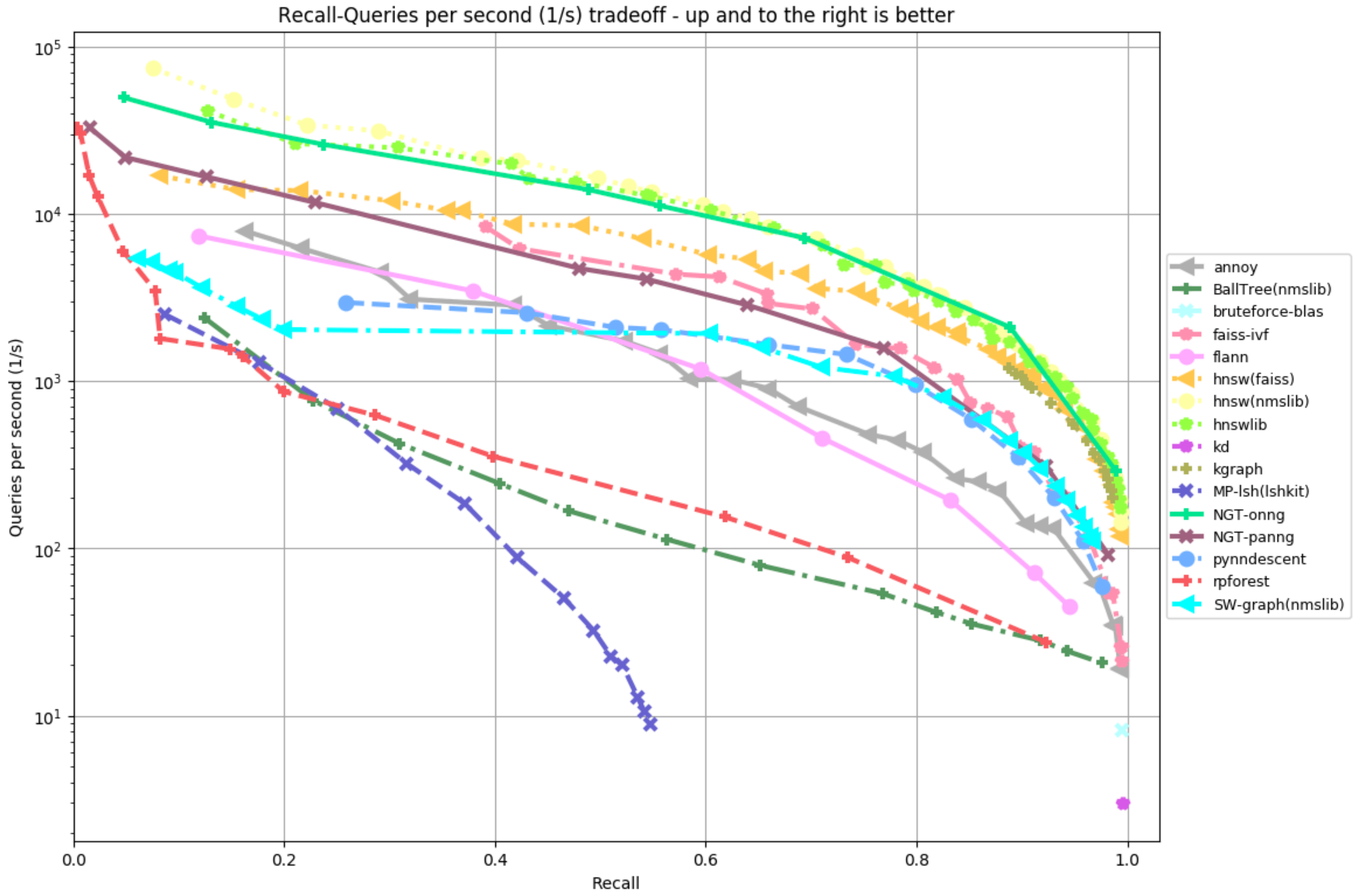
Evaluating a knn algorithm

1. Algorithmic performance –> recall
2. Computing performance -> qps

Evaluating a knn algorithm



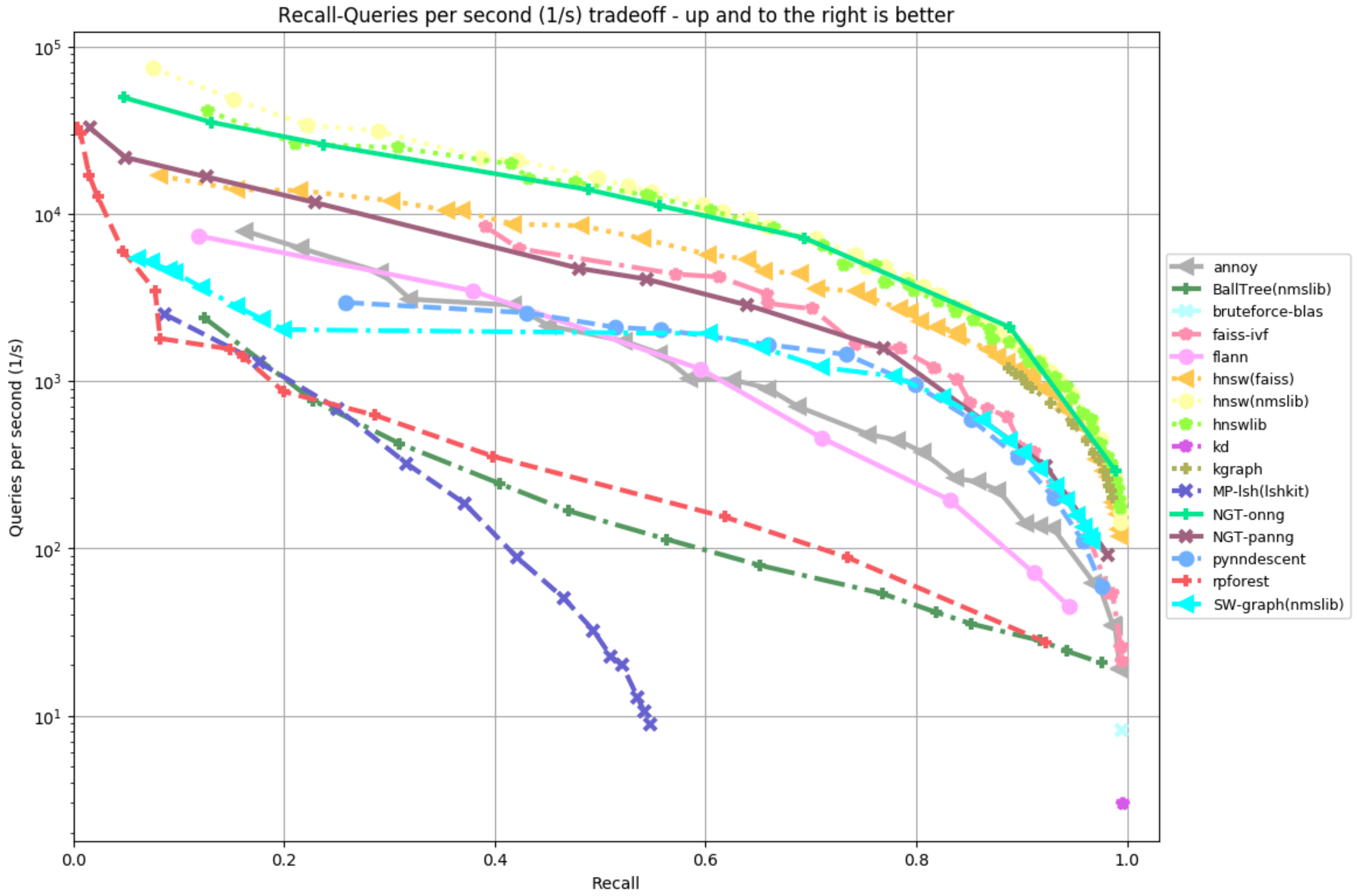
And the winner is…



http://ann-benchmarks.com

But wait, on which dataset???

And the winner is…



http://ann-benchmarks.com

Keys to choosing the right KNN algorithm

1. Pick a distance
2. Analyze your dataset
3. Define a recall target
4. Know your compute budget
5. Incorporate tech debt / code simplicity
6. Build a brute-force benchmark

An example with Annoy

```
dim = 10
index = AnnoyIndex(dim, 'angular’)

index.add_item(…)

num_trees = 16
index.build(num_trees)

index.get_nns_by_item(…)
```

And now… hands-on work!