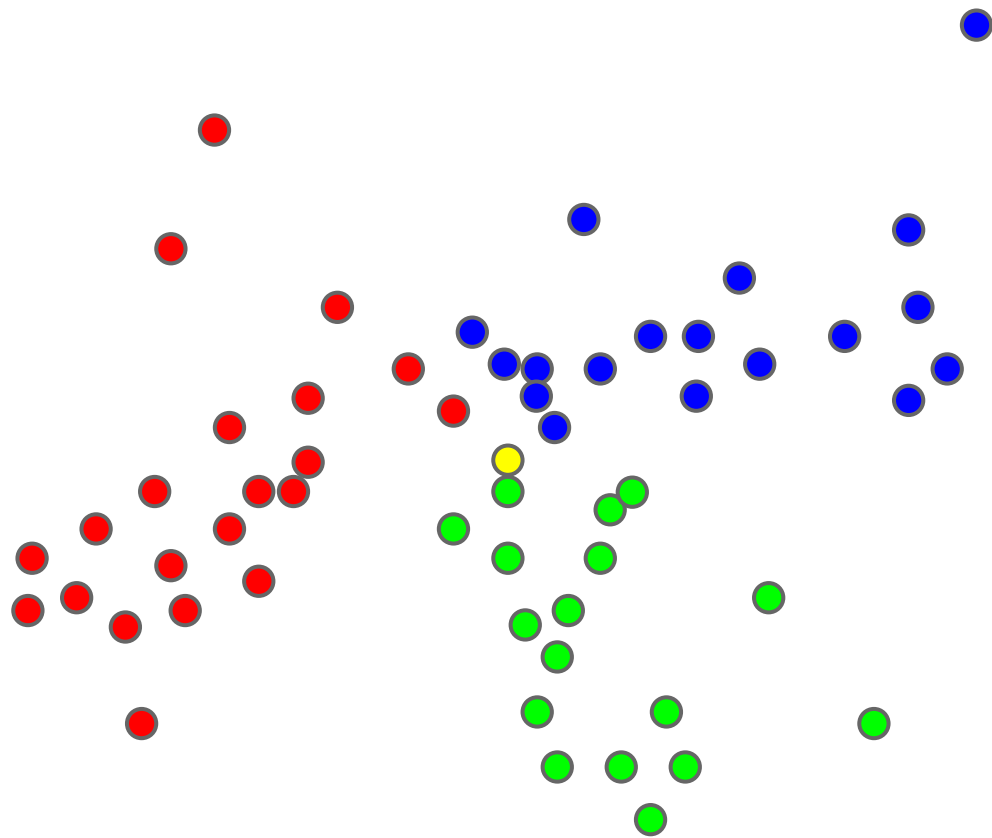


Scaling up *k*-nearest neighbours classification



Andrew Clegg
Pearson Technology
[@andrew_clegg](#)

Objective

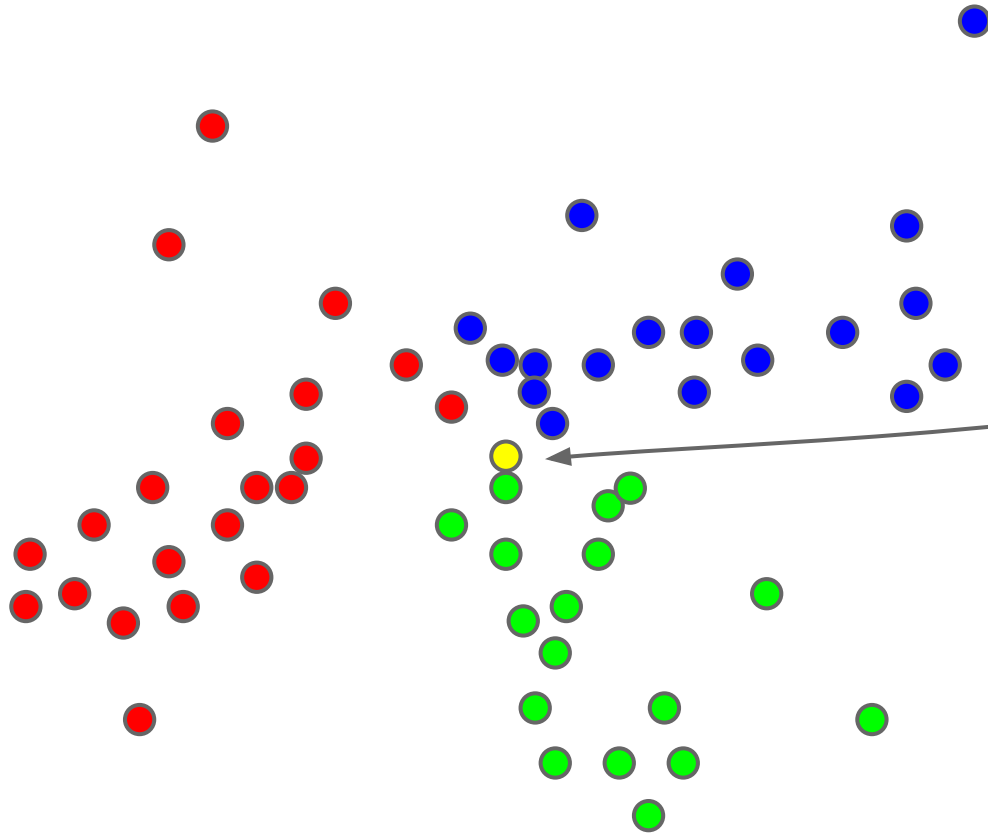
Classify an unseen item into one of two or more categories.

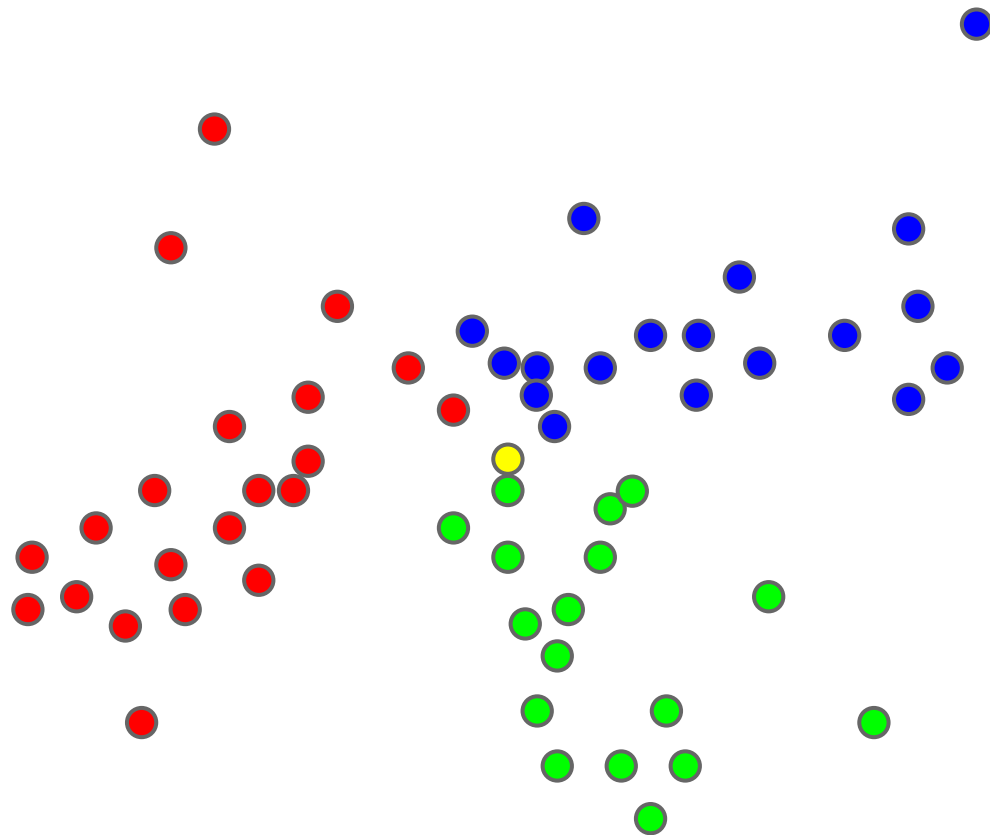
Classification is based on similarity to previously-seen items.

k = number of items to take into account.

n items in an m -dimensional space.

(Example shown has 2 dimensions.)





Ingredients

- Some data
- A similarity measure

That's all you need for a naïve approach.

Training

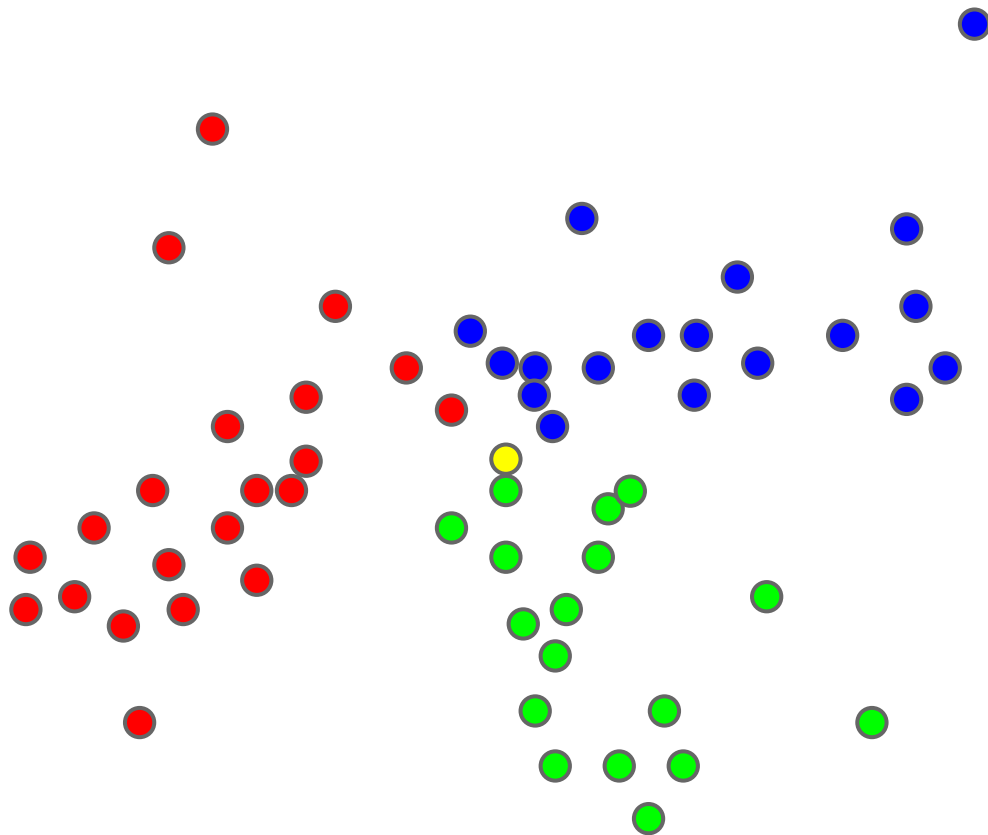
1. Add each labeled item to a collection

... done.

Querying

1. Get k nearest neighbours of query item
2. Take majority vote of their labels
3. Apply label to query item

Simples.



Variations

- Weighted kNN

Weight each vote by distance or rank.

- kNN regression

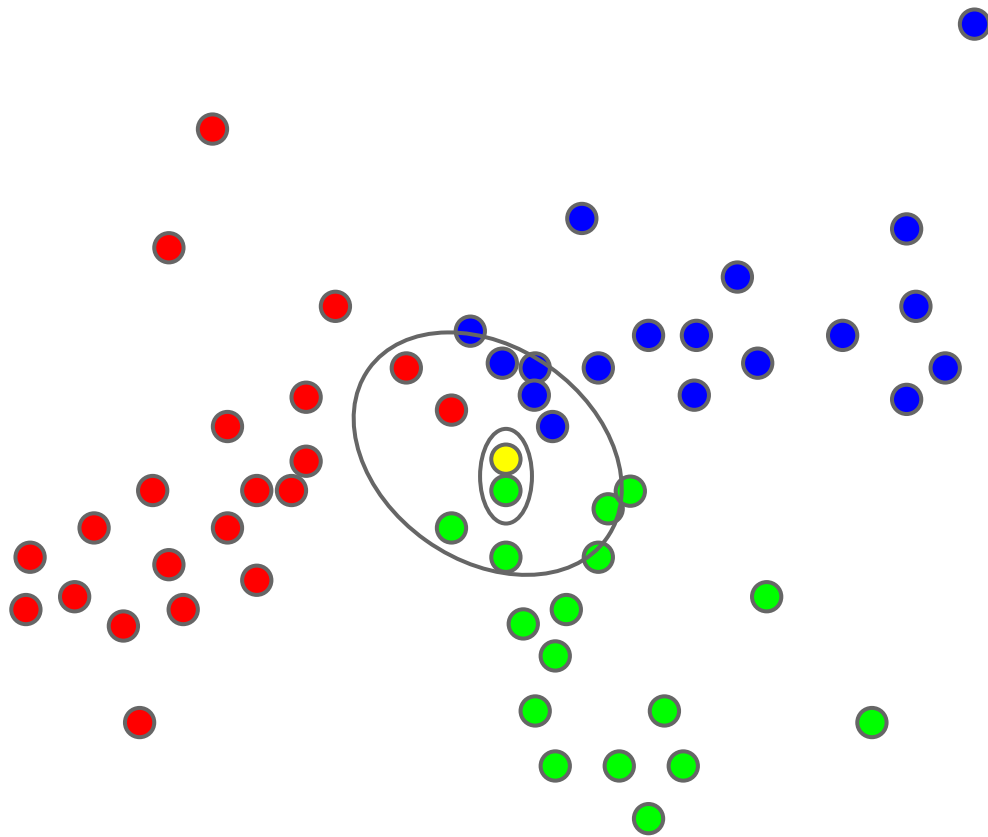
Goal is to predict a numeric value.

Take [weighted] average of value, across k nearest training instances.

- Large-margin nearest neighbours
- Shared nearest neighbours
- Fixed-radius nearest neighbours
- Graph nearest neighbours

Related:

- k-means clustering
- Deduplication / record linkage



Advantages

- Basically model-free

Not many assumptions needed.

- Parametric at *query* time

k and weighting strategy can be tuned at runtime without expensive retraining.

Perfect for A/B testing, bandit algorithms, etc.

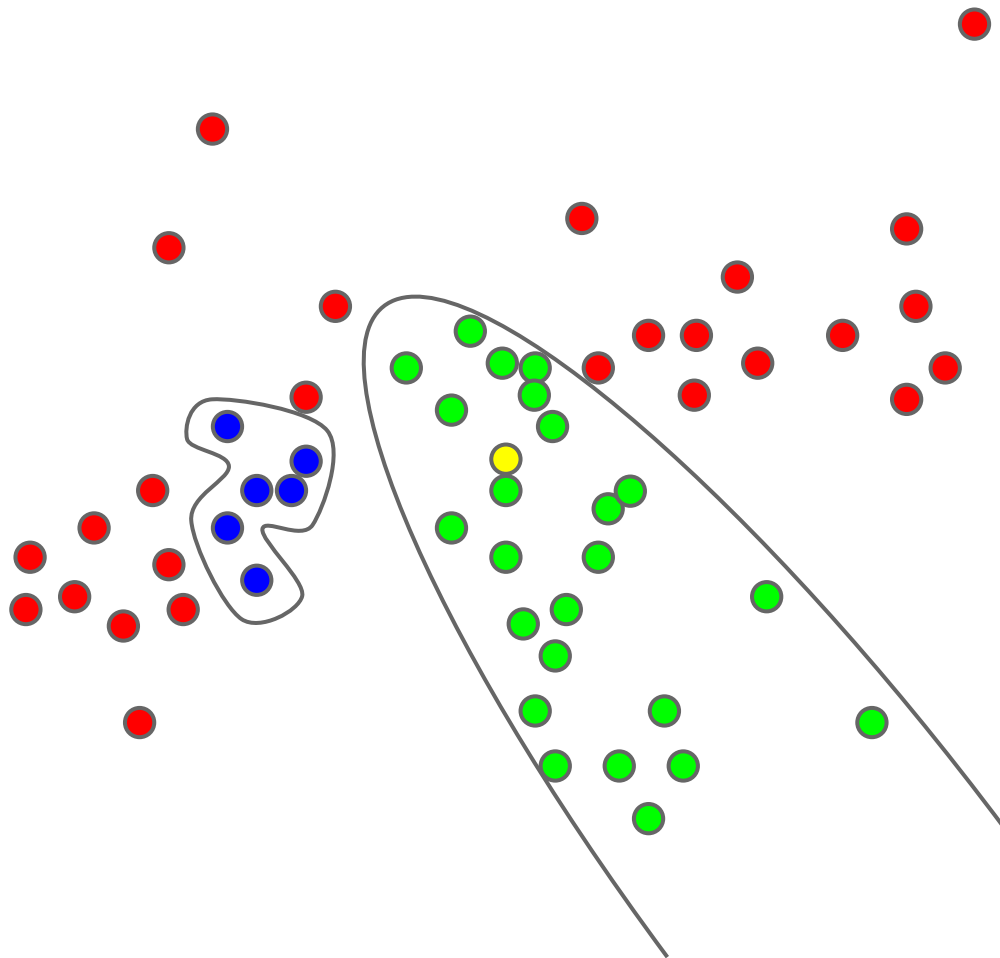
k lets you adjust bias/variance tradeoff.

- Transparent

Easy to explain why a new instance got a particular label.

- Updatable

Add new 'training' items is fast.



Advantages

- Flexible

Happy to deal with data that's non-linear, non-convex, or irregularly shaped.

- Parallelizable

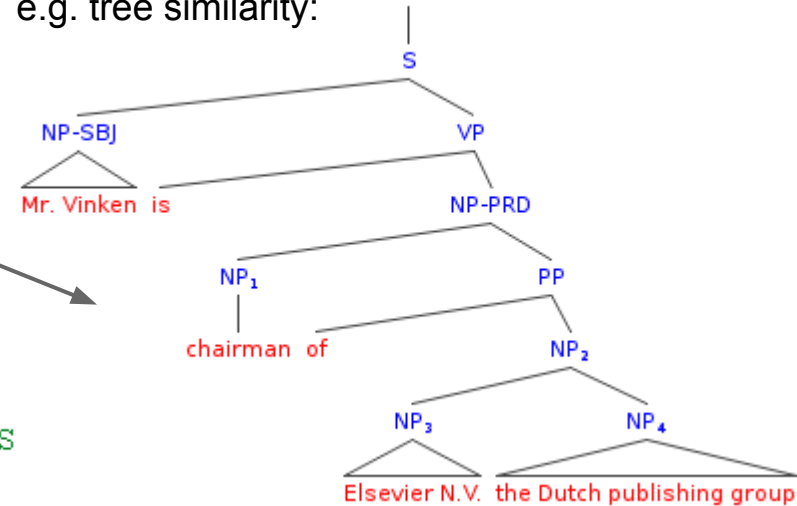
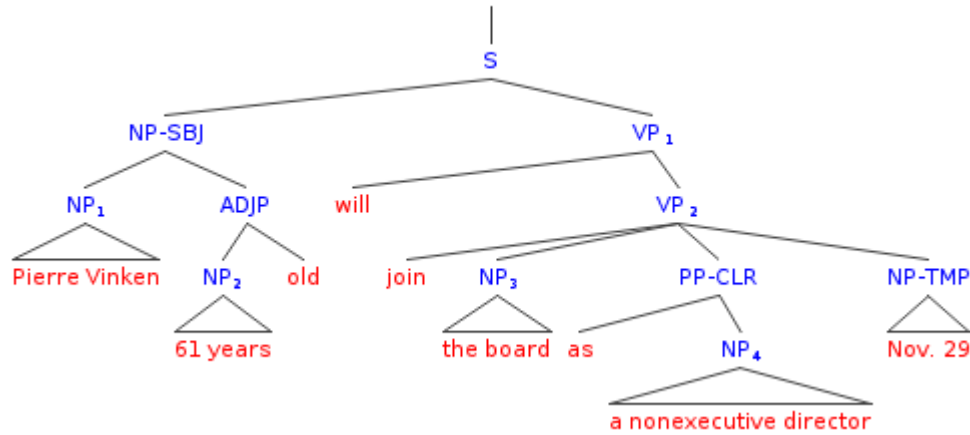
Trivially easy to query in parallel:

1. Shard dataset arbitrarily
2. Get k top hits for each shard
3. Merge results from all shards

Advantages


Amenable to domain-specific similarity measures.


e.g. tree similarity:



HSHLQCHKRTHTGKPYECNQCGKAFSQHGLLQRHKRTHTGKPYMNVINMVKPLHNS

PSHLQYHERTHTGKPYECHQCGQAFKKCSLLQRHKRTHTGKPYE-CNQCGKAFAQ-

n dimensions 

m items 

0	0	0	7	0	0	9	2	0	0	0	3
1	0	0	3	4	0	0	6	0	0	0	5
...											

Typical use case

Often used with n numeric attributes, and a distance metric or pseudo-metric:

- Euclidean
- Manhattan
- Cosine
- Pearson correlation
- ...

The right one to choose depends on your data.
If in doubt, cross-validate.

Data often high-dimensional and/or sparse --
this affects the choice of metric.

n and m can both grow

In some applications, dimensionality grows over time as new data comes in.

	<i>the</i>	<i>cat</i>	<i>sat</i>	<i>on</i>	<i>mat</i>	<i>quick</i>	<i>brown</i>	...
<i>Email 1</i>	2	1	1	1	1	0	0	
<i>Email 2</i>	2	0	0	0	0	1	1	
...								

	<i>Song 1</i>	<i>Song 2</i>	<i>Song 3</i>	<i>Song 4</i>	<i>Song 5</i>	<i>Song 6</i>	<i>Song 7</i>	...
<i>User 1</i>	2	1	1	1	1	0	0	
<i>User 2</i>	2	0	0	0	0	1	1	
...								

$$O(m * n)$$

Disadvantages

- Query time complexity

Comparing new items to whole dataset is **slow**.

Distance metrics in m dimensions are typically $O(m)$ so each query is $O(m * n)$.

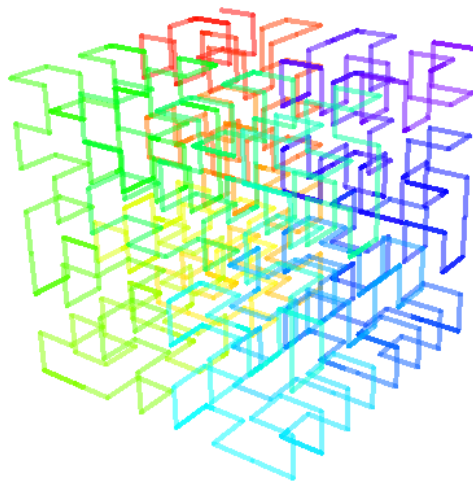
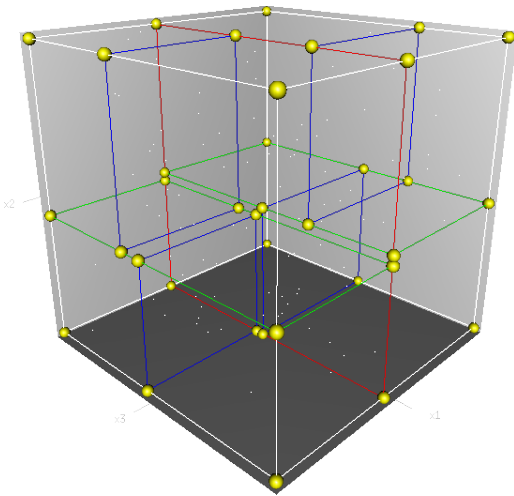
More generally, $O(c * n)$ where c is the cost of comparing two items.

- Storage requirements

Generally, this is the whole data set. Plus any indices etc. to speed up queries, more later...

- Curse of dimensionality

The more dimensions you have, the less different the distances get. Various solutions. But that's a whole different talk.



Strategies

- Reduce num. items compared
- Reduce num. dimensions compared

Or both.

- Deterministic (exact)

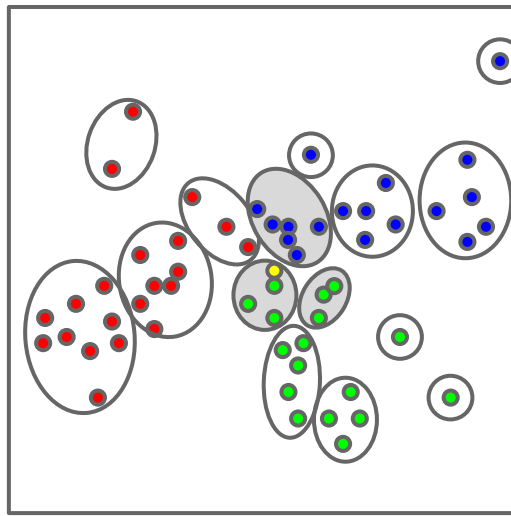
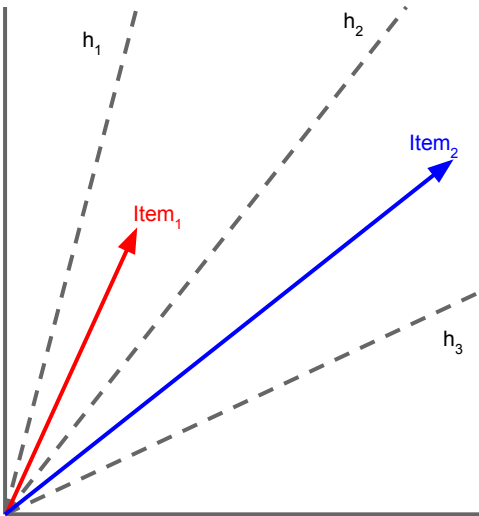
Definitely finds the k nearest neighbours.

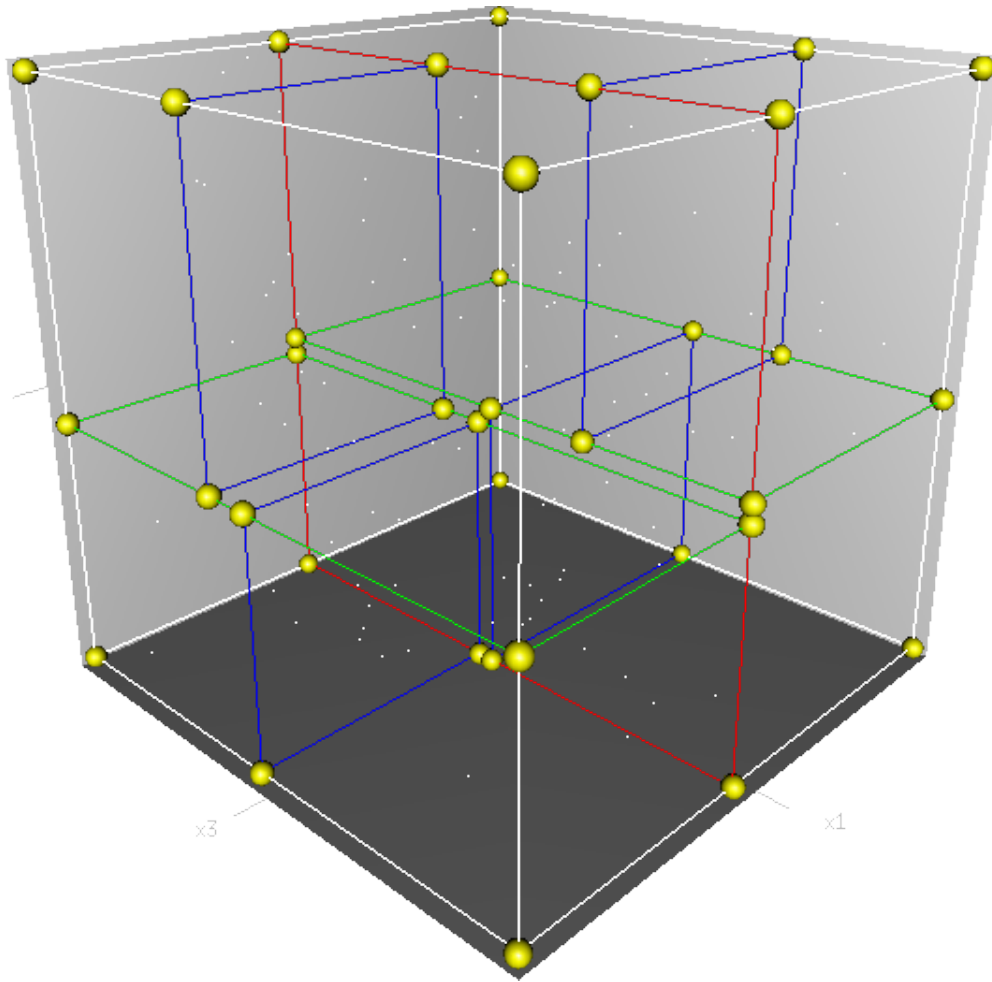
- Probabilistic (approximate)

Finds k neighbours which are *probably near enough* to get a good result.

This may involve an approximate search to find candidate neighbours, followed by exact comparisons to each candidate.

Most approximate methods have provable error bounds, and tunable error vs. performance tradeoffs.





Space partitioning trees

Lots of types: k-d trees, R-trees, X-trees ...

- Basic idea

Partition search space into nested regions.

Represent this partitioning as a tree.

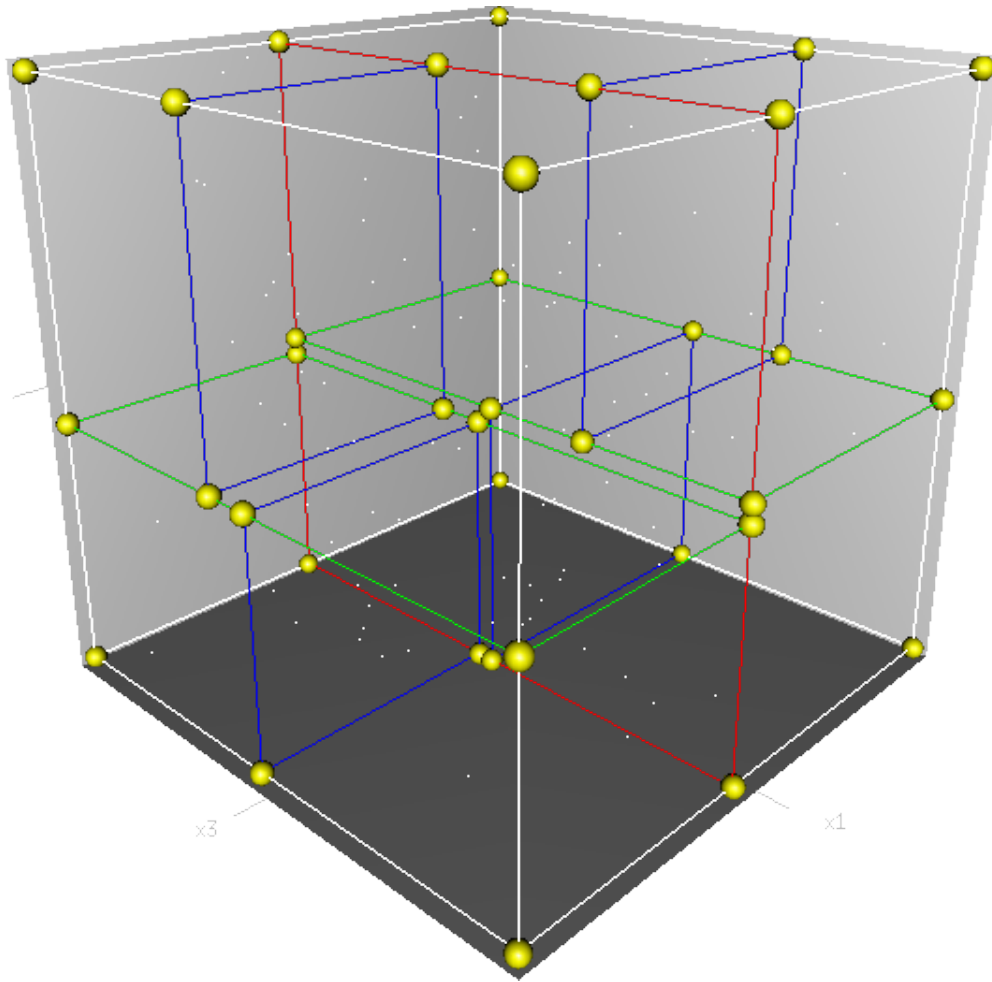
- Search strategy

Find region (leaf node) containing query item.

Get all neighbours found there.

Expand outwards until enough neighbours are found.

Exact and approximate variations.



Space partitioning trees

- Typical performance

$O(\log n)$ for both insertion and querying -- with ideal data. Worst case $O(n)$.

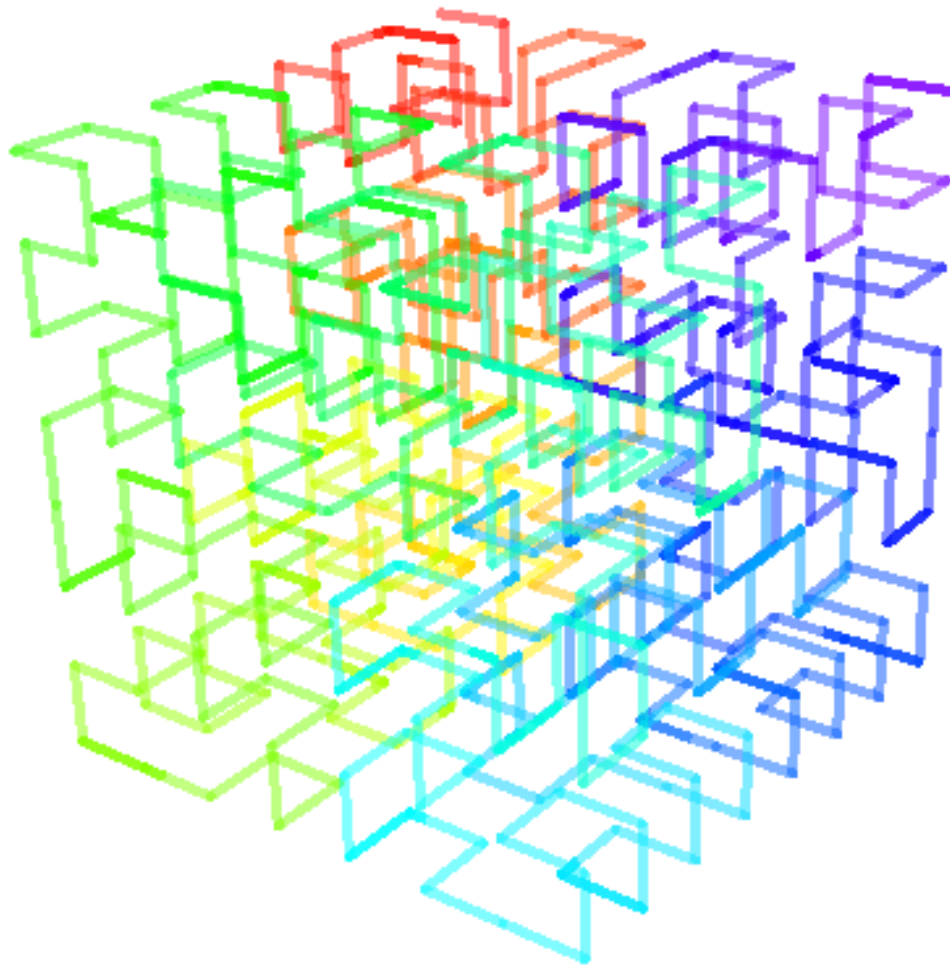
Construction time: $O(mn \log n)$ (kd-trees) or $O(n \log n)$ (most others).

- Suitable for data with...

Low-ish dimensionality (10s not 100s) and finite spatial bounds. Data known in advance.

At higher dimensions, storage increases, search degrades towards $O(n)$ and accuracy of approximate methods suffers.

Best when entire dataset is known up front. Otherwise tree will become unbalanced.



Space filling curves

- Basic idea

Fill search space with curve such that every item has a linear 'address' on that curve.

Items adjacent on curve are close in distance. But not necessarily vice versa.

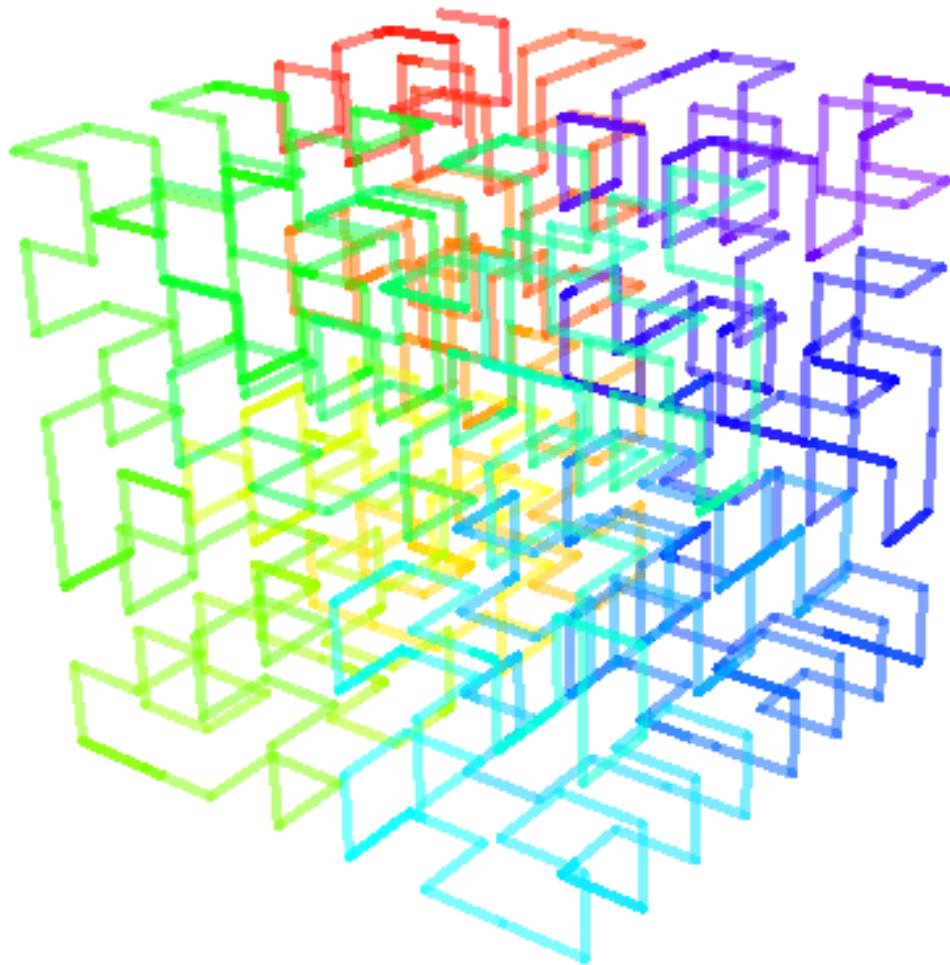
So, use multiple curves, rotated or shifted.

- Search strategy

Find candidate items close to query item on *any* curve.

Calculate true distance for each.

This is an approximate method -- some points may be near in space, but far on any curve.



Space filling curves

- Typical performance

Insertion: $O(m \log n)$.

Querying: $O(m \log n)$ for retrieving candidates plus additional time for exact matches.

Construction depends on dimensionality and order ('curviness'), but not n .

- Data suitability

Similar restrictions to space partitioning trees.

- Practical advantages

Range queries are more cache- and disk-friendly than tree traversals.

Can be performed efficiently with standard B-tree indices, e.g. using an RDBMS.

"locality" => 01011010

11110010

"localize" => 01011010

11110010

"location" => 01010010

11110010

"vocation" => 01010011

11110010

"vacating" => 01010011

01110010

Locality sensitive hashing

- Basic idea

LSH algorithms yield identical or similar hashes for similar input data.

Sometimes known as a 'sketch' or 'fingerprint'.

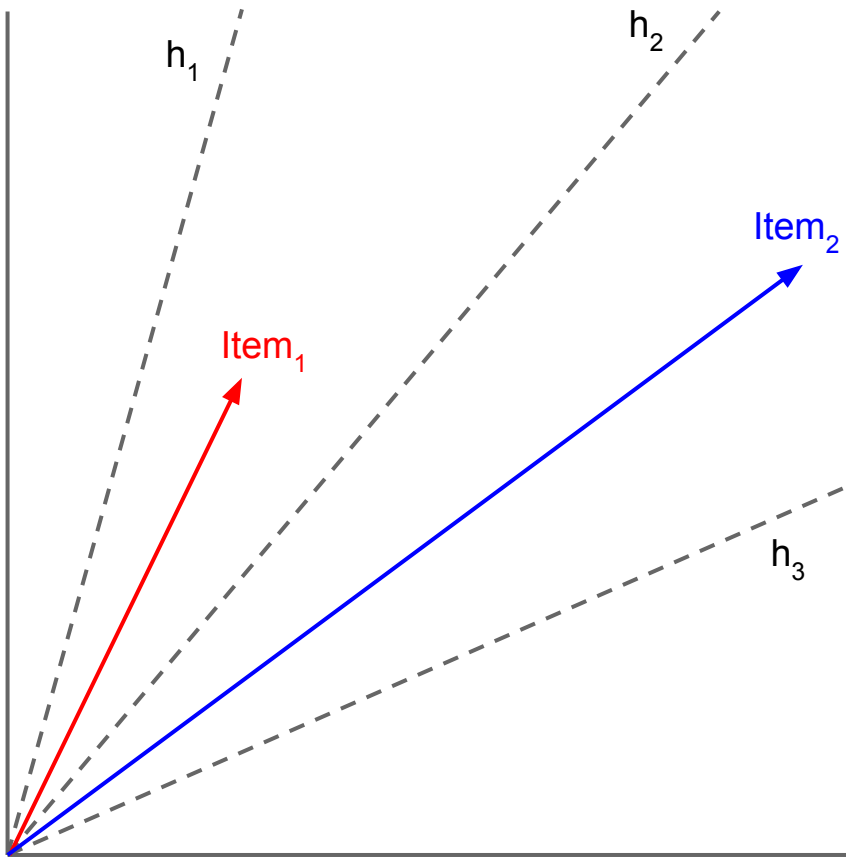
Effective in fairly high-dimensional space.

- Application to *k*NN

Find candidate matches by looking for items with same or similar hash to query item.

Or, use multiple random hashes from same family. Get candidates which are equal under any of these hashes.

Then perform exact NN on these candidates.



Random hyperplanes method

- Estimator for cosine similarity

Draw h random hyperplanes in vector space.

For each hyperplane:

Is the vector above it (1) or below it (0)?

Hash($Item_1$) = 011

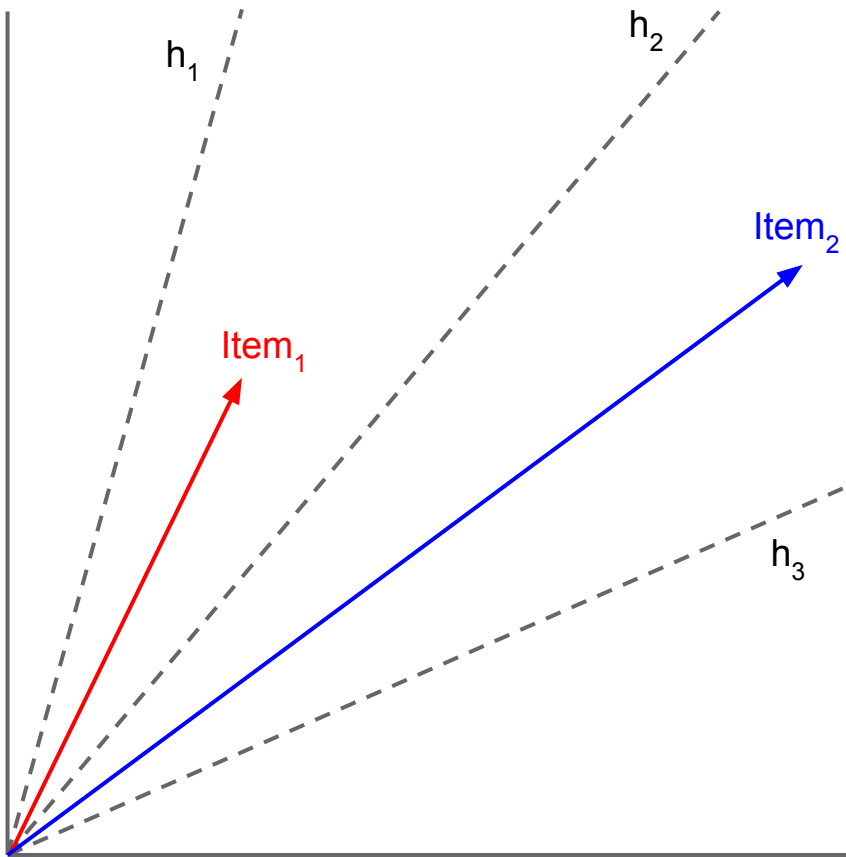
Hash($Item_2$) = 001

Hamming distance between hashes approximates cosine distance between vectors.

- Increasing h increases granularity

Hashes more costly to calculate and store.

Smaller number of items per unique hash (can be seen as precision/recall tradeoff).



Random hyperplanes method

- Search strategy (basic approach)

Find items where hash matches query item.

This is probably constant w.r.t. dataset size.

Assuming even data and good hash, this gives expected $r = n/h$ results.

Then do exact NN: $O(m * r)$

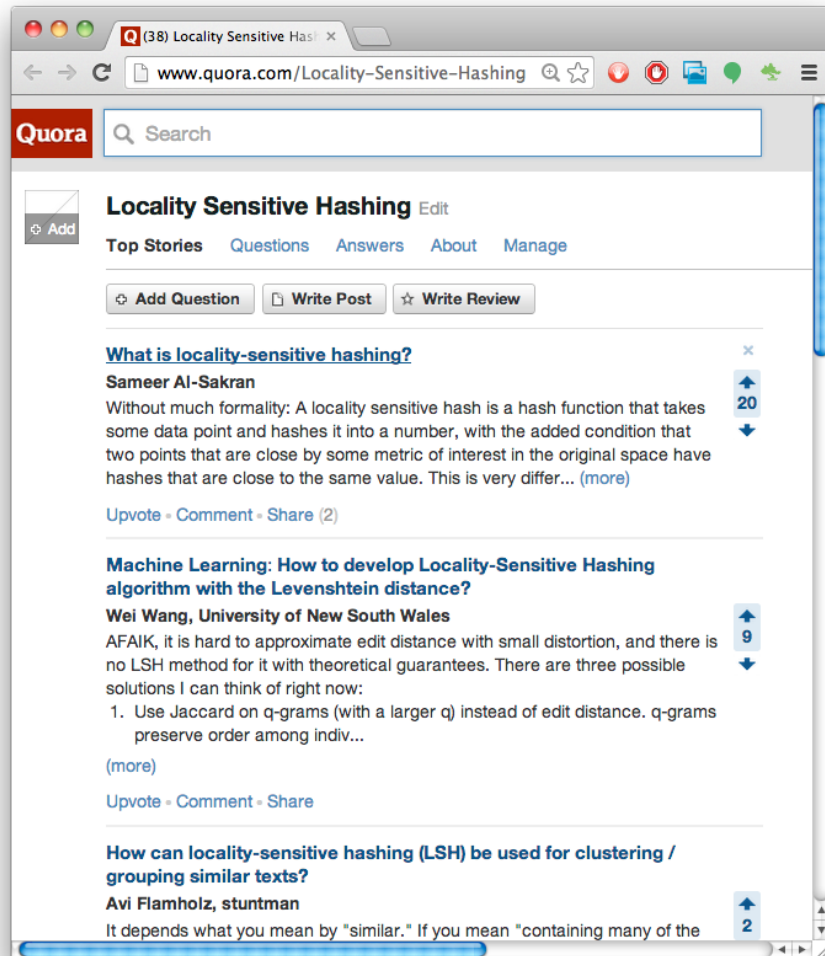
- Fallback strategy might be required

Clumpy data, high h = potentially less than k candidates found.

Can use secondary hash with lower h .

(Could just use subset of original hash.)

Or best guess/unknown if use case allows.



Other LSH methods - many more exist

- Random projection

Like hyperplanes, but for Euclidean distance.

- Bit sampling

Approximates Hamming similarity between two equal-length bit strings.

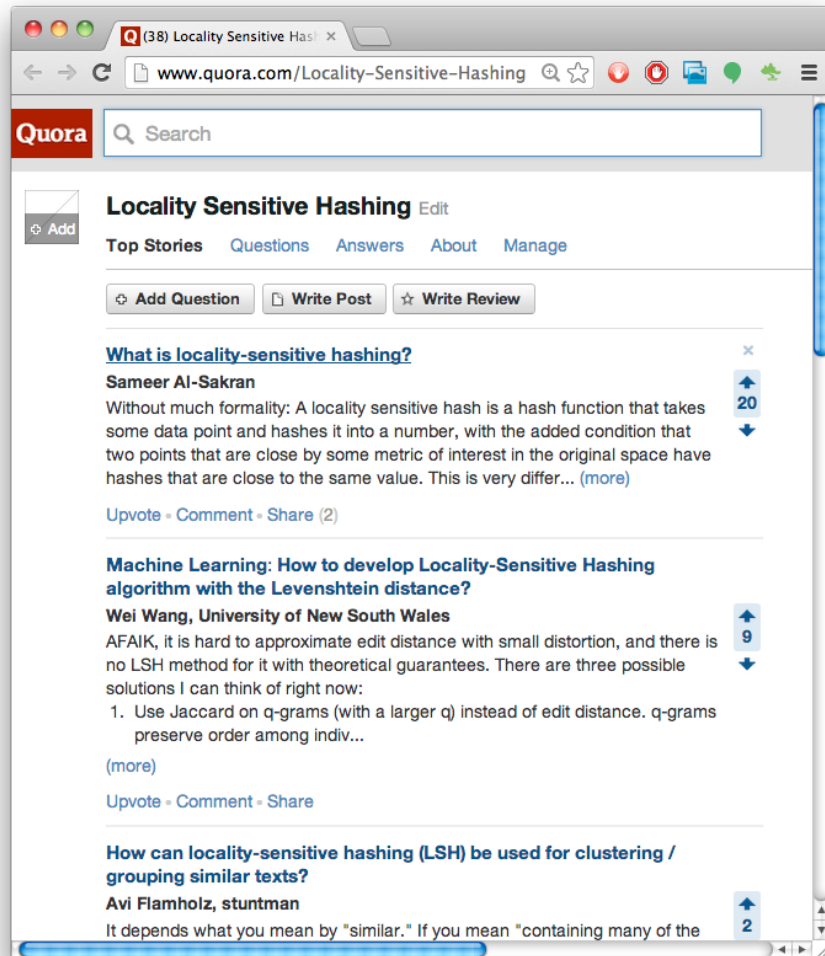
Only compare random subset of bits.

Can be used for expanding candidate results sets from *other* LSH methods!

- MinHash

Approximates Jaccard similarity between two sets of countable items.

Slightly too complex to describe in three lines!



Related work

- Minimal loss hashing

Learns *domain-specific* LSH functions from labelled or semi-labelled training data.

- Multi-index hashing

Efficient exact NN for Hamming distance.

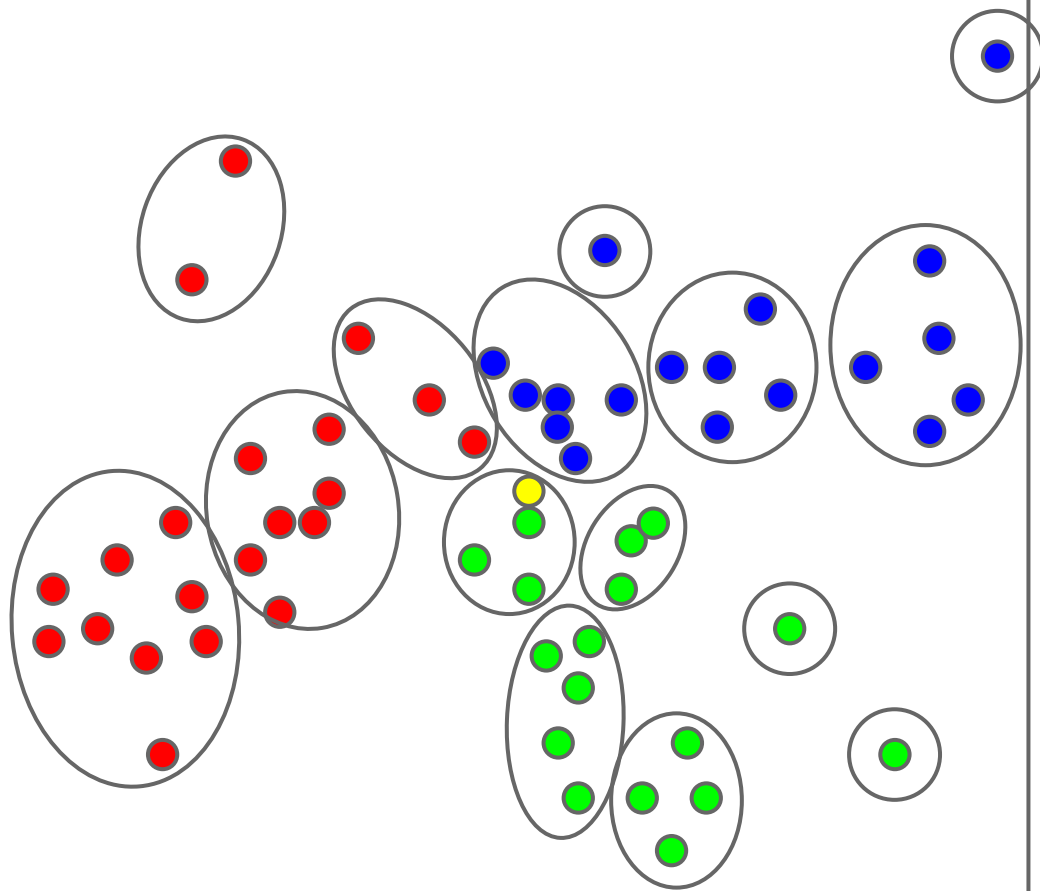
- Feature hashing

Dimensionality reduction trick for large or unbounded feature sets.

Just hash features using a standard hash function (i.e. not LSH) into e.g. 24 bit codes.

Train on these instead of original features.

Turns out the effect of collisions is negligible!



Cluster-based kNN

- Basic idea

Start by clustering data into many clusters.

Use these as a simplified representation of the data.

- Needs a fast clustering algorithm

k-means++, k-means#, k-means||
Streaming algorithms

All have time vs storage vs accuracy tradeoffs
of their own.

Streaming methods a good idea if training data
will grow over time.

Cluster-based kNN

- Search strategy 1

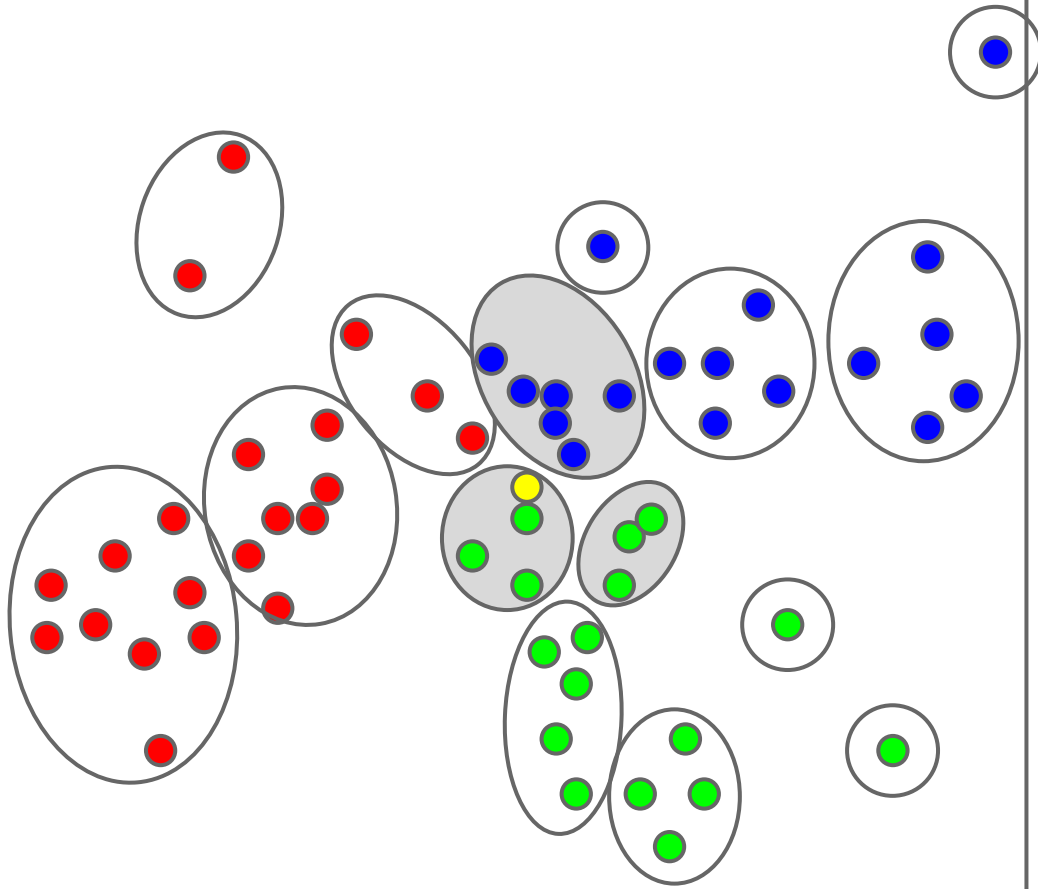
Let c = number of clusters.

Find q nearest cluster centroids to query item:
 $O(m * c)$

This gives expected $r = qn/c$ results.

Search these for k nearest neighbours to query item: $O(m * r)$

Use these to predict query item label.



Cluster-based kNN

- Search strategy 2

During clustering step, label clusters with majority class label, or weighted labels.

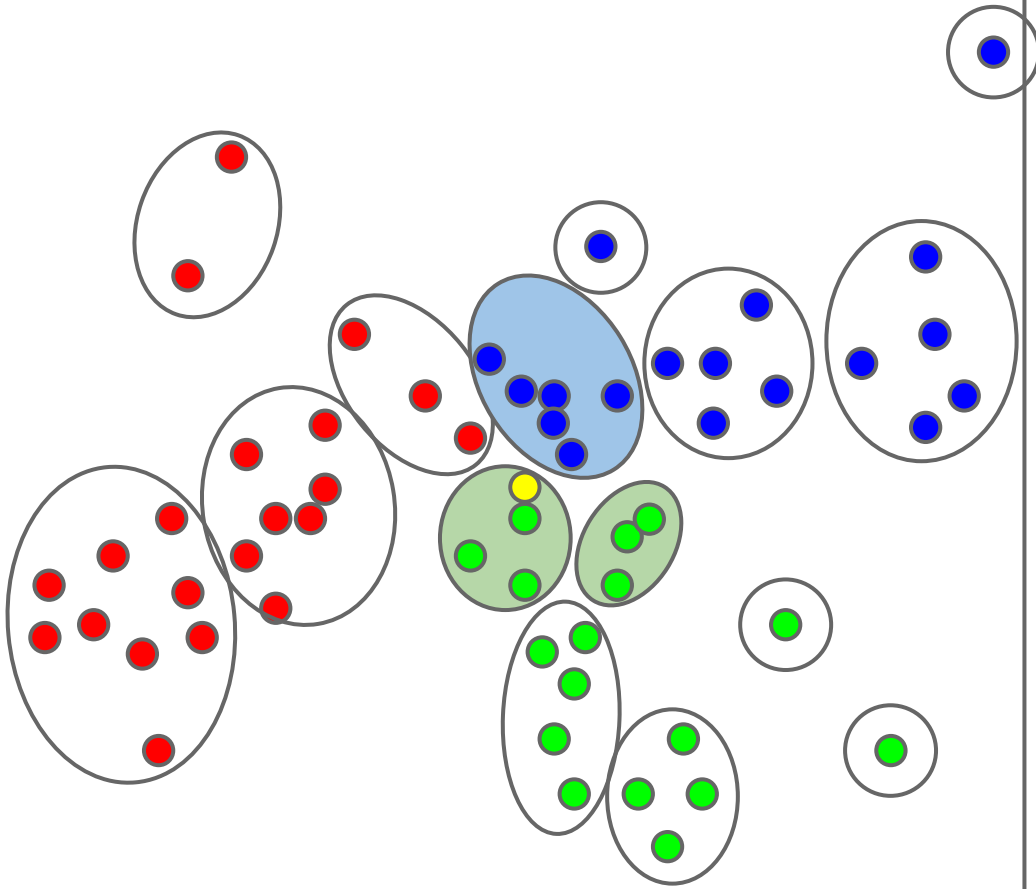
Let c = number of clusters again.

Find q nearest cluster centroids to query item:
 $O(m * c)$

Use their labels to predict query item label.

Query time not actually affected by amount of data! But clustering will take longer.

Accuracy will be damaged by poor-quality clustering.



A
a
aa
aal
aalii
aam
Aani
aardvark
aardwolf
Aaron
Aaronic
Aaronical
Aaronite
Aaronitic
Aaru
Ab
aba
Ababdeh
Ababua
...

kNN for text classification

Certain properties make distance measures for text difficult.

- Text is very sparse

[Edinburgh Twitter Corpus](#):
31M distinct tokens.

Average length of a tweet ([OUP blog](#)):
15 words.

- Importance of features very variable

If two tweets contain "everyone" this tells you very little about their similarity.

But if two tweets contain "hadoop" this tells you much more about how similar they are.



elasticsearch.



kNN for text classification

- Suggestion

Use a search engine for neighbour finding.

Index all training docs, then use *more like this* query to find most similar docs to query doc.

- Advantages

Takes advantage of sparsity automatically:
only docs with 1+ shared term considered.

Accounts for varying importance of words:
TFIDF weighting.

Operational advantages: caches, parallelism,
bloom filters, proper APIs.

Mature, well-optimized code.



elasticsearch.



kNN for text classification

- Bonus features

Many useful text processing and IR functions:

Tokenization

Stemming

Stopword removal

Word and character n-grams

Synonym expansion

Unicode handling

Fuzzy term matching

Boosting/reranking

...

Pluggable similarity functions if TFIDF cosine model isn't working for you.

Can be used for non-text data too.



elasticsearch.



kNN for text classification

- Time complexity of queries?

Not well documented in a formal sense.

Perhaps because there are so many variables:

Number of docs in index

Number of terms in index

Number of terms in query

Number of docs matching each term

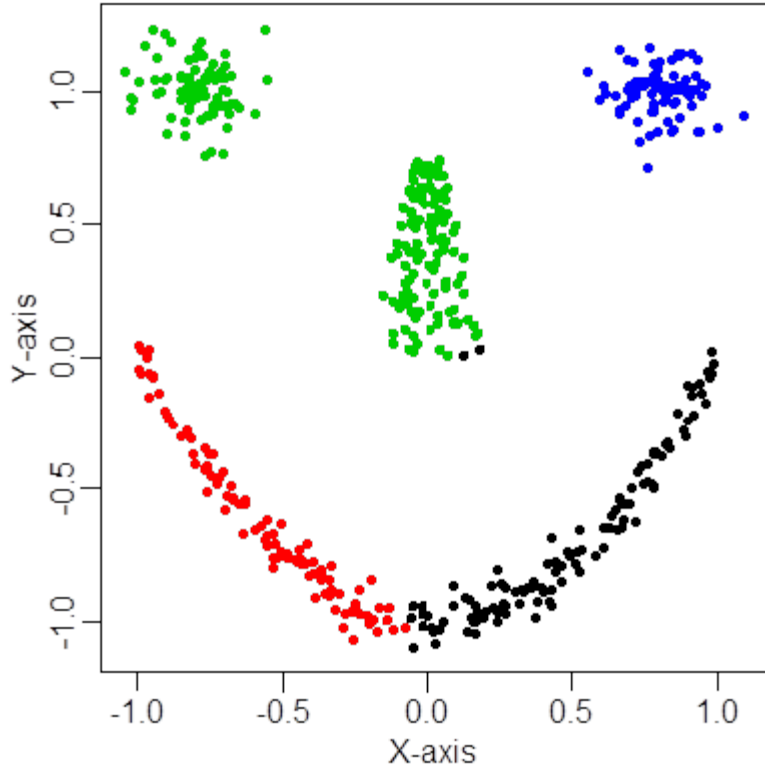
How many segments (sub-indices) the index consists of

Which on-disk data structures the index is configured to use

...

Total docs matched probably most important.

Happily scales to millions of docs and tens of thousands of terms, on modest hardware.



Thanks!

Any questions?

More from me:

https://twitter.com/andrew_clegg

<https://github.com/andrewclegg>

<https://github.com/pearson-enabling-technologies>

Smiley face dataset from R's [mlbench](#) package,
via [University of Montana](#).

Template for diagrams

