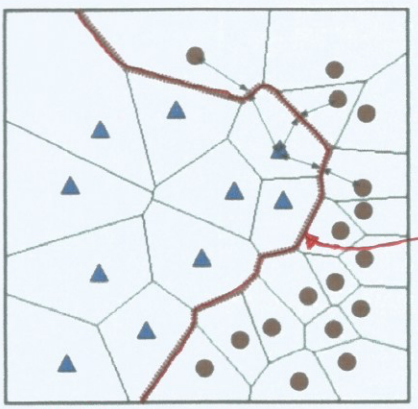
NEAREST NEIGHBOURS

VORONOI TESSELATION

→ is a partition of a plane into regions close to each of a given set of objects

These regions are called Voronoi cells



1-NN example

→ For each point there is a corresponding region consisting of all points of the plane closer to that point than to any other

DECISION BOUNDARY — Non-linear

└ Points at same distance from 2 diff. training examples

→ Notice that algo. is sensitive to outliers

└ Single mislabeled example dramatically changes boundary

→ No notion of a prior

└ No notion tht. one class may be more frequent than another class

SOLN: — Use more than one nearest neighbour to make decision

[k-NN where $k > 1$]

k-NN CLASSIFICATION ALGO.

→ No training phase; the algo. gets the training set & testing set at the same time

→ Given the training examples $\{x_i, y_i\}$ and a testing point x:

1) Compute distance $D(x, x_i)$ to every training example x_i

2) Select k closest instances $x_{i1} \cdots x_{ik}$ and their labels $y_{i1} \cdots y_{ik}$

3) Output the class y^* which is most frequent in $y_{i1} \cdots y_{ik}$

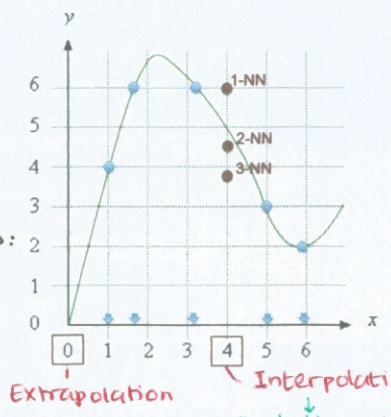
k-NN REGRESSION ALGO.

→ The y_i will now be a real-valued target

→ Algo. is similar to k-NN classification algo. but for step 3:

3) Output the mean of $y_{i1}, \ldots, y_{ik}$: $\hat{y} = \frac{1}{k} \sum_{j=1}^{k} y_{ij}$

Instead of taking the most frequent class, we'll avg. the labels



Interpolation: predicting a value for a point which is within the range of the training examples

SELECTING VALUE FOR K

→ If value of k is too high, all the instances will be classified in the most probable class

→ If too low, the DB is unstable — small changes to training set causes large changes in classification

→ To select k, try different values of k and pick the k that performs best on the validation set — If we use training set, we'd always get $k=1$

PRACTICAL ISSUES

1) Resolving ties

→ Ties occur when there are equal no. of votes for 2 diff. classes

→ To break ties, you can — do it randomly (flip coin)
- use the prior (pick class w/ greater prior)
- use 1-NN classifier to decide

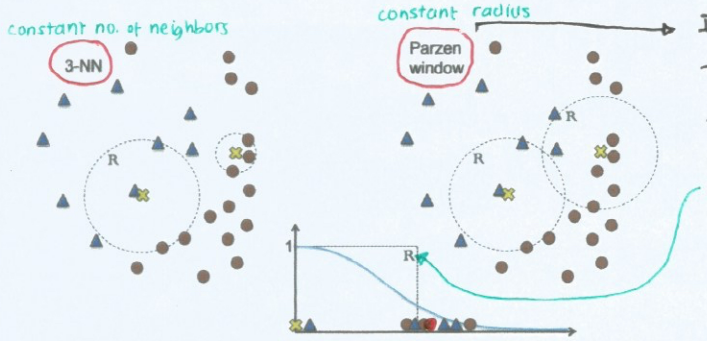
2) Missing values

→ have to fill in, otherwise can't compute distance

→ The value shld. affect distance as possible

→ can use avg. value across entire dataset

KERNELISED NN METHOD



In the Parzen version of NN, we consider the training instances that fall in a fixed region around the testing instance

This radius R is adding some variance to our classification.
i.e. If we make R a little bit smaller, we have to break a tie and if we make it a bit bigger, the point is now classified red.

↳ Hence we replace the 'boxy' fn w/ a fn tht. is smoother [The blue line]

↳ KERNEL FN — Give more weights to neighbors closer to the test pt.

↳ In kernelised NN method, every training instance contributes to the answer in proportion to the kernel value for that training instance & the testing instance

↳ Removes the said variance

$$f(x) = \text{sign}\left[\sum_i Y_i \cdot \underbrace{1_{\|x_i - x\| \le R}}_{\text{Kernelised NN}}\right] = \text{sign}\left[\sum_i Y_i \cdot \underbrace{K(x_i, x)}_{\text{Kernelised NN}}\right]$$

→ similar form to kernelised SVM: $\text{sign}[\sum_i \alpha_i Y_i K(x_i, x)]$
The difference lies in α_i

PROS & CONS OF k-NN

(+)	(−)
→ Almost no assumptions abt. the data • assumptions implied by the distance fn (only locally!) • non-parametric approach ↳ not fitting any distribution to the data	→ Need to handle missing values → Sensitive to outliers → Sensitive to lots of irrelevant attributes ↳ Like NB and unlike DTs → Computationally expensive • need to store ALL training examples • need to compute distance to ALL examples $O(nd)$ — d: cost of computing distance; n: no. of training examples • $\uparrow n$ — system will become slower

MAKING k-NN FASTER

→ Training takes $O(d)$ time but testing takes $O(nd)$, hence we can:

1) **Reduce d** through dimensionality reduction

2) **Reduce n** by not comparing to all training examples

↳ IDEA — Quickly identify $m \ll n$ potential nearest neighbors

↳ **K-D trees** work if you are working w/ low-dimensional, real-valued data
- $O(d \log_2 n)$, only works when $d \ll n$ and can miss neighbors

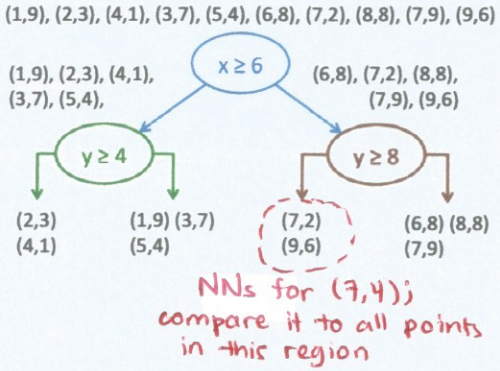
↳ **Inverted lists** work w/ high-dimensional, discrete (sparse) data
- $O(n'd')$ where $d' \ll d$, $n' \ll n$ and you don't miss neighbors

↳ **Locality-sensitive hashing** works w/ high-dimensional, real-valued/discrete data
- $O(n'd)$ where $n' \ll n$ and can miss neighbors

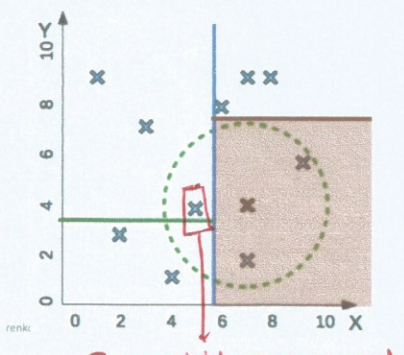
K-D TREES

→ Pick random dimension
↓
Find median
↓
Split data
↺ Repeat

e.g. Testing pt. is (7,4)



NNs for (7,4); compare it to all points in this region



This neighbor is missed

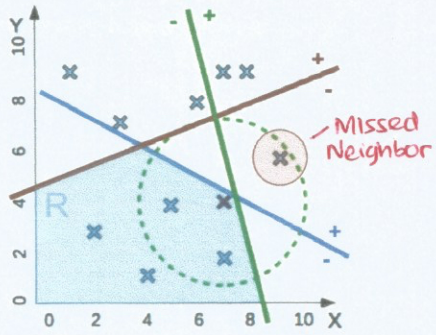
LOCALITY-SENSITIVE HASHING (LSH)

→ Start by drawing random hyperplanes $h_1 \dots h_k$

↳ Space will be sliced into 2^k regions (Polytopes)

→ Compare testing pt. only to training pts. within the same region.

→ Complexity is $O(\underbrace{kd}_{\text{To find the region R}} + \underbrace{d\frac{n}{2^k}}_{\text{Comparison to } \frac{n}{2^k} \text{ points in R}}) \approx O(d \log_2 n)$



INVERTED LIST

→ Is a data struct. used by search engines

→ List all training e.g. that contain particular attribute

→ ASSUMPTION: most attribute values are zero (Sparseness)

→ $O(d\sqrt{n})$

D1: "send your password"	spam
D2: "send us review"	ham
D3: "send us password"	spam
D4: "send us details"	ham
D5: "send your password"	spam
D6: "review your account"	ham

new email: "account review"

↳ Given a new testing example, merge inverted lists for attributes present

send → 1 2 3 4 5

your → 1 5 6

review → 2 6

account → 6

password → 1 3 5

No chance that any NNs will be missed