Nonparametric Modelling

COMP9417 Machine Learning and Data Mining

Term 2, 2022

Acknowledgements

Material derived from slides for the book
"Machine Learning" by T. Mitchell
McGraw-Hill (1997)
`http://www-2.cs.cmu.edu/~tom/mlbook.html`

Material derived from slides by Andrew W. Moore
`http:www.cs.cmu.edu/~awm/tutorials`

Material derived from slides by Eibe Frank
`http://www.cs.waikato.ac.nz/ml/weka`

Material derived from slides for the book
"Machine Learning" by P. Flach
Cambridge University Press (2012)
`http://cs.bris.ac.uk/~flach/mlbook`

Aims

This lecture will enable you to describe two kinds of approach typically referred to in machine learning as *nonparametric*. The first is tree learning, and the second is known as “nearest neighbour”. Both of these can be applied to either classification or regression tasks. Following it you should be able to:

- describe the representation of tree-structured models
- reproduce the top-down decision tree induction (TDIDT) algorithm
- define node “impurity” measures of tree learning
- describe learning regression and model trees
- describe overfitting in terms of model complexity
- describe k-nearest neighbour for classification and regression

What is Nonparametric Modelling in Machine Learning ?

Surprisingly difficult to define precisely parametric *vs.* nonparametric

- Linear models for regression and classification
    - Learning is finding good values for a fixed set of parameters
    - Parameters fixed by features in the dataset (its dimensionality)
- Other types of models do not have parameters fixed
    - Trees learning automatically selects parameters to include or leave out
    - Nearest Neighbour methods are “model-free” !
- Some more complex methods also can be viewed as nonparametric
    - Random Forests
    - Deep Learning
    - Probabilistic Programming
    - ...

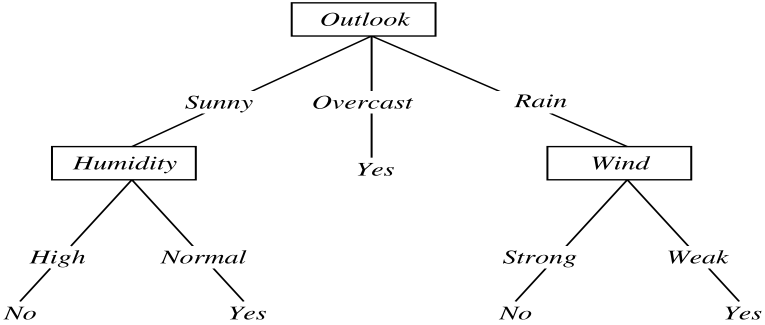
Why use decision trees?

- Trees in some form are probably still the single most popular data mining tool
  - Easy to understand
  - Easy to implement
  - Easy to use
  - Computationally efficient (even on big data) to learn and run
- They do *classification*, i.e., predict a categorical output from categorical and/or real inputs
- Tree learning can also be used for predicting a real-valued output, i.e., they can do *regression*
- There are some drawbacks, though — e.g., can have high variance

Training Examples

Day	Outlook	Temperature	Humidity	Wind	PlayTennis
D1	Sunny	Hot	High	Weak	No
D2	Sunny	Hot	High	Strong	No
D3	Overcast	Hot	High	Weak	Yes
D4	Rain	Mild	High	Weak	Yes
D5	Rain	Cool	Normal	Weak	Yes
D6	Rain	Cool	Normal	Strong	No
D7	Overcast	Cool	Normal	Strong	Yes
D8	Sunny	Mild	High	Weak	No
D9	Sunny	Cool	Normal	Weak	Yes
D10	Rain	Mild	Normal	Weak	Yes
D11	Sunny	Mild	Normal	Strong	Yes
D12	Overcast	Mild	High	Strong	Yes
D13	Overcast	Hot	Normal	Weak	Yes
D14	Rain	Mild	High	Strong	No

Decision Tree for $PlayTennis$



Decision Trees

Decision tree representation:

- Each internal node tests an attribute (feature)
- Each branch corresponds to attribute (feature) value or threshold
- Each leaf node assigns a classification value

How would we represent the following expressions ?

- $\wedge, \vee$, XOR
- M of N
- $(A \wedge B) \vee (C \wedge \neg D \wedge E)$

Decision Trees

$X \wedge Y$

```
X = t:
| Y = t: true
| Y = f: no
X = f: no
```

$X \vee Y$

```
X = t: true
X = f:
| Y = t: true
| Y = f: no
```

Decision Trees

X XOR Y

```
X = t:
| Y = t: false
| Y = f: true
X = f:
| Y = t: true
| Y = f: false
```

So decision trees are, in some sense, *non-linear* classifiers (or regressors).

Decision Trees

2 of 3

```
X = t:
| Y = t: true
| Y = f:
| | Z = t: true
| | Z = f: false
X = f:
| Y = t:
| | Z = t: true
| | Z = f: false
| Y = f: false
```

In general, decision trees represent a *disjunction of conjunctions* of constraints, or tests, on the attributes values of instances.

Top-Down Induction of Decision Trees (TDIDT)

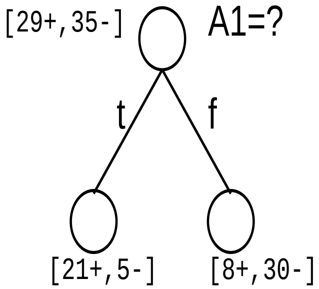
Main loop:

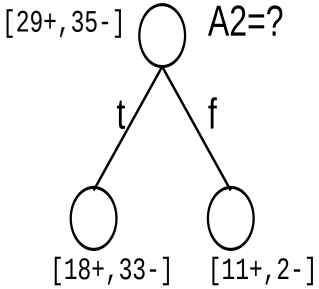
1. Select A as the "best" decision attribute for next $node$
2. Assign A as decision attribute for $node$
3. For each value of A, create new descendant of $node$
4. Sort training examples to leaf nodes
5. If training examples perfectly classified, Then STOP, Else iterate over new leaf nodes

Essentially, this is the top-level of the ID3 algorithm[1] — the first efficient symbolic Machine Learning algorithm.

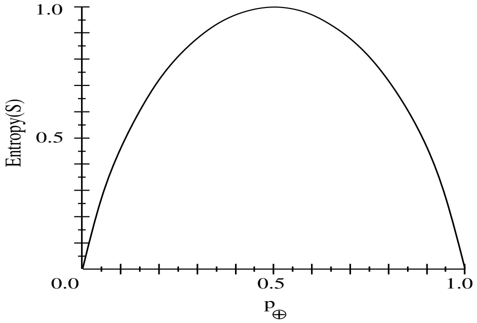
[1] See: Quinlan (1986).

Which attribute is best?





Entropy



Consider a 2-class distribution, where:

S is a sample of training examples
$p_\oplus$ is the proportion of positive examples in S
$p_\ominus$ is the proportion of negative examples in S

Entropy

Entropy measures the "impurity" of S

$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

A "pure" sample is one in which all examples are of the same class.

A decision tree node with low impurity means that the path from root to the node represents a combination of attribute-value tests with good classification accuracy.

Entropy

$Entropy(S)$ = expected number of bits needed to encode class ($\oplus$ or $\ominus$) of randomly drawn member of S (under the optimal, shortest-length code)

Why ?

Information theory: optimal length code assigns $-\log_2 p$ bits to message having probability p.

So, expected number of bits to encode $\oplus$ or $\ominus$ of random member of S:

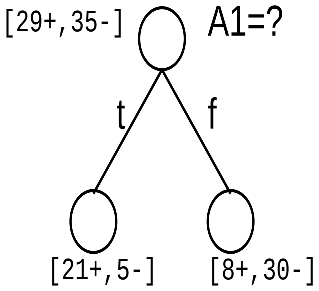
$$p_{\oplus}(-\log_2 p_{\oplus}) + p_{\ominus}(-\log_2 p_{\ominus})$$

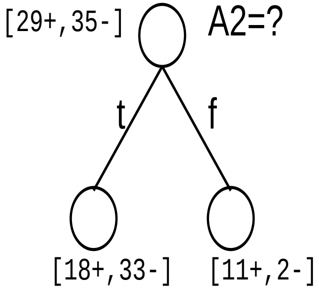
$$Entropy(S) \equiv -p_{\oplus} \log_2 p_{\oplus} - p_{\ominus} \log_2 p_{\ominus}$$

Information Gain

- $Gain(S, A)$ = expected reduction in entropy due to sorting on A

$$Gain(S, A) \equiv Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$$





Information Gain

$$\begin{aligned}
Gain(S, A1) &= Entropy(S) \; - \left(\frac{|S_t|}{|S|} Entropy(S_t) + \frac{|S_f|}{|S|} Entropy(S_f) \right) \\
&= 0.9936 \; - \\
&= ((\frac{26}{64}(-\frac{21}{26}\log_2(\frac{21}{26}) - \frac{5}{26}\log_2(\frac{5}{26}))) + \\
&\quad (\frac{38}{64}(-\frac{8}{38}\log_2(\frac{8}{38}) - \frac{30}{38}\log_2(\frac{30}{38})))) \\
&= 0.9936 \; - (\; 0.2869 + 0.4408 \;) \\
&= 0.2658
\end{aligned}$$

Information Gain

$$\begin{aligned} Gain(S, A2) &= 0.9936 \ - (\ 0.7464 + 0.0828 \ ) \\ &= 0.1643 \end{aligned}$$

Information Gain

So we choose A1, since it gives a larger expected reduction in entropy.

Attribute selection – impurity measures more generally

Estimate class probability of class k at node m of the tree as $\hat{p}_{mk} = \frac{|S_{mk}|}{|S_m|}$.

Classify at node m by predicting the majority class, $\hat{p}_{mk}(m)$.

Misclassification error:

$$1 - \hat{p}_{mk}(m)$$

Entropy for K class values:

$$-\sum_{k=1}^{K} \hat{p}_{mk} \log \hat{p}_{mk}$$

CART (Breiman et al. (1984)) uses the “Gini index”:

$$\sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk})$$

Attribute selection – impurity measures more generally

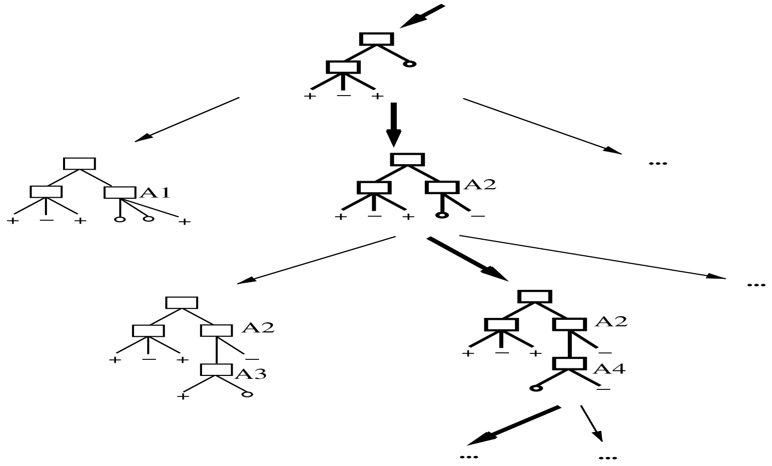
Why not just use accuracy, or misclassification error ?

In practice, not found to work as well as others

Entropy and Gini index are more sensitive to changes in the node probabilities than misclassification error.

Entropy and Gini index are differentiable, but misclassification error is not (Hastie et al. (2009)).

TDIDT search space is all trees !



Applies greedy search to maximise information gain . . .

Inductive Bias of TDIDT

Note hypothesis space H is complete (contains all finite discrete-valued functions w.r.t attributes)

- So H can represent the power set of instances X !

$\rightarrow$Unbiased?

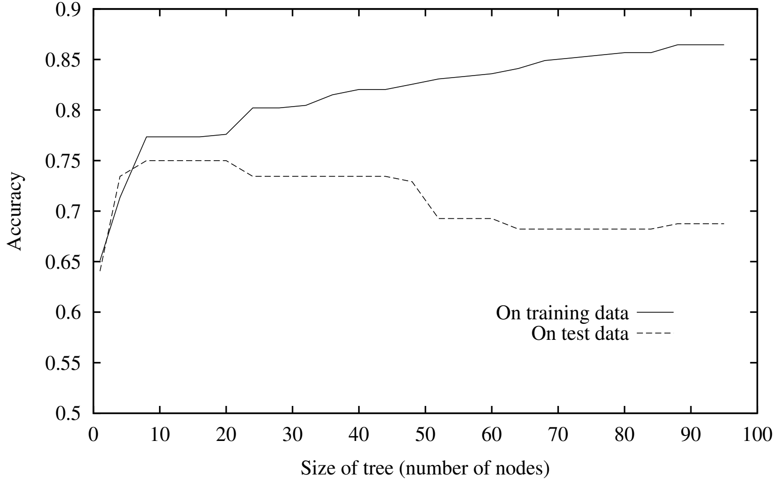
Not really...

- Preference for short trees, and for those with high information gain attributes near the root
- Inductive bias is a *preference* for some hypotheses, rather than a *restriction* of hypothesis space H
- An incomplete search of a complete hypothesis space *versus* a complete search of an incomplete hypothesis space
- Occam's razor: prefer the shortest hypothesis that fits the data
- Inductive bias: approximately, "prefer shortest tree"

Why does overfitting occur?

- Greedy search can make mistakes. It can end up in local minima — so a sub-optimal choice earlier might result in a better solution later (*i.e.*, pick a test whose information gain is less than the best one)
- But there is also another kind of problem. Training error is an optimistic estimate of the true error of the model, and this optimism increases as the training error decreases
    - Suppose we could quantify the "optimism" of a learning algorithm . . .
    - Say we have two models h_1 and h_2 with training errors e_1 and e_2 and optimism o_1 and o_2.
    - Let the true error of each be $E_1 = e_1 + o_1$ and $E_2 = e_2 + o_2$
    - If $e_1 < e_2$ and $E_1 > E_2$, then we will say that h_1 has overfit then training data
- So, a search method based purely on training data estimates may end up overfitting the training data

Overfitting in Decision Tree Learning



Avoiding Overfitting

How can we avoid overfitting?

For tree learning the answer is **pruning**

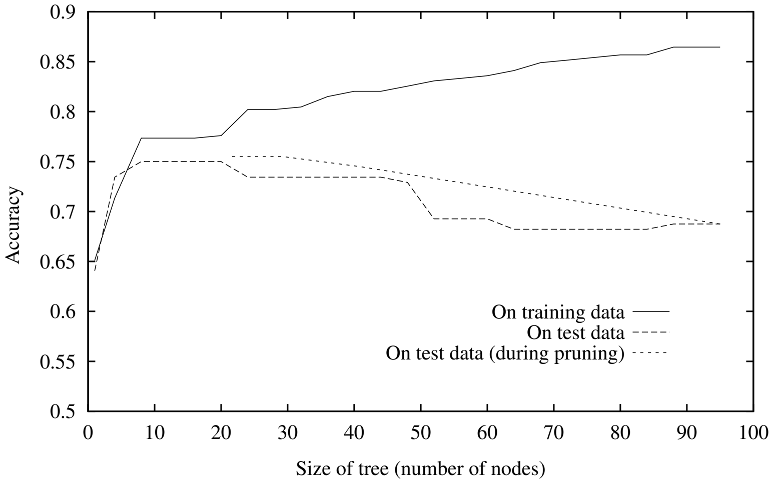
Two main approaches

- **pre-pruning** stop growing when further data splits are not useful
- **post-pruning** grow full tree, then remove sub-trees which may be overfitting

Post-pruning more common:

- can prune using a *pessimistic* estimate of error
  - measure training set error
  - adjust by a factor dependent on "confidence" hyperparameter
- or can use cross-validation to estimate error during pruning

Avoiding Overfitting – Post-pruning

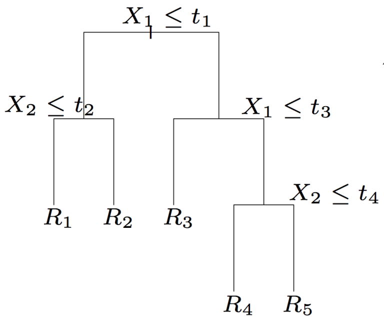


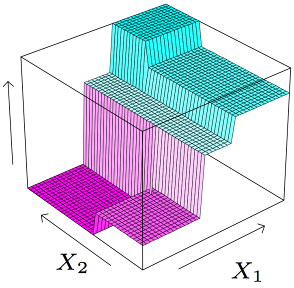
Effect of reduced-error pruning on tree learning to avoid overfitting.

Regression trees

- Differences to decision trees:
  - Splitting criterion: minimizing intra-subset variation
  - Pruning criterion: based on numeric error measure
  - Leaf node predicts average class values of training instances reaching that node
- Can approximate piecewise constant functions
- Easy to interpret
- More sophisticated version: model trees

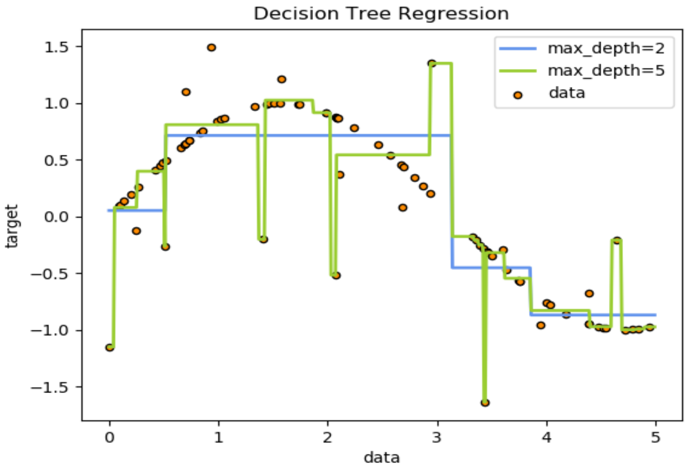
A Regression Tree and its Prediction Surface



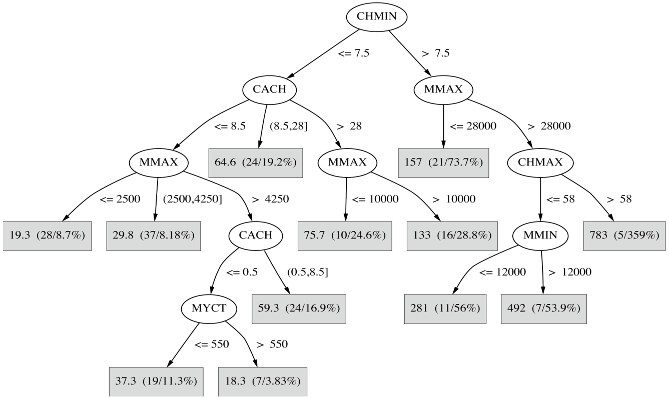


“Elements of Statistical Learning” Hastie, Tibshirani & Friedman (2001)

Regression Tree on sine dataset



Regression Tree on CPU dataset



Tree learning as variance reduction

- Variance of a Boolean (i.e., Bernoulli) variable with success probability p is $p(1-p)$.
- Can interpret goal of tree learning as minimising the class variance in the leaves.
- In regression problems we can define the variance in the usual way:

$$\mathrm{Var}(Y) = \frac{1}{|Y|} \sum_{y \in Y} (y - \overline{y})^2$$

If a split partitions the set of target values Y into mutually exclusive sets $\{Y_1, \ldots, Y_l\}$, the weighted average variance is then

$$\mathrm{Var}(\{Y_1, \ldots, Y_l\}) = \sum_{j=1}^{l} \frac{|Y_j|}{|Y|} \mathrm{Var}(Y_j) = \ldots = \frac{1}{|Y|} \sum_{y \in Y} y^2 - \sum_{j=1}^{l} \frac{|Y_j|}{|Y|} \overline{y}_j^2$$

The first term is constant for a given set Y and so we want to maximise the weighted average of squared means in the children.

Learning a regression tree

Imagine you are a collector of vintage Hammond tonewheel organs. You have been monitoring an online auction site, from which you collected some data about interesting transactions:

#	Model	Condition	Leslie	Price
1.	B3	excellent	no	4513
2.	T202	fair	yes	625
3.	A100	good	no	1051
4.	T202	good	no	270
5.	M102	good	yes	870
6.	A100	excellent	no	1770
7.	T202	fair	no	99
8.	A100	good	yes	1900
9.	E112	fair	no	77

Learning a regression tree

From this data, you want to construct a regression tree that will help you determine a reasonable price for your next purchase.
There are three features, hence three possible splits:

$\text{Model} = [\text{A100}, \text{B3}, \text{E112}, \text{M102}, \text{T202}]$
$[1051, 1770, 1900][4513][77][870][99, 270, 625]$
$\text{Condition} = [\text{excellent}, \text{good}, \text{fair}]$
$[1770, 4513][270, 870, 1051, 1900][77, 99, 625]$
$\text{Leslie} = [\text{yes}, \text{no}]$ $[625, 870, 1900][77, 99, 270, 1051, 1770, 4513]$

The means of the first split are 1574, 4513, 77, 870 and 331, and the weighted average of squared means is $3.21 \cdot 10^6$. The means of the second split are 3142, 1023 and 267, with weighted average of squared means $2.68 \cdot 10^6$; for the third split the means are 1132 and 1297, with weighted average of squared means $1.55 \cdot 10^6$. We therefore branch on Model at the top level. This gives us three single-instance leaves, as well as three A100s and three T202s.

Learning a regression tree

For the A100s we obtain the following splits:

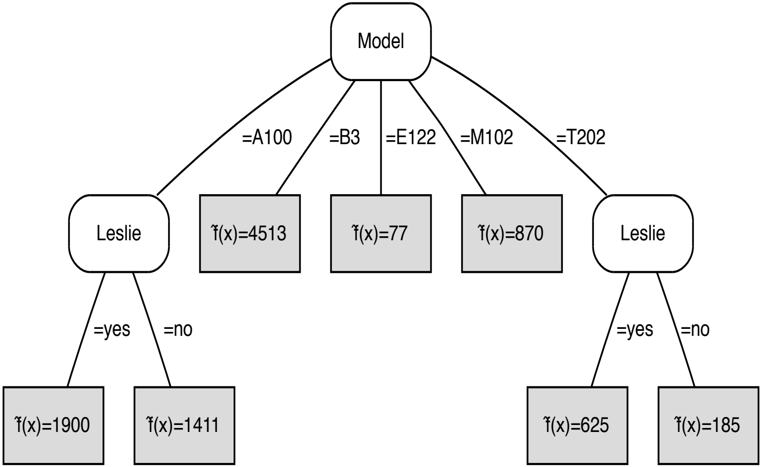
$\text{Condition} = [\text{excellent}, \text{good}, \text{fair}]$ $[1770][1051, 1900][]$
$\text{Leslie} = [\text{yes}, \text{no}]$ $[1900][1051, 1770]$

Without going through the calculations we can see that the second split results in less variance (to handle the empty child, it is customary to set its variance equal to that of the parent). For the T202s the splits are as follows:

$\text{Condition} = [\text{excellent}, \text{good}, \text{fair}]$ $[][270][99, 625]$
$\text{Leslie} = [\text{yes}, \text{no}]$ $[625][99, 270]$

Again we see that splitting on Leslie gives tighter clusters of values. The learned regression tree is depicted on the next slide.

A regression tree



A regression tree learned from the Hammond organ dataset.

Model trees

- Like regression trees but with linear regression functions at each node
- Linear regression applied to instances that reach a node after full tree has been built
- Only a subset of the attributes is used for LR
    - Attributes occurring in subtree (+maybe attributes occurring in path to the root)
- Fast: overhead for Linear Regression (LR) not large because usually only a small subset of attributes is used in tree

Two uses of features (Flach (2012))

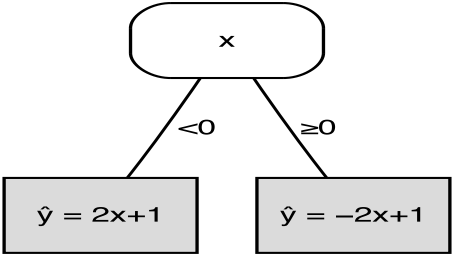
Suppose we want to approximate $y = \cos \pi x$ on the interval $-1 \leq x \leq 1$.

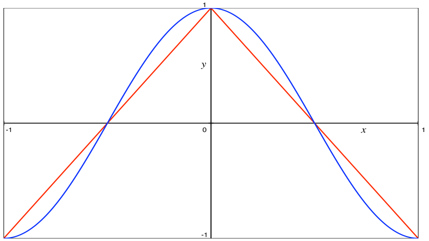
A linear approximation is not much use here, since the best fit would be $y = 0$.

However, if we split the x-axis in two intervals $-1 \leq x < 0$ and $0 \leq x \leq 1$, we could find reasonable linear approximations on each interval.

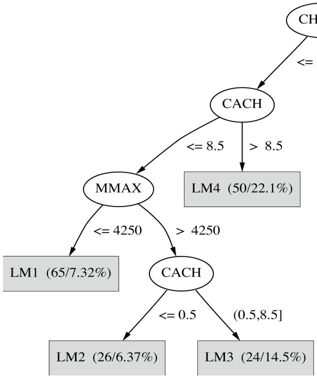
We can achieve this by using x both as a splitting feature and as a regression variable (next slide).

A small model tree





Model Tree on CPU dataset

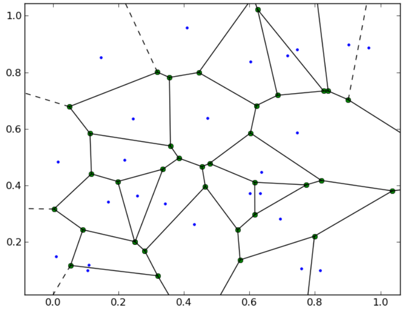


Nearest neighbour classification

Nearest neighbour classification

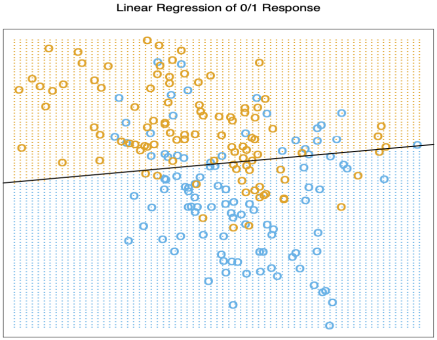
- Related to the simplest form of learning: rote learning or memorization
    - Training instances are searched for instance that **most closely resembles** new or *query* instance
    - The *instances* themselves represent the knowledge
    - Called: *instance-based*, *memory-based* learning or *case-based* learning; often a form of *local* learning
- The *similarity* or *distance* function defines "learning", i.e., how to go beyond simple memorization
- Intuitive idea — instances "close by", i.e., neighbours or *exemplars*, should be classified similarly
- Instance-based learning is *lazy* learning
- Methods: *nearest-neighbour*, *k-nearest-neighbour*, *lowess*, . . .
- Ideas also important for *unsupervised* methods, e.g., clustering

Nearest neighbour classification



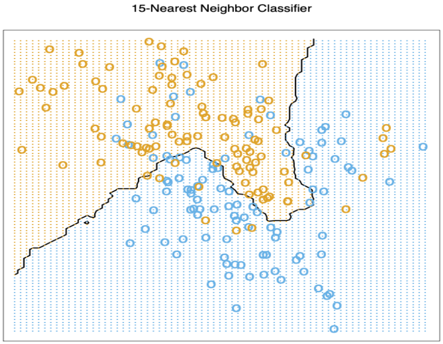
Nearest neighbour is a classification (or regression) algorithm that predicts whatever is the output value of the nearest data point to some query point.

Why use Nearest Neighbour ?



A 2-dimensional data set where the problem is to learn a classifier separating the blue = 0 class from the orange = 1 class. Coloured circles denote data points, shaded areas denotes classifications.

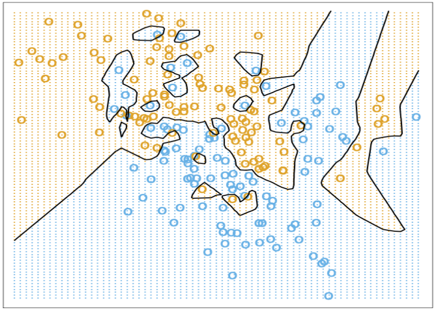
Why use Nearest Neighbour ?



Same problem as previous slide, classification using k-NN where $k = 15$.

Why use Nearest Neighbour ?

1-Nearest Neighbor Classifier



Same problem as previous slide, classification using k-NN where $k = 1$.

Minkowski distance

Minkowski distance If $\mathcal{X} = \mathbb{R}^d$, the *Minkowski distance* of order $p > 0$ between points $\mathbf{x}, \mathbf{y} \in \mathcal{X}$ is defined as

$$\mathrm{Dis}_p(\mathbf{x}, \mathbf{y}) = \left(\sum_{j=1}^{d} |x_j - y_j|^p \right)^{1/p} = ||\mathbf{x} - \mathbf{y}||_p$$

where $||\mathbf{z}||_p = \left(\sum_{j=1}^{d} |z_j|^p \right)^{1/p}$ is the p-*norm* (sometimes denoted L_p or l_p norm) of the vector $\mathbf{z}$.

Note: sometimes p is omitted when writing the norm, often when $p = 2$.

Minkowski distance

- The 2-norm refers to the familiar *Euclidean distance*

$$\mathrm{Dis}_2(\mathbf{x},\mathbf{y}) = \sqrt{\sum_{j=1}^{d}(x_j - y_j)^2} = \sqrt{(\mathbf{x}-\mathbf{y})^{\mathrm{T}}(\mathbf{x}-\mathbf{y})}$$

  which measures distance 'as the crow flies'.
- The 1-*norm* denotes *Manhattan distance*, also called *cityblock distance*:

$$\mathrm{Dis}_1(\mathbf{x},\mathbf{y}) = \sum_{j=1}^{d}|x_j - y_j|$$

  This is the distance if we can only travel along coordinate axes.

Nearest Neighbour

Stores all training examples $\langle x_i, f(x_i) \rangle$.

Nearest neighbour:

- Given query instance x_q, first locate nearest training example x_n, then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

k-Nearest neighbour:

- Given x_q, take vote among its k nearest neighbours (if discrete-valued target function)
- take mean of f values of k nearest neighbours (if real-valued)

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

k-Nearest Neighbour (kNN) Algorithm

Training algorithm:

- For each training example $\langle x_i, f(x_i) \rangle$, add the example to the list *training_examples*.

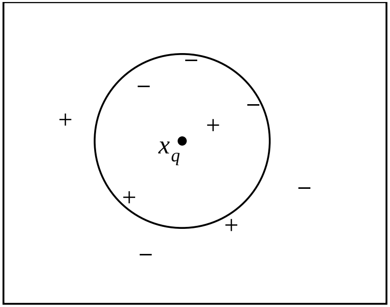
Classification algorithm:

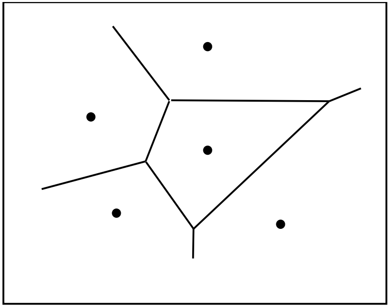
- Given a query instance x_q to be classified,
    - Let $x_1 \dots x_k$ be the k instances from *training_examples* that are *nearest* to x_q by the distance function
    - Return

$$\hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} I[v, f(x_i)]$$

where $I[a, b] = 1$ if $a = b$ and 0 otherwise.

Nearest Neighbour is “model-free”





2 classes, $+$ and $-$ and query point x_q. On left, note effect of varying k.
On right, $1-$NN induces a Voronoi tessellation of the instance space.
Formed by the perpendicular bisectors of lines between points.

Normalization and other issues

- Different attributes measured on different scales
- Need to be *normalized* (why ?)

$$a_r = \frac{v_r - \mathsf{min} v_r}{\mathsf{max} v_r - -\mathsf{min} v_r}$$

where v_r is the actual value of attribute r

- Nominal attributes: distance either 0 or 1
- Common policy for missing values: assumed to be maximally distant (given normalized attributes)

When To Consider Nearest Neighbour

- Instances map to points in $\mathbb{R}^d$
  - or can define a meaningful distance measure on features
- Low-dimensional data, say, less than 20 features per instance
  - or number of features can be reduced ...
- Lots of training data
- No requirement for "explanatory" model to be learned

When To Consider Nearest Neighbour

Advantages:

- Statisticians have used k-NN since early 1950s
- Can be very accurate
    - at most twice the "Bayes error" for 1-NN[2]
- Training is very fast
- Can learn complex target functions
- Don't lose information by generalization - keep all instances

[2]See Hastie et al. (2009).

When To Consider Nearest Neighbour

Disadvantages:

- Slow at query time: basic algorithm scans entire training data to derive a prediction
- “Curse of dimensionality”
- Assumes all features are equally important, so easily fooled by irrelevant features
    - Remedy: feature selection or weights
- Problem of noisy instances:
    - Remedy: remove from data set
    - not easy – how to know which are noisy ?

When To Consider Nearest Neighbour

What is the inductive bias of k-NN ?

- an assumption that the classification of query instance x_q will be most similar to the classification of other instances that are nearby according to the distance function

Nearest-neighbour classifier

- kNN uses the training data as exemplars, so training is $O(n)$, but prediction is also $O(n)$!
- 1NN perfectly separates training data, so low bias but high variance[3]
- By increasing the number of neighbours k we increase bias and decrease variance (what happens when $k = n$?)
- Easily adapted to real-valued targets, and even to structured objects (nearest-neighbour retrieval). Can also output probabilities when $k > 1$
- Warning: in high-dimensional spaces everything is far away from everything and so pairwise distances are uninformative (curse of dimensionality)

[3]See Hastie et al. (2009).

Local (nearest-neighbour) regression

Nearest neighbour for numeric prediction

Store all training examples $\langle x_i, f(x_i) \rangle$.

Nearest neighbour:

- Given query instance x_q,
- first locate nearest training example x_n,
- then estimate $\hat{y} = \hat{f}(x_q) = f(x_n)$
- k-Nearest neighbour:
- Given x_q, take mean of f values of k nearest neighbours

$$\hat{y} = \hat{f}(x_q) = \frac{\sum_{i=1}^{k} f(x_i)}{k}$$

Local regression

Use kNN to form a local approximation to f for each query point x_q using a linear function of the form

$$\hat{f}(x) = b_0 + b_1 x_1 + \ldots + b_d x_d$$

where x_i denotes the value of the ith feature of instance x.
Where does this linear regression model come from ?

- fit linear function to k nearest neighbours
- or quadratic or higher-order polynomial . . .
- produces "piecewise approximation" to f

Distance-Weighted kNN

- Might want to weight nearer neighbours more heavily ...
- Use distance function to construct a weight w_i
- Replace the final line of the classification algorithm by:

$$\hat{f}(x_q) \leftarrow \arg\max_{v \in V} \sum_{i=1}^{k} w_i I[v, f(x_i)]$$

where

$$w_i \equiv \frac{1}{\mathrm{Dis}(x_q, x_i)}$$

and $\mathrm{Dis}(x_q, x_i)$ is distance between x_q and x_i, such as Euclidean distance.

Distance-Weighted kNN

For real-valued target functions replace the final line of the algorithm by:

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

(denominator normalizes contribution of individual weights).

Now we can consider using *all* the training examples instead of just k

$\rightarrow$ using all examples (i.e., when $k = n$) with the rule above is called "Shepard's method"

Evaluation

Lazy learners do not construct an explicit model, so how do we evaluate the output of the learning process ?

- 1-NN – training set error is always zero !
    - each training example is always closest to itself
- k-NN – overfitting may be hard to detect

Use leave-one-out cross-validation (LOOCV) – leave out each example and predict it given the rest:

$$(x_1, y_1), (x_2, y_2), \ldots, (x_{i-1}, y_{i-1}), (x_{i+1}, y_{i+1}), \ldots, (x_n, y_n)$$

Error is mean over all predicted examples. Fast — no models to be built !

Curse of Dimensionality

Bellman (1960) coined this term in the context of dynamic programming.

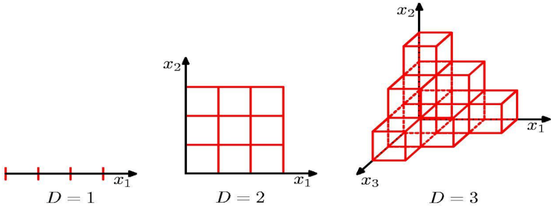
Imagine instances described by 20 features, but only 2 are relevant to target function — “similar” examples will appear “distant”.

Curse of dimensionality: nearest neighbour is hard for high-dimensional instances x.

One approach:

- feature weighting where jth feature values are multiplied by weight z_j, where $z_1, \ldots, z_d$ are chosen to minimize prediction error
- Use cross-validation to automatically choose weights $z_1, \ldots, z_d$
- Note: setting z_j to zero eliminates this dimension altogether

Curse of Dimensionality



- number of “cells” in the instance space grows exponentially in the number of features
- with exponentially many cells we would need exponentially many data points to ensure that each cell is sufficiently populated to make nearest-neighbour predictions reliably

Summary

- Nonparametric models essentially take a more “flexible” view of modelling data for classification or regression tasks
    - For example, providing the ability to learn non-linear models
- Tree learning is a practical method for many tasks, widely used
- Nearest neighbour for classification and regression based on distance in feature space from nearest training example
- Both kinds of approach also provide additional perspectives on model complexity

Breiman, L., Friedman, J. H., Olshen, R. A., and Stone, C. J. (1984). *Classification and Regression Trees*. Wadsworth, Belmont.

Flach, P. (2012). *Machine Learning*. Cambridge University Press.

Hastie, T., Tibshirani, R., and Friedman, J. (2009). *The Elements of Statistical Learning*. Springer, 2nd edition.

Quinlan, J. R. (1986). Induction of decision trees. *Machine Learning*, 1(1):81–106.