Remainder: Classification

- Multiple inputs $x$ (can be continuous, discrete or both).
- Single discrete output $y$.
- Goal: predict output on future unseen inputs.
  Issues: model complexity, overfitting.
- From a probabilistic point of view, we are using Bayes rule:
  \[ p(y|x) = \frac{p(x|y)p(y)}{p(x)} = \frac{p(x|y)p(y)}{\sum_{y'} p(x|y')p(y')} \]

Voronoi Tessellation, Decision Surfaces

- For continuous inputs, we can view the problem as one of segmenting the input space into regions which belong to a single class, i.e. constant output.
- Such a segmentation is the “Voronoi tessellation” for our classifier.
- The boundaries between regions are the “decision surfaces”.
- Training a classifier == defining decision surfaces.

Probabilistic Model, Bayes Error Rate

- Model original data as coming from joint pdf $p(x, y)$.
  Classification == trying to learn conditional density $p(y|x)$.
- Even if we get the perfect model, our error rate may not be zero. Why? Classes may overlap.
- The best we could ever do if our cost function is number of errors is to guess $y^* = \arg\max_y p(y|x)$.
  (The error rate of this procedure is known as the “Bayes error”.)
Finally: a real algorithm!

To classify a test point, choose the most common class amongst its $K$ nearest neighbours in the training set.

**Algorithm K-NN**

c-test ← KNN(K, x-train, c-train, x-test) 

\[
d(m,n) = \text{distance between } x \text{-} \text{train}(m) \text{ and } x \text{-} \text{test}(n) \\
\text{n(n,l)} = \text{index of l-th smallest entry of } d(:,n) \text{ [*]} \\
c(n,l) = c \text{-} \text{train(n(n,l))} \\
c \text{-} test(n) = \text{most common value in } c(n,1:K) \text{ [**]} 
\]

- If ties at * when $l = K$, increase $K$ for that $n$ only.
- If ties at **, decrease $K$ for that $n$ only.
- confidence $\approx$ (#votes for class) / $K$

**Error Bounds for NN**

- Amazing fact: asymptotically, $\text{err}(1\text{-NN}) \leq 2 \text{err(Bayes)}$:

\[
e_B \leq e_{1,NN} \leq 2e_B - \frac{M}{M-1}e_B^2
\]

this is a tight upper bound, achieved in the “zero-information” case when the classes have identical densities.

- For K-NN there are also bounds. e.g. for two classes and odd $K$:

\[
e_B \leq e_{K,NN} \leq \sum_{i=0}^{(K-1)/2} \binom{k}{i} \left[e_B^{i+1}(1-e_B)^{k-i} + e_B^{k-i}(1-e_B)^{i+1}\right]
\]

**More on K-NN**

- Typical distance = squared Euclidean $d(m,n) = \sum d(x_d^m - x_d^n)^2$
- If Euclidean distance is used, decision surfaces are piecewise linear.
- Trick: remember the $K$-th smallest distance so far, and break out of the summation over dimensions if you exceed it.
- In low-d with lots of training points you can build “KD trees”, “ball trees” or other data structures to speed up the query time.
- In high-d, save time by computing the distance of each training point from the min corner and using the “annulus bound”.

**Example: USPS Digits**

- Take 16x16 grayscale images (8bit) of handwritten digits.
- Use Euclidean distance in raw pixel space (dumb!) and 7-nn.
- Classification error (leave-one-out): 4.85%.

Example 7 Nearest Neighbours
**Nonparametric (Instance-Based) Models**

- Q: What are the parameters in K-NN? What is the complexity?
  A: the scalar \( K \) and the entire training set.

Models which need the entire training set at test time but (hopefully) have very few other parameters are known as **nonparametric**, **instance-based** or **case based**.

- What if we want a classifier that uses only a small number of parameters at test time? (e.g. for speed or memory reasons)
  Idea 1: single linear boundary, of arbitrary orientation
  Idea 2: many boundaries, but axis-parallel & tree structured

**Linear Classification for Binary Output**

- Goal: find the line (or hyperplane) which best separates two classes:
  \[
  c(x) = \text{sign}[x^T \hat{w} - w_0]
  \]

  - \( \hat{w} \) is a vector perpendicular to decision boundary
  - This is the opposite of non-parametric: only \( d + 1 \) parameters!
  - Typically we augment \( x \) with a constant term \( \pm 1 \) ("bias unit") and then absorb \( w_0 \) into \( w \), so we don’t have to treat it specially.

**Fisher’s Linear Discriminant**

- Observation: If each class has a Gaussian distribution (with same covariances) then the Bayes decision boundary is linear:
  \[
  w^* = \Sigma^{-1}(\mu_0 - \mu_1)
  \]
  \[
  w_0^* = \frac{1}{2} \hat{w}^T(\mu_0 + \mu_1) - \hat{w}^T(\mu_0 - \mu_1) \left[ \frac{\log p_0 - \log p_1}{(\mu_0 - \mu_1)^T \Sigma^{-1} (\mu_0 - \mu_1)} \right]
  \]

- Idea (Fisher’36):
  - Assume each class is Gaussian even if they aren’t!
  - Fit \( \mu_i \) and \( \Sigma \) as sample mean and sample covariance (shared).
  - This also maximizes the ratio of **cross-class scatter** to **within class scatter**: \( (\hat{z}_0 - \hat{z}_1)^2 / (\text{var}(\hat{z}_0) - \text{var}(\hat{z}_1)) \)

**Digits again**

Train to discriminant "S" from others.
Error = 3.59%
Linear Discriminants are Perceptrons

- The architecture we are using
  \[ c(x) = \text{sign}[x^\top w - w_0] \]
  can be thought of as a circuit/network.
- It was studied extensively in the 1960s and is known as a perceptron.
- There is another way to train the weights, other than Fisher.

**Algorithm perceptronTrain** (Rosenblatt'56)

```plaintext
w ← perceptronTrain(x-train, c-train) {
  w = "small" random values;
  do {
    errors=0;
    for n=1:N {if(c-train(n) != \text{sign}[w'*xtrain(n)]) then {
      w = w + c - train(n)*xtrain(n); errors++;
    }}
  } until(errors==0)
}
```

Perceptron Learning Rules

- Now: cycle through examples, when you make an error, add/subtract
  the example from the weight vector depending on its true class.
- Amazingly, for separable training sets, this always converges.
  (We absorb the threshold as a "bias" variable always equal to -1.)
- For non-separable datasets, you need to remember the sets of weights
  which you have seen so far, and combine them somehow.
- One way: keep the set that survived unchanged for the longest num-
  ber of (random) pattern presentations. (Gallant’s pocket algorithm.)
- Better way: Freund & Shapire’s voted perceptron algorithm.
  Remember all sets and the length of time they survived.
- Perceptron, voted-perceptron, weighted-majority, kernel perceptron,
  Winnow, and other algorithms have a frumpy reputation but they are
  actually extremely powerful and useful, especially using the kernel
  trick. Try these before more complex classifiers such as SVMs!

Tree Structured Axis-Aligned Classifiers

- What if we want more than two regions?
- We could consider a fixed number of arbitrary linear segments
  but even cheaper is to use axis-aligned splits.
- If these form a hierarchical partition, then the classifier is called a
  decision tree or classification tree.
- Each internal node tests one attribute; leaves assign a class.
- Equivalent to a disjunction of conjunctions of constraints on
  attribute values (if-then rules).

Cost Function for Decision Trees

- Define a measure of “class impurity” in a set of examples.
  Push each example down the tree, how “pure” are leaves?
- Goal: minimize expected sum of impurity at leaves at test time.
- Two problems:
  1) We don’t know true distribution \( p(x, y) \).
  2) Search: even if we knew \( p(x, y) \) finding optimal tree is NP.
- So we will take a suboptimal (greedy) approach.
Learning (Inducing) Decision Trees

- Need to pick the order of split axes and values of split points.
  Many algorithms: CART, ID3, C4.5, C5.0.
- Almost all have the following structure:
  1. Put all examples into the root node.
  2. At each node: search all dimensions, on each one chose split
     which most reduces impurity; chose the best split.
  3. Sort the data cases into the daughter nodes based on the split.
  4. Recurse until a leaf condition:
     - number of examples at node is too small
     - all examples at node have same class
     - all examples at node have same inputs
  5. Prune tree down to some maximum number of leaves.
     (Possibly using a different impurity measure than for growing.)

Impurity Measures

- When considering splitting data $D$ at a node on $x_i$, we measure:
  $$\text{Gain}(D; x_i) = I(D) - \sum_{v \in \text{split}(x_i)} \frac{|D_{iv}|}{|D|} I(D_{iv})$$
- Common impurity measures:
  Entropy: $I(D) = -\sum_c p_c(D) \log p_c(D)$ (two classes)
  Misclass: $I(D) = 1 - p_c^*$
  Gini: $I(D) = \sum_c \sum_{c' \neq c} p_c(D)p_{c'}(D)Z_c$
$$= \sum_c p_c(D)(1 - p_c(D))$$
  (Gini is the average error if we stochastically classify with node prior)
- These often favour multi-way splits.
- One solution: normalize by “split information”:
  $$S(D) = -\sum_v \frac{|D_{iv}|}{|D|} \log \frac{|D_{iv}|}{|D|}$$

Restrict to Binary Splits

- A better solution is to always constrain ourselves to binary splits.
- For ordered discrete or real valued nodes, split is natural.
  Also easy to compute.
- For a discrete attribute with $M$ settings, looks like we need to consider $2^M - 1$ splits. But for two classes, there is a trick:
  1. Order the settings according to $p(c|x_i = m)$.
  2. Search exhaustively over $q$, grouping first $q$ and last $M - q$.
  3. Optimal split is one of those.

Real Valued Attributes

- For real valued attributes, what splits should we consider?
  - Idea1: discretize the real value into $M$ bins.
  - Idea2: Search for a scalar value to split on.
    Sounds hard! Lots of real values. But there is a trick:
    Only need to consider splits at midpoints between observed values.
    In fact, only need to consider splits at midpoints between observed values with different classes.
- Complexity: $N \log N + 2N|C|$
Algorithm: DT

root of decision tree = SplitNode(train-data,nmin)

subtree ← SplitNode(D)  
  c = most common class in D  
  if (all class(D) same) or (all x(D) same) or (size(D) < nmin)  
    then return a leaf of class c  
  else for each xi measure Gain(D;xi)  
    return a node which splits on best xi and has daughters:  
      - SplitNode(Div) for all split vals v with nonempty Div  
      - leaf of class c for values with empty Div  

G ← Gain(D,i)  
  G = I(D)  
  for each value v in split(xi)  
  Div = cases in D with xi=v  
  G = G - I(Div)*size(Div)/size(D)

Pruning Decision Trees

- Finding the “optimal” pruned tree.  
  It can be shown that if you start with a tree \( T_0 \) and insist on using  
  a rooted subtree of it, the following sequence of trees contains the  
  optimum tree for all numbers of leaves:  
  1. Let \( U(node) = I(node) - I(subtree-rooted-at-node) \)  
  2. Replace the non-leaf node with the smallest value of:  
     \( U(node)/leaves-below-node \)  
     with a leaf node having majority class.

- Even after pruning, decision trees still have problems:  
  - cannot capture additive structure (OR), for this MARS is better  
  - cannot deal with linear combinations of variables

Overfitting in Trees

- Just as with most other models, decision trees can overfit.  
  In fact they are quite powerful.

- eg: Expressive power of binary trees  
  Q: If all input and outputs are binary, what class of Boolean  
  functions can DTs represent?  
  A: All Boolean functions.

- Hence we must **regularize** to control capacity.

- Typically we do this by limiting the number of leaf nodes.  
  Formally, we define: \( \Phi(T) = \sum_{leaves} I(l) + \alpha|leaves| \).

- Minimizing this for any \( \alpha \) is equivalent to finding the tree of a fixed  
  size with smallest impurity. (cf. Lagrange multipliers).

- Practically, we achieve this via pruning.  
  Often we use Gini/Entropy to grow tree and Misclass to prune it.

DT Variants

- ID3 (Quinlan)  
  - split values are all possible values of \( x_i \)  
  - \( I(D) \) is entropy - no pruning

- C4.5, C5.0 (Quinlan)  
  - binary splits  
  - \( I(D) \) is entropy - error-pruning  
  - “rule simplification”

- CART (Breiman et. al)  
  - binary splits  
  - \( I(D) \) is Gini  
  - minimum-leaf subtree pruning
Open questions...

- How do we chose $K$ in K-NN?
- How do we chose $T_{max}$ for decision trees?
- Can Fisher's Discriminant overfit?
- What about nearest-neighbour or tree-based regression models?

Still to come: Logistic regression, Neural Nets for Classification, Class-Conditional Models (Gaussian and Naive Bayes)