1 Some definitions

- overfitting: Given H, h overfits if $\exists h' \in H$ such that h' has smaller error over all the instances even though h has a smaller error over the training examples.

2 Decision Trees

2.1 ID3 Algorithm

- Constructs trees topdown. Greedy algorithm. Hypothesis space of ID3: set of decision trees. Complete space, maintains only a single hypothesis. Uses all traning examples at each step (reduced sensitivity to individual error).
  - A $\leftarrow$ best attribute
  - assign A as decision attribute for Node
  - for each value of A, create a descendant of node
  - sort training examples to leaves
  - if examples perfectly classified, stop
  - else iterate over leaves
- $Entropy(S) = \sum_{i=1}^{c} -p_i lg(p_i)$ (p_i is proportion belonging to class i, also base can vary–what would cause us to do that?)
- $Gain(S, A) = Entropy(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|} Entropy(S_v)$
  - S_v: subset of S for which attribute A has value v

2.2 Inductive Bias of ID3

- prefers shorter trees
- highest info gain attributes

2.3 Pruning

- Reduced error (?)
- Rule post-pruning (?)
  - grow the tree
  - convert tree into equivalent set of rules
  - prune (generalize) each rule by removing preconditions that result in improving its estimated accuracy
  - sort pruned rules by estimated accuracy. Consider them in this sequence when classifying subsequent instances.

2.4 Adapting Decision Trees to Regression(?)

- splitting criteria: variance
- leaves: average local linear fit

3 Regression and Classification

- Least squared error:The objective consists of adjusting the parameters of a model function to best fit a data set. A simple data set consists of n points (data pairs) (x_i, y_i), i = 1, ..., n, where x_i is an independent variable and y_i is a dependent variable whose value is found by observation. The model function has the form $f(x, \beta)$, where the m adjustable parameters are held in the vector $\boldsymbol{\beta}$. The goal is to find the parameter values for the model which "best" fits the data. The least squares method finds its optimum when the sum, S, of squared residuals $S = \sum_{i=1}^{n} {r_i}^2$ is a minimum. A residual is defined as the difference between the actual value of the dependent variable and the value predicted by the model.

  $r_i = y_i - f(x_i, \boldsymbol{\beta})$ An example of a model is that of the straight line in two dimensions. Denoting the intercept as β_0 and the slope as β_1, the model function is given by $f(x, \boldsymbol{\beta}) = \beta_0 + \beta_1 x$.

4 Neural Networks

4.1 Perceptrons

$$o(x_1 ... x_n) = \begin{cases} 1, & \text{if } w_0 + w_1 x_1 + ... + w_n x_n > 0, \\ 0, & \text{otherwise.} \end{cases}$$

where $w_0, ..., w_n$ is a real-valued weight. Note that w_0 is a threshold that must be surpassed for the perceptron to output 1. Alternatively: $o(\vec{x}) = sgn(\vec{w}\vec{x})$. $H = \{\vec{w} | \vec{w} \in \mathbb{R}^{n+1}\}$.

4.2 Perceptron Training Rule vs Delta Rule

- Perceptron training rule: begin with random weights, apply perceptron to each training example, update perceptron weights when it misclassifies. Iterates through training examples repeatedly until it classifies all examples correctly.
  - $w_i \leftarrow w_i + \Delta w_i$
  - $\Delta w_i = \eta(t - o)x_i$, t: target output for current training example. o:output generated for current training example. η: learning rate.
- To converge, Perceptron training rule needs data to be linearly separable(Decision for this hyperplane is $\vec{w}\vec{x} > 0$) and for η to be sufficiently small.
- Delta rule uses *gradient descent.*
  - (?) task of training linear unit (1st stage of a perceptron without the threshold): $o(\vec{x}) = \vec{w}\vec{x}$
  - training error: $E(\vec{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$, where D: training examples, t_d: target output for training example d, and o_d: output of linear unit for training example d.
  - Gradient descent finds global minimum of E by initializing weights, then repeatedly modifying until it hits the global min. Modification: alters in the direction that gives steepest descent. $\nabla E(\vec{w}) = [\frac{\partial E}{\partial w_0}, ..., \frac{\partial E}{\partial w_n}]$
  - Training rule for gradient descent: $w_i \leftarrow w_i + \Delta w_i$
    $\Delta w_i = -\eta \nabla E(\vec{w})$
  - Training rule can also be written in its component form: $w_i \leftarrow w_i + \Delta w_i$
    $\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$

- Efficient way of finding $\frac{\partial E}{\partial w_i} = \sum_{d \in D}(t_d - o_d)(-x_{id})$, where x_{id} (?) represents single input component x_i for training example d.
- Rewrite: $\Delta w_i = \eta \sum_{d \in D}(t_d - o_d)(x_{id})$ (true gradient descent)
- Problems: slow; possibly multiple local minima in error surface (?-I thought error function was smooth, and would always find the global minimum. Example why not?)
- (?) Stochastic gradient descent: $\Delta w_i = \eta(t - o)x_i$ (known as delta rule). Error rule: $E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2$ (?-relationship to the other gradient descent? Why don't we need to separate it by x_{id} anymore? Is this a vector?)
- Stochastic versus True gradient descent
  - true: error summed over all examples before updating weights. stochastic: weights updated upon examining each training example
  - summing over multiple examples require more computation per weight update step. But using true gradient, so can use a larger step size
  - Stochastic avoids multiple local minima because it uses $\nabla E_d(\vec{w})$ not $\nabla E(\vec{w})$

4.3 Threshold Unit

Unit for multilayer networks. Want a network that can represent highly nonlinear functions. Need unit whose output is nonlinear, but the output is also differentiable function of its inputs. $o = \sigma(\vec{w}\vec{x})$ where $\sigma(y) = \frac{1}{1-e^y}$

BACKPROP

$$E(\vec{w}) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

where outputs: set of output units in network, t_{kd} target, o_{kd} output associated with k^{th} output unit and training example d. (?)
Algorithm BACKPROP

- until termination condition is met:
- for i = 1 to m (m is the number of training examples)
  - set $a^{(1)} = x^{(i)}$ (i^{th} training example)
  - Perform forward propagation by computing $a^{(l)}$ for $l = 2, ..., L$ (L is total number of layers) $a^{(l)} = \sigma(w^{(l-1)}a^{(l-1)})$ = output of the l^{th} layer.
  - Using $y^{(i)}$ compute $\delta^{(L)} = a^{(L)} - y^{(i)}$ ($y^{(i)}$ is the target for the i^{th} training example)
  - Then calculate (??) $\delta^{(L-1)}$ up until $\delta^{(2)}$ ($\delta^{(l)}$ is the "error" of layer l and

$$\delta^{(l)} = w^{(l)}\delta^{(l+1)} .* \sigma'(w^{(l)}a^{(l)})$$

  - update $w^{(l)} = w^{(l)} + \Delta w^{(l)}$ (represents a vector of the weights of layer l) where

$$\Delta w^{(l)} = \eta \delta^{(l)} .* x^{(l)}$$

Momentum

$$\Delta w_n^{(l)} = \eta \delta^{(l)} .* x^{(l)} + \alpha w^{(l)}(n-1)$$

where n is the iteration (adds a momentum α)

- $E_d(\vec{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$ error on training example d
- How to derive the BACKPROP rule??
- BACKPROP for multi-layer networks may converge only at a local minimum (because error surface for multi-layer networks may contain many different minima).
- Alternative Error Functions?
- Alternative Error Minimization Procedures

Recurrent Networks What do I need to know about recurrent networks?

5 Instance Based Learning

5.1 k-NN

- discrete:

$$\hat{f}(x_q) = argmax_{v \in V} \sum_{i=1}^{k} \delta(v, f(x_i))$$

  where $\delta(a, b) = 1$ if $a = b$ and 0 otherwise.

- continuous (for a new value, x_q:

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

6 Support Vector Machines

Maximal Margin Hyperplanes: if data linearly separable, then $\exists(\vec{w}, b)$ such that $\vec{w}^T\vec{x} + b \geq 1$ $\forall \vec{x_i} \in P$ and $\vec{w}^T\vec{x} + b \leq -1$ $\forall \vec{x_i} \in N$ (N, P are the two classes). Want to minimize

$$\vec{w}^T\vec{w}$$

subject to constraints of linear separability.

6.1 Kernel Induced Feature Spaces

Map to higher dimensional *feature space*, construct a separating hyperplane. $X \to H$ is $\vec{x} \to \phi(\vec{x})$.
Decision function is $f(\vec{x}) = sgn(\phi(\vec{x})w^* + b^*)$ (* means optimal weight and bias)
Kernel function: $K(\vec{x}\vec{z}) = \phi(\vec{x})^T\phi(\vec{z})$. If K exists, we don't even need to know what ϕ is.
Mercer's condition:
What if data is not linearly separable? (slack variables?)
Lagrangian?