Contents

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 Lazy vs Eager

- k-NN, locally weighted regression, and case-based reasoning are lazy
- BACKPROP, RBF is eager (why?), ID3 eager
- Lazy algorithms may use query instance x_q when deciding how to generalize (can represent as a bunch of local functions). Eager methods have already developed what they think is the global function.

3 Decision Trees

3.1 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 of S 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

3.2 Inductive Bias of ID3

- prefers shorter trees
- highest info gain attributes

3.3 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.

3.4 Adapting Decision Trees to Regression(?)

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

4 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 $f(x, \boldsymbol{\beta}) = \beta_0 + \beta_1 x$.

5 Neural Networks

5.1 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: $o(\vec{x}) = sgn(\vec{w}\vec{x})$. $H = \{\vec{w} | \vec{w} \in \mathbb{R}^{n+1}\}$.

5.2 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: $w_i \leftarrow w_i + \Delta w_i$
    $\Delta \vec{w} = -\eta \nabla E(\vec{w})$
  - 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 $\nabla E(\vec{w})$

- The cost function for a neural network is non-convex, so it may have multiple minima. Which minimum you find with gradient descent depends on the initialization.

5.3 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 $\sigma(y) = \frac{1}{1-e^y}$

5.4 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
    $$\Delta w^{(l)} = \eta\delta^{(l)}.*x^{(l)}$$

5.5 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?

5.6 Radial Basis Functions

- $\hat{f}(x) = w_0 + \sum_{u=1}^{k} w_u Kern_u(d(x_u, x))$
- Equation can be thought of as training a 2-layer network. First layer computes $Kern_u$, second layer computes a linear combination of these first layer values.
- Kernel is defined such that $d(x_u, x)\uparrow \implies Kern_u\downarrow$
- RBF gives global approximation to target function represented by linear combinations of many local kernel functions (smooth linear combination).
- Faster to train than BACKPROP because input and output layer are trained separately.
- RBF is eager: represents global function as a linear combo of multiple local kernel functions. Local approximations RBF creates are not specifically targeted to the query.
- A type of ANN constructed from spatially localized kernel functions. Sort of the 'link' between k-NN and ANN?

6 Instance Based Learning

6.1 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):

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

- distance-weighted: $w_i = \frac{1}{d(x_q,x_i)^2}$. If $x_q = x_i$ assign $\hat{f}(x_q) = f(x_i)$ (if more than one, do a majority).

- real valued distance weighted:

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

- Inductive Bias of k-NN: assumption that nearest points are most similar

- k-NN is sensitive to having many irrelevant attributes 'curse of dimensionality' (can deal with it by 'stretching the axes', add a weight to each attribute. Can even get rid of some of the attributes by setting the weight =0)

6.2 Locally Weighted Linear Regression

- f approximated near x_q using $\hat{f}(\vec{x}) = \vec{w} \cdot \vec{x}$ (is this appropriate notation?)

- Error function using kernel: $E(x_q) = \frac{1}{2}\sum_{k \in K}(f(x) - \hat{f}(x))^2 Kern(d(x_q, x))$ where K is the set of k closest x to x_q.

7 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.
Or, maximize $\frac{2}{|w|}$ while $y_i(\vec{w}^T\vec{x_1} + b) \geq 1$ $\forall i$. Note $y_i = \{+1, -1\}$. Or minimize $\frac{1}{2}|w|^2$. This is quadratic programming problem.
$W(\alpha) = \sum_i \alpha_i - \frac{1}{2}\sum_{i,j}\alpha_i\alpha_j y_i y_j x_i^T x_j$. $w = \sum_i \alpha_i x_i y_i$. α_i mostly 0 $\implies$ only a few of the x's matter.

7.1 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?
Mercer's Theorem?

7.2 Relationship between SVMs and Boosting

$H_{trial}(x) = \frac{sgn(\sum_i \alpha_i x_i)}{\sum_i \alpha_i}$. As we use more and more weak learners, the error stays the same, but the confidence goes up. This equates to having a big margin (big margins tend to avoid overfitting).

8 Boosting

Boosting problem: set of weak learners combined to produce a learner with an arbitrary high accuracy.
The original boosting problem asks whether a set of weak learners can be combined to produce a learner with an arbitrary high accuracy. A weak learner is a learner whose performance (at classification or regression) is only slightly better than random guessing. AdaBoost: trains multiple weak classifiers on training data, then combines into single boosted classifier. Weighted sum of weak classifiers with weights dependent on weak classifier accuracy.
N training examples: x_i, $y_i \in \{-1, +1\}$. Each example i has an observation weight w_i (how important example i is for our current learning task).
Classifier G: $err_S = \sum_{i=1}^{N} w_i I(y_i \neq G(x_i))$
Using weights: $err = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G(x_i))}{\sum_{i=1}^{N} w_i}$
In this way, our error metric is more sensitive to misclassified examples that have a greater importance weight. Denominator is only for normalization (we want an answer between 0 and N).
Boosting: weights are sequentially updated. Algorithm:

- initialize $w_i = \frac{1}{N}$
- for $m = 1$ to M:
  - fit $G_m(x)$ using w_i's
  - compute
$$err_m = \frac{\sum_{i=1}^{N} w_i I(y_i \neq G_m(x_i))}{\sum_{i=1}^{N} w_i}$$
  - $\alpha_m = \frac{log(1 - err_m)}{err_m}$
  - $w_i \leftarrow w_i \cdot exp(\alpha_m I(y_i \neq G_m(x_i))$ for $i = 1 \ldots N$
- $G(x) = sign[\sum_{m=1}^{M} \alpha_m G_m(x)]$ In this way, classifiers that have a poor accuracy (high error rate, low α_m) are penalized in the final sum.

Question : where are these G_m's coming from? Are they pre-set or are they created by the algorithm?

9 Computational Learning Theory

9.1 Definitions

- H–hypothesis space. $c \in H$–true hypothesis. $h \in H$–candidate hypothesis. $S \subseteq H$–training set.

- Consistent learner: Learner outputs a hypothesis such that $h(x) = c(x)\ \forall x \in S$
- Version space: $VS(S) = \{h \in H$:h consistent wrt to $S\}$ (ie, hypothesis consistent with training examples)
- training error: fraction of training examples misclassified by h.
- true error: fraction of examples that would be misclassified on sample drawn from D (distribution over inputs). $error_D(h) = Pr_{x \sim D}[c(x) \neq h(x)]$
- C is PAC-learnable by learner L using $H \iff L$ will output $h \in H$ (with probability $1 - \delta$) such that $error_D(h) \leq \varepsilon$ in time and samples polynomial in $1/\varepsilon$, $1/\delta$, $|H|$.
- ε-exhausted version space: $VS(S)$ exhausted iff $\forall h \in VS(S)\ error_D(h) \leq \varepsilon$.

9.2 Haussler Theorem

Bounds true error.
Let $error_D(h_i) > \varepsilon$ for $i = 1 \ldots k$ (some h_i's in H). How much data do we need to "knock out" all these hypotheses?
$Pr_{x \sim D}[h_i(x) = c(x)] \leq 1 - \varepsilon$ (probability that h_i matches true concept is low)
$Pr(h_i$ consistent with c on m examples$) \leq (1 - \varepsilon)^m$ (independent).
$Pr(\exists h_i$ consistent with c on m examples$) = k \cdot (1 - \varepsilon)^m \leq |H| \cdot (1 - \varepsilon)^m$
$-\varepsilon \geq ln(1 - \varepsilon) \implies (1 - \varepsilon)^m \leq exp(-\varepsilon m)$
Upper bound that VS not ε-exhausted after m samples: $|H| \cdot exp(-\varepsilon m)$.
Want: $|H| \cdot exp(-\varepsilon m) \leq \delta$ (solve for m).
$m \geq \frac{1}{\varepsilon}(ln(|H|) + ln(\frac{1}{\delta})$

9.3 Infinite Hypotheses Spaces

- Examples: linear separators, ANNs, decision trees (continuous inputs)
- $m \geq \frac{1}{\varepsilon}(8VC(H)lg(\frac{13}{\varepsilon}) + 4lg(\frac{2}{\delta})$
- shatter: A set of instances S is shattered by H if every possible dichotomy of S $\exists h \in H$ that is consistent with this dichotomy.
- $VC(H)$ is size of largest finite subset of instance space that can be shattered by H.
- C PAC-learnable iff VC dimension is finite.

10 Bayesian Learning

10.1 Equations and Definitions

- $P(h)$: probability that a hypothesis h holds
- $P(D)$: probability that training data D will be observed
- Bayes' Rule:

$$P(h|D) = \frac{P(D|h)P(h)}{P(D)}$$

- Find most probable $h \in H$ given D:

$$h_{map} = argmax_{h \in H} P(h|D) = argmax_{h \in H} P(D|h)P(h)$$

- if every $h \in H$ a priori equally probable:

$$h_{ml} = argmax_{h \in H} P(D|h)$$

BRUTE FORCE MAP learning algorithm Output h_{map}
Let's assume:

- D is noise-free
- Target function $c \in H$
- all h (a priori) are equally likely

Then $P(h) = \frac{1}{|H|}$

$$P(D|h) = \begin{cases} 1, & \text{if } d_i = h(x_i) \forall d_i \in D, \\ 0, & \text{otherwise.} \end{cases}$$

$$P(D) = \frac{|VS_{H,D}|}{|H|}$$

$|VS_{H,D}|$ is the set of hypotheses in H that are consistent with D. Consistent learned outputs an h with zero error over training examples.
Therefore

$$P(h|D) = \begin{cases} \frac{1}{|VS_{H,D}|}, & \text{if } h \text{ consistent with } D \\ 0, & \text{otherwise.} \end{cases}$$

Every consistent hypothesis is a MAP hypothesis (with these assumptions)!

10.2 ML and Least-Squared Error

Under certain assumptions any learner that minimizes squared error between the outputs of hypothesis h and training data will output an ML hypothesis. No idea why. ?? ML hypothesis is the one that minimizes the sum of squared errors over the training data.

10.3 Bayes Optimal Classifier

$$P(v_j|D) = \sum_{h_j \in H} P(v_j|h_i)P(h_i|D)$$

(probability that correct classification is v_j)

$$v_{map} = argmax_{v_j \in V} P(v_j|D)$$

10.4 Bayesian Belief Networks

Naive Bayes Classify given attributes: $v_{map} = argmax_{v_j \in V} P(v_j|a_1, ..., a_n)$. Rewrite using Bayes' rule and use naive assumption that all a_i are conditionally independent given v_j. $v_{NB} = argmax_{v_j \in V} P(v_j) \prod_i P(a_i|v_j)$.
Whenever naive assumption is satisfied, v_{NB} same as MAP classification.

EM Algorithm

- arbitrary initial hypothesis
- repeatedly calculates expected values of the hidden variables
- recalculates the ML hypothesis

This will converge to local ML hypothesis, along with estimated values for hidden variables (why?)

11 Evaluating Hypotheses

12 Randomized Optimization

12.1 MIMIC

Directly model distribution.
Algorithm:

- generate samples from $P^{\theta_t}(x)$
- set θ_{t+1} to the n'th percentile
- retain only those samples such that $f(x) \geq \theta_{t+1}$
- estimate $P^{\theta_{t+1}}(x)$
- repeat!

12.2 Simulated Annealing

Algorithm:

- for finite number of iterations:
- sample new point x_t in $N(x)$
- Jump to new sample with probability $P(x, x_t, T)$
- decrease T

$$P(x, x_t, T) = \begin{cases} 1, & \text{if } f(x_t) \geq f(x), \\ exp(\frac{f(x_t)-f(x)}{T}), & \text{otherwise.} \end{cases}$$

Genetic Algorithms WHAT IS??

13 Information Theory

Definitions We'll use shorthand: Just write x instead of $X = x$ for all the possible values that a random event X could take on. (Am I using the terms correctly?)

- Mutual Information: $I(X, Y) = H(X) - H(X|Y)$
- Entropy: $H(A) = -\sum_{s \in A} P(s) lg(P(s))$
- Joint entropy: $H(X, Y) = -\sum_{x \in X} \sum_{y \in Y} P(x, y) lg(P(x, y))$
- Conditional Entropy: $H(Y|X) = -\sum_{x \in X} \sum_{y \in Y} P(x, y) lg(P(y|x))$
- If X independent of Y: $H(Y|X) = H(Y)$ and $H(Y, X) = H(Y) + H(X)$
- Kullback-Leibler divergence: $KL(p||q) = -\sum_{x \in X} p(x) lg(\frac{p(x)}{q(x)})$ for two different distributions p, q.