Contents

1 Machine Learning Principles

There are three main divisions of Machine Learning.

- Supervised Learning - Given a labeled dataset, learn from it and apply it to a new unlabeled dataset (ie identify the function or approximation that produces the proper label).
- Unsupervised Learning - Creating structure from inputs that have no labels
- Reinforcement Learning - Getting feedback on actions over time to learn from.

The goal of machine learning is to (loosely) generalize information about a particular system. It's a form of induction (ie. rules are created based on data) as opposed to deduction (applying a rule to a datapoint). While each division above is defined differently, they can be combined together. For example, we can use unsupervised learning to summarize the data (ie provide a set of labels), and then run supervised learning to label new data points accordingly.

2 Key Terms

Classification A mapping of inputs to a discrete label

Cross Validation Dividing the training sets randomly into a smaller training set and a validation set, running training on the smaller testing set and run it on the validation set. Applying this multiple times can increase potential accuracy on the final test set

Inductive Bias Set of assumptions that, when combined with the training data, describe the classifications assigned by the learner to future data instances.

Overfitting Essentially tuning your algorithm too closely to your training dataset that it does worse in general on the testing set. Mathematically speaking, suppose we took the set of all the possible hypotheses that could explain the data H, and we propose a hypothesis h. If there's an alternate hypothesis (let's call it h') that performs better on the test data (ie has a smaller error), our hypothesis is said to overfit. This can be the case even if our hypothesis performs better on our training set.

Regression A mapping of continuous inputs to outputs (ie approximating a function to produce a continuous value)

Squared Error A mapping of continuous inputs to outputs (ie approximating a function to produce a continuous value)

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

3 Supervised Learning

3.1 Decision Trees

Decision trees allows the ability generate a function that produces a discrete value output. Decision trees lend themselves easily to be written as a series of if-then statements in code. This is because decision trees can be written in disjunction of conjunctions. For example, each branching at a feature can be considered an OR statement, and each parent child relationship is an AND statement. From the Mitchell's textbook, they describe it well with the following statement.

We will play tennis IF (the outlook is sunny AND the humidity is normal) OR (the outlook is overcast) OR (the outlook is rain AND the wind is weak).

3.1.1 Good Problems

The reasons below are general guidelines that indicate Decision Trees are valuable.

- There are a fix set of features with discrete possible values (though extensions have been implemented to allow continuous values for features). For example, saying the temperature is hot, warm, or cold, is more discrete than saying the temperature is in the range from 30 degrees to 100 degrees.
- The problem is a classification problem. Decision trees in the end produce a value (eg. whether to go outside or not). They tend to be not good at providing a real-value (eg. What is the overall ranking this person will receive).
- The description of the approximation needs to be defined (usually in a disjunctive format).
- Training data may contain errors - Decision trees have mechanisms to work around missing data and errors in classification in general.

3.1.2 Implementation Details

Implementation involving what attribute to branch off and in what order. The most common choice here is the ID3 algorithm.

ID3 Algorithm

The ID3 algorithm involves looking at the statistic known as *information gain*. At a high level, a good attribute to branch off of is an attribute that roughly can divide equally between the discrete options (for example half of the dataset. In addition, we'd like the division to be reasonably valid (ie it classifies most accurately). Another measure from Information Theory called *Entropy* is used. Entropy defines a measure on how impure the data is. For a dataset S combined with the proportion of correct attribute values, p_i for c possible attribute values, we can define Entropy as follows.

$$Entropy(S) = \sum_{i=1}^{c} -p_i \log_2 p_i \tag{1}$$

As an edge case, we define $\log_2 0$ to be 0. This will generate a curve that prefers even splits among the attribute values. If all of the answers are correct and all are incorrect, no new information is obtained.

We use Entropy to define *Information Gain* $Gain(S, A)$ over a dataset S and an attribute A as a definition of how much entropy is removed when splitting on that attribute.

$$Gain(S, A) = Entropy(S) - \sum_{v \in values(A)} \frac{|S_v|}{|S|} Entropy(S_v) \tag{2}$$

where $Values(A)$ is the set of all possible values an attribute A could have and S_v is the set of all values, $s \in S$ where that attribute A is set to v (Also described as $A(s) = v$).

Bias

Inductive Bias is to prefer shorter trees over longer trees and favor trees that place high information gain closer to the root. This is an example of a *Preference Bias* where it does not explicitly eliminate the hypothesis space but assigns a value to each hypothesis and chooses the best one. In contrast, a *Restriction Bias* rules out hypotheses explicitly and is never considered again.

3.1.3 Risks

- Decision trees use an inductive learning method by searching through the hypothesis space. It performs a hill climbing algorithm (though could be changed nowadays) which leads to potential local optimums.
- Decision trees use a greedy algorithm which lends to an inability to backtrack
- Decision can lead to overfitting. This can be resolved by applying a pruning process either by restricting the tree height to begin with or removing nodes after an overfitted tree (better in practice).

There are two main types of pruning: Reduced Error Pruning and Rule-Post Pruning. Reduced Error Pruning consists of when removing a subtree produces the same, if not better error rate on the test data. Rule Post-Pruning involves converting an overfitted tree into its set of rules (disjunction of conjunctions), removing the conditions and evaluating the rule. Remove the rule if doing so performs no worse than before. Using a Cross-Validation is useful for this case.

3.1.4 Variants

Continuous Value Attributes

To include continuous values for an attribute, divide the attribute up into discrete intervals. The best way to consider is to compare when the classification boundary line changes when mapping the attribute value to its classification.

Alternate Measurements

Instead of ID3, we can use the GainRatio which penalizes a value based on its split information (ie how many discrete values there are). This measurement can be useful when fields are have a large number of possible values (like dates). The equations are provided below.

$$SplitInformation(S, A) = -\sum_{i=1}^{c} \frac{|S_i|}{|S|} \log_2(\frac{|S_i|}{|S|}) \tag{3}$$

$$GainRatio(S, A) = \frac{Gain(S, A)}{SplitInformation(S, A)} \tag{4}$$

Missing Attribute Values

Examples with missing attribute values can be handled easily in Decision Trees. There are multiple strategies like assigning them the most common value among the data in that tree node. More complex is to employ probability to derive the likelihood the value is one over another. An updated algorithm C4.5 handles this case and more.

Higher Cost Attributes

The $InformationGain$ function can be adjusted in terms of the cost of an attribute. An example here is a medical diagnosis decision tree where it may reduce the value for an invasive surgery to be less desirable though informational than say a blood test with some failure rate. Some values are described noting that they do not guarantee the optimal tree. Examples include:

$$\frac{Gain^2(S, A)}{Cost(A)} \tag{5}$$

$$\frac{2^{Gain(S,A)} - 1}{(Cost(A) + 1)^w} \tag{6}$$

where $w \in [0, 1]$ is a constant to determine the relative importance.

3.2 Neural Networks

Originally motivated from Biology with the neural system, neural networks have evolved to very loosely model a series of neurons.

3.2.1 Good Problems

The reasons below are general guidelines that indicate Neural Networks are valuable.

- There are a large set of features to be learned. Examples such as pixels can help understand correlated or independent features more easily.
- The problem is a classification, regression, or a combination of both. Neural Networks are very versatile in this manner.
- An allowance of training upfront - Neural Networks require longer training times usually, especially in comparison of decision tree learning.
- A quick evaluation of the target function - The advantage from the upfront training is a robust and much quicker evaluation function.

- Training data may contain errors - Neural Networks have mechanisms to work around noisy data and errors in classification in general.
- Description of approximation need not need definition - in contrast to decision tree, the ability to explain "how" the algorithm interprets the input is not needed.

3.2.2 Perceptrons

The base unit that are "networked" together in a Neural Network is a Perceptron. A perceptron takes in a vector (or collection) of inputs and evaluates piecewise to a discrete set of two values (positive or negative case). Each input is weighted and it must exceed a threshold to activate or emit a positive result. In the book, positive and negative results were 1 and -1 respectively whereas the video lectures used 1 and 0 respectively. Mathematically speaking, a perceptron can be defined as follows:

$$o(x_1...x_n) = \begin{cases} 1, & \text{if } w_0 + w_1x_1 + ... + w_nx_n > 0, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

where $w_0, ..., w_n$ is a real-valued weight. The videos do not include w_0 term, but they do introduce a threshold θ. In this case $\theta = -w_0$ is a threshold that must be surpassed for the perceptron to output 1. Alternatively, we can use the sgn function (outputting -1 or 1).

$$o(\vec{x}) = sgn(\vec{w}\vec{x}). \tag{8}$$

In terms of hypothesis space, H, a perceptron can produce an infinite number of possibilities (equal to all possible real valued weight vectors).

$$H = \{\vec{w}|\vec{w} \in \mathbb{R}^{n+1}\} \tag{9}$$

Logic Representation

A perceptron can be described as representing many boolean functions like AND, OR, NAND, and NOR. A single perceptron can not, however, represent XOR. This leads to the final theorem that every boolean function can be represented by some network of perceptrons, which indicates the expressiveness of these. Example representations are included in the table below.

INSERT TABLE HERE

3.2.3 Implementation Details

Much of the implementation of a Neural Network requires a method of training each perceptron and setting the weights for each. One of the most effective methods to handle this is to use Gradient Descent.

Perceptron Training Rule

The basic idea to train the perceptrons is to start with a random set of weights, apply it to a training datasets and update perceptron weights until it makes an error. The weights are then adjusted to work towards a valid solution. The process is iterated until the weights converge to set of values that work well on the entire training set. A weight is adjusted by adding some Δ value to it.

$$w_i \leftarrow w_i + \Delta w_i \tag{10}$$

This adjustment can be calculated by learning a rate that moves closer to the correct classification.

$$\Delta w_i = \eta(t - o)x_i \tag{11}$$

where t is the target output for current training example (ie. the correct answer) and o is the output generated for current training example (ie our wrong answer). The learning rate, represented by η moves the weights into a direction to ensure that the weights are making smaller iterations towards the correct answer. It's usually set to some small value like 0.1 and can be adjusted to decay as iterations increase.

Convergence occurs if the data is linearly separable (ie. there exists some hyperplane that separates classifications.

Delta Rule

Because datasets are not linearly separable, there exists a problem with convergence that result in the algorithm not agreeing to halt. In this case, we can employ gradient descent and the delta rule. In this case, the training will converge to a "best fit." A linear unit is used in the beginning (ie there is no threshold value θ). In order to determine the best fit, a notion of an error value needs to be defined.

$$E(\vec{w}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2 \tag{12}$$

where D is our set of training examples, t_d is the target output for the training example d (a classification or regression), and o_d is the output for the training example d.

Gradient descent finds a minimum by following the path of the hyperplane in the direction that points towards the minimum. Gradient Descent comes from calculus to be a vector of the partial derivative of the error with respect to each weight. The value of the gradient descent means the area where the error increases the greatest. This negation of the gradient descent therefore produces the greatest *decrease* in E, the error. The method then follows the same process of updating the weights.

$$\vec{w} \leftarrow w + \Delta\vec{w} \tag{13}$$

where we use the negative gradient descent to detemine the delta.

$$\Delta\vec{w} = -\eta\nabla E(\vec{w}) \tag{14}$$

where η is the learning rate again. Separating it out for each component weight w_i.

$$\Delta w_i = -\eta\frac{\partial E}{\partial w_i} \tag{15}$$

Calculation of the gradient descent is straightforward and easy to construct. To summarize, the updated change in weights per component is:

$$\Delta w_i = \eta\sum_{d \in D}(t_d - o_d)x_{id} \tag{16}$$

Stochastic Gradient Descent

Gradient Descent has a couple challenges:

- It does not guarantee a global minimum it can get stuck at a local minimum.
- It can be very slow to find that minimum requiring many iterations.

A solution is to apply a *stochastic gradient descent* which updates the weights following each individual training example instead of all training examples (what gradient descent did). Note the lack of the summation in the updated error equation.

$$E_d(\vec{w}) = \frac{1}{2}(t_d - o_d)^2 \tag{17}$$

There are three key differences:

- As stated before gradient descent is summed overall examples before updating weights. Stochastic gradient descent updates weights after each individual example.
- It can be thought that gradient descent takes into account all training examples which is computationally expensive, but they can have a larger step size as a result. Stochastic gradient descent needs a smaller descent.
- Although not guaranteed, stochastic gradient descent can avoid falling into a local minima because it uses different error gradients to guide the search.

3.2.4 Multilayer networks

Most of what we described above has been a focus on calculating weights for a single perceptron. Single perceptrons can only express linear decisions. Multilayer networks can express more complex decisions. The common method of training a multilayer network is through *Backpropagation*. Multiple layers of linear units still produce linear results. Instead of the linear unit, we will use a differentiable threshold unit such as a sigmoid unit. It behaves very much like the discrete perceptron described above, but it provides a smoothed continuous curve to provide both scenarios.

$$o = \sigma(\vec{w}\vec{x}) \tag{18}$$

where

$$\sigma(y) = \frac{1}{1 - e^y} \tag{19}$$

The advantage of this scenario is that it is differentiable, which will make it easier to define the gradient descent.

Backpropagation Algorithm

The BACKPROPAGATION algorithm learns the weights within a multilayer network (assuming the network is of fixed size and number of connections) through a similar gradient descent method to minimize the error. The redefined error function is below.

$$E(\vec{w}) = \frac{1}{2}\sum_{d \in D}\sum_{k \in outputs}(t_{kd} - o_{kd})^2 \tag{20}$$

where *outputs* is the set of output units in the network, and t_kd and o_kd are the same target and output values but associated with the k^{th} output unit and training example d. The algorithm

behaves very similarly as before (feed weights to nodes, run it in the network, calculate error, and adjust weights based on gradient descent). One major difference in the algorithm is the update of weights relies on a more complex δ instead of the general error $(t - o)$. The size of the network needs to be defined in order to run the algorithm, so various sizes will needed to be tested in order to identify the best output. Gradient descent will only guaranteed to converge to a local minima (but it often still produces better results).

3.2.5 Risks

3.3 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?

3.4 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?

4 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.

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

5.2 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.

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.
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.

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?
Mercer's Theorem?

6.2 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).

7 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?

8 Computational Learning Theory

8.1 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 $\forall h \in VS(S)\ error_D(h) \leq \varepsilon$.

8.2 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).
$m \geq \frac{1}{\varepsilon}(ln(|H|) + ln(\frac{1}{\delta})$

8.3 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.

9 Bayesian Learning

9.1 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)!

9.2 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.

9.3 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)

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

9.4 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?)

10 Evaluating Hypotheses

11 Randomized Optimization

11.1 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!

11.2 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

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

12 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.