Topic 0 – Intro and Important Concepts

An Unbiased Learner

Intuition: Choose H that can express every teachable concept (i.e., H is the power set of X)

1. Consider H = disjunction, conjunctions and negations of out earlier hypotheses in H
   - S ← $\{x_i \vee x_j \ldots\}$, where x_i and x_j are positive training instances
   - G ← $\{\neg x_i \vee \neg x_j \ldots\}$, where x_i and x_j are negative training instances
2. Need training examples for every input instance in X to converge to the target concept, eventually $S \equiv G$

Limitation: cannot classify new unobserved input instances, or cannot generalize beyond observed training examples

Inductive vs Deductive

Inductive is not provably correct, guesses unknown

Deductive is provably correct based on memory.

Active vs passive Learning

Active learning can select input to use, actively selects what to learn

Passive learning just takes in everything

Inductive Bias

Let $L(x, D_c)$ denote the classification of input instance x by some learning algorithm L after learning from training examples D_c

Definition: The **inductive bias** of L is a minimal set of assertions B such that for any target concept c and training examples D_c,

$$\forall x \in X, B \wedge D_c \wedge x \rightarrow \big(c(x) = L(x, D_c)\big)$$

Alternatively, knowledge space results in query space.

We put preference for some hypotheses, we do not restrict the hypothesis space

Occam's Razor

We prefer short /simple hypotheses. We prefer a model with less assumption. Simple means good.

Arguments in favor:

- Fewer short hypotheses than long hypotheses (so a short hypothesis that comes out is very likely true, long hypothesis can be one of many)
  - Short / simple hypothesis that fits data unlikely to be a coincidence
  - Long / complex hypothesis that fits data may be a coincidence

Arguments opposing:

- Many ways to define small sets of hypotheses (e.g., all trees with a prime number of nodes that use attributes beginning with "Z") {some weird restriction that is very complex}
- Small sets of short/simple hypothesis can be obtained using different hypothesis representations

Just a principal, a bit fuzzy. Used to explain decisions we make. If we do not follow can cause overfitting.

Overfitting

Definition: Hypothesis $h \in H$ **overfits** the set D of training examples iff

$$\exists h' \in H\{h\}\ (error_D(h) < error_D(h')) \wedge (error_{D_X}(h) > error_{D_X}(h'))$$

Where $error_D(h)$ and $error_{D_X}(h)$ denotes the errors of h over D and set D_X of examples corresponding to instance space X.

h performs well on training data D than h' however h performs worse on data D_X than h'.

Can be caused by noise or error. Can be caused by limited data

How to avoid overfitting?

- Stop growing DT when expanding a node is not statistically significant
- Allow DT to grow and overfit the data, then post prune it.

Use methods like pruning, k-fold, leave one out

How to select "best" DT?

- Measure performance over training examples / data
- Measure performance over a separate validation dataset
- MDL: minimize size(tree) and size(misclassifications)
  - Use extra variable, regulating value, to minimize size

Reduced-Error Pruning

What is pruning? Remove all subtrees and become a leaf node, then get value using PLURALITY-VALUE

Partition data into training and validation sets
Do until further pruning if harmful

1. Evaluate impact on validation set of pruning each possible node
2. Greedily remove the one that most improves the validation set accuracy
    - Produce smallest version of most accurate subtree

Rule-Post-Pruning

convert learned DT to an equivalent set of rules by creating one rule for each path form the root to a leaf

$$cond1 \land cond2 \land \ldots$$

- Prune (generalize) each rule by removing any precondition that improves its estimated accuracy
- Can then sort pruned rules by estimated accuracy into desired sequence for use when classifying unobserved instances.

Inductive Learning Assumption:

- Any hypothesis found to approximate the target function well over a sufficiently large set of training examples will also approximate the target function well over other unobserved examples.
- If we train the model well, it can guess well. This is how we find values for inputs we have not previously seen.

Topic 1 – Concept Learning

Definition: An input instance $x \in X$ **satisfies** (all constraints of) a hypothesis $h \in H$ iff $h(x) = 1$. In other words, h classifies x as a positive example.

Definition: A hypothesis h is **consistent** with a set of training examples D iff $h(x) = c(x)$ for all $\langle x, c(x)\rangle \in D$. In other words, h correctly classifies the training examples.

Definition: h_j is **more general than or equal to** h_k (denoted $h_j \geq_g h_k$) iff any input instance x that satisfies h_k also satisfies h_j.

$$\forall x \in X\ (h_k(x) = 1) \rightarrow \left(h_j(x) = 1\right)$$

$\geq_g$ relation defines a partial ordering (reflexive, antisymmetric and transitive) over H and not a total ordering.

Definition: h_j is **(Strictly) more general than** h_k (denoted by $h_j > h_k$) iff $h_j \geq h_k$ and $h_k \ngeq_g h_j$.

Definition: h_j is **more specific than** h_k iff h_k is more general than h_j.

Concept: Boolean valued function over a set of input instances (each comprising input attributes).

Concept learning: is a form of supervised learning. Infer an unknown Boolean-valued function from training example. Search for a hypothesis $h \in H$ that is consistent with D

How to represent a hypothesis

Hypothesis h is a conjunction of constraints on input attributes.
Each constraint can be:

- A specific value ("Water=warm")
- Don't care (?)
- No value allowed (∅)

Every hypothesis containing 1 or more null (∅) symbols represents an empty set of input instance. Hence classifying every instance as a negative example.

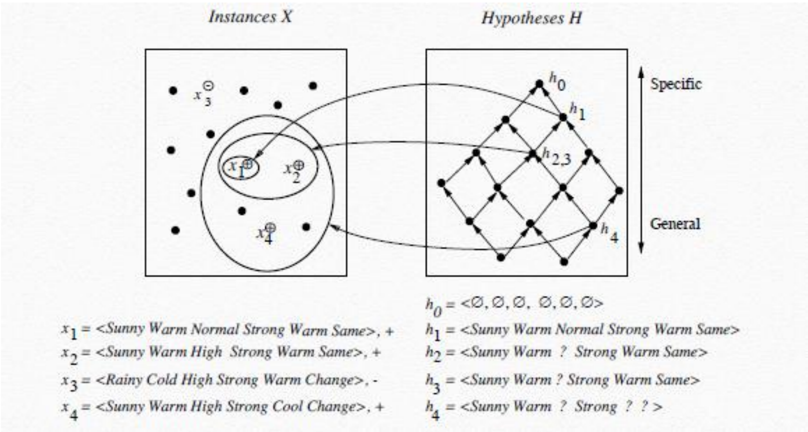
Synthetically distinct hypothesis: (all possible values + 1(?) + 1(∅)) for each input instance

Semantically distinct hypothesis: (all possible values + 1(?)) for each input instance + 1, all empty instance is the same

Find-S

Find the most specific hypothesis (usually all null ∅). Whenever it wrongly classifies a positive training example as negative, "minimally" generalize it to satisfy its input instance.

1. Initialize h to most specific Hypothesis in H
2. For each positive training instance x
    - For each attribute constraint a_i in h:
        i. If x satisfies constraint a_i in h, do noting
        ii. Else replace a_i in h by the next more general constraint that is satisfied by x
3. Output hypothesis h



Proposition 1: h if consistent with D iff every positive training instance satisfies h and every negative instance does not satisfy h
Proposition 2: Suppose that $c \in H$. Then, h_n is consistent with $D = \{\langle x_k, c(x_k)\rangle\}_{k-1,\ldots,n}$.

Limitations of Find-S

- Can't tell whether Find-S has learned target concept
- Can't tell when training examples are inconsistent (i.e., contains error or noise)
- Picks a maximally specific h
- Depending on H, there might be several (vector space)

Version Spaces

Definition: The **Version Space** $VS_{H,D}$ with respect to hypothesis space H and training examples D, is the subset of hypotheses from H consistent with D

$$VS_{H,D} = \{h \in H \mid h \text{ is consistent with } D\}$$

- If $c \in H$, then a large enough D can reduce $VS_{H,D}$ to $\{c\}$
- If D is insufficient, then $VS_{H,D}$ represents the uncertainty of what the target concept is
- $VS_{H,D}$ contains all consistent hypotheses, including maximally specific hypotheses

List-Then-Eliminate Algorithm

Intuition: List all hypotheses in H Then, eliminate any hypothesis found inconsistent with any training example.

1. $VersionSpace \leftarrow$ a list containing every hypothesis in H
2. For each training example $\langle x, c(x)\rangle$
   - Remove from $VersionSpace$ any hypothesis h for which $h(x) \neq c(x)$
3. Output the list of hypotheses in $VersionSpace$

Limitation: Prohibitively expensive to exhaustively enumerate all hypotheses in finite H

Definition: The **general boundary** G of $VS_{H,D}$ is the set of maximally general members of H that is consistent with D.

$$G = \left\{g \in H \mid g \text{ consistent with } D \wedge \left(\neg\exists g' \in H \left(g' >_g g\right) \wedge (g' \text{ consistent with } D)\right)\right\}$$

Definition: The **specific boundary** S of $VS_{H,D}$ is the set of maximally specific members of H that is consistent with D.

$$S = \left\{s \in H \mid s \text{ consistent with } D \wedge \left(\neg\exists s' \in H \left(s >_g s'\right) \wedge (s' \text{ consistent with } D)\right)\right\}$$

These boundaries are independent of the sequence/order of the training examples
Every member of Version Space lies between these boundaries
Vector Space representation theorem:

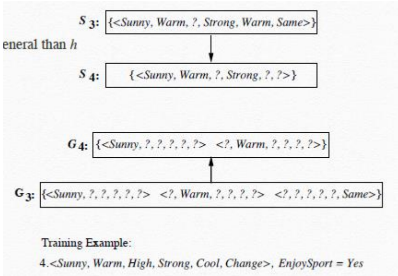
$$VS_{H,D} = \left\{h \in H \mid \exists s \in S, \exists g \in G, g \geq_g h \geq_g s\right\}$$

Candidate Elimination Algorithm

Intuition: Start with most general and specific hypotheses. Each training example "minimally" generalizes S and specializes G to remove inconsistent hypotheses from version space.

1. $G \leftarrow$ maximally general hypotheses in H (all "?")
2. $S \leftarrow$ maximally specific hypotheses in H (all "∅")
3. For each training example d
   - If d is a positive example
     i. Remove from G any hypothesis inconsistent with d
     ii. For each $s \in S$ not consistent with d
        - Remove s from S

        - Add to S all minimal generalizations h of s such that h is consistent with d, and some members of G is more general than h
        - Remove from S any hypothesis that is more general than another hypothesis in S
- If d is a negative example
    i. Remove from S any hypothesis inconsistent with d
    ii. For each $g \in G$ not consistent with d
        - Remove g from G
        - Add to G all minimal specializations/specific h of s such that h is consistent with d, and some members of S is more specific than h
        - Remove from G any hypothesis that is more specific than another hypothesis in G



Properties of Candidate-Elimination

- If there is Error or Noise in training data
    - S and G reduced to $\emptyset$ with sufficiently large data
- Insufficiently expressive hypothesis representation (input instance not representative, some variables are missing)
    - biased hypothesis space $\rightarrow c \notin H? \rightarrow S$ and G also reduced to $\emptyset$ with sufficiently large data
- What input instance should an active learner query next for a training example. (actively select training example to use)
    - Query input instance that satisfies exactly half of hypotheses in Version Space (if possible)
    - Version Space reduces by half with each training example, hence requiring at least $\lfloor \log_2(VS_{H,D}) \rfloor$ examples to find target concept c

Proposition 3: An input instance x satisfies every hypothesis in $VS_{H,D}$ iff x satisfies every member of S.
Proposition 4: An input instance x satisfies none of the hypotheses in $VS_{H,D}$ iff x satisfies none of the members of G.

- How to classify unobserved input instance? What degree of confidence?
    - Majority vote what is the most probable classification, assuming all hypothesis in H are equally probable *a priori*

Inductive Bias of Candidate-Elimination

$$B = \{c \in H\}$$

Assumption: Candidate-Elimination outputs a classification $L(x, D_c)$ of input instance x if this vote among hypotheses in $VS_{H,D}$ is unanimously positive or negative, and does not output a classification otherwise.

Topic 2 – Decision Tree

Why study decision tree

	Concept Learning	Decision Tree Learning
Target function / concept	Binary Outputs	Discrete outputs
Training data	Noise-free	Robust to noise
Hypothesis space	Restricted (hard bias)	Complete, expressive
Search strategy	Complete: version space Refine search per example	Incomplete: prefer shorter trees (soft bias) Refine search using all examples No backtracking
Exploit Structure	General to specific ordering	Simple to complex ordering

Another possible representation for hypotheses. At each level splits up the data based on some input attribute. Because of this, decision trees can express any function of the input. A leaf it the output (true / false).

Target Concept $C \Longleftrightarrow (Path_1 \vee Path_2 \vee \dots)$ where each $Path$ is a conjunction of attribute-value tests required to follow that path leading to a leaf with value true. $Path = (Patrons = full \wedge Time = 06.00)$. This results in substantially simpler than "true" decision tree – a more complex hypothesis isn't justified by small amount of data.

AIM: Find a small tree consistent with the training examples

IDEA: greedily choose "most important" attribute as root of tree or subtree

```
PLURALITY-VALUE(examples):
   Return majority voting of examples

function DECISION-TREE-LEARNING(examples, attributes, parent_examples) returns tree:
   if examples is empty then return PLURALITY-VALUE(parent_examples)
   else if all examples have the same classification then return the classification
   else if attributes is empty then return PLURALITY-VALUE(examples)
   else
   A ← argmax_{a∈attributes}(IMPORTANCE(a, examples))
   tree ← a new decision with root test A
   for each value v_k of A do:
     exs ← {e: e ∈ examples and e.A = v_k}
     subtree ← DECISION-TREE-LEARNING(exs, attributes-A, examples)
     add branch to tree with label (A = v_k) and subtree subtree
   return tree
```

Choosing "Most Important" Attribute

Intuition: A good attribute splits the examples into subsets that are (ideally) "all positive" or "all negative"

By using Information Theory, to implement the $IMPORTANCE$ function in DECISION-TREE-LEARNING algorithm, use entropy to measure uncertainty of classification

Entropy measures uncertainty of certain data set $C \in \{c_1, \dots, c_k\}$

$$H(C) = -\sum_{i=1}^{k} P(c_i) \log_2 P(c_i)$$

Define $B(q)$ as entropy of Boolean r.v. that is true with probability q: $B(q) = -(q \log_2 q + (1-q)\log_2(1-q))$

Or simply, for a training set containing p positive examples and n negative examples, entropy of target concept C on this set is

$$H(c) = B\left(\frac{p}{p+n}\right) = -\frac{p}{p+n}\log_2\frac{p}{p+n} - \frac{n}{p+n}\log_2\frac{n}{p+n}$$

- If $p = n \neq 0$, then $H(C) = B\left(\frac{1}{2}\right) = 1$ (maximum uncertainty)
- If $(p \neq 0, n = 0)$ or $(p = 0, n \neq 0)$, then $H(C) = 0$ (no uncertainty)
- If $p = 2, n = 4, H(C) = B\left(\frac{2}{6}\right) \in (0,1)$ (some uncertainty)

A chosen attribute A divides the training set E into subsets $E_1, \dots, E_d$ corresponding to the d distinct values of A. Each subset E_i has p_i positive and n_i negative examples

$$H(C|A) = \sum_{i=1}^{d} \frac{p_i + n_i}{p+n} B\left(\frac{p_i}{p_i + n_i}\right)$$

Information gain of target concept C from the attribute test on A is the expected reduction in entropy:

$$Gain(C, A) = B\left(\frac{p}{p+n}\right) - H(C|A) = \text{Entropy H(C) of the node - expected reaming entropy after } A$$

Choose the attribute A with the largest $Gain$.

Hypothesis Space Search

Decision tree learning is guided by $IMPOTANCE$ function. Information gain heuristic to search through the space of DTs from simplext to increasingly complex

Inductive Bias of DECISION-TREE-LEARNING

- Shorter trees are preferred
- Trees that place high information gain attributes close to the root are preferred

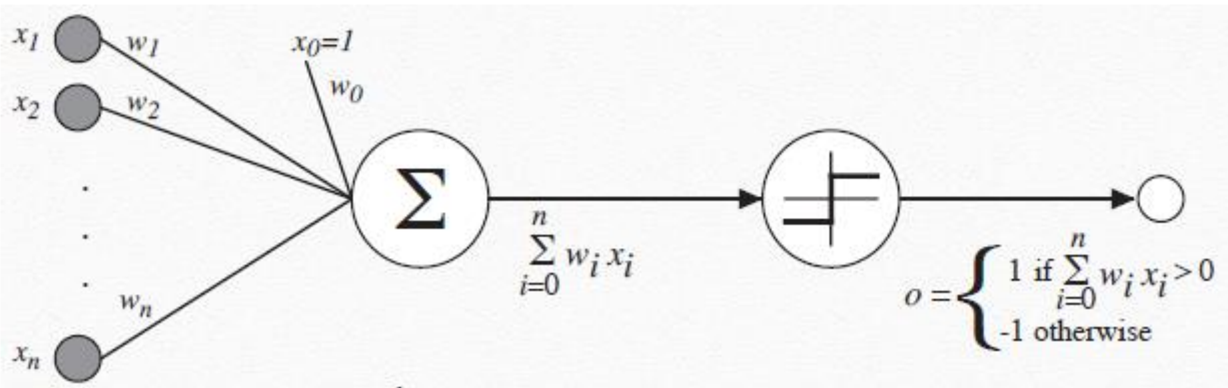
Common Problem Faced

- Continuous-valued attributes
  - Use a discrete valued input attribute to partition the values into discrete intervals.
- Attributes with many values
  - Some values can many possible values (ex: Date, then have one value for each)
  - $IMPORTANCE$ will most likely pick this attribute (cos most likely each has all positive)
  - Solve by using GainRatio
  - $GainRatio(C, A) = \frac{Gain(C,A)}{SplitInformation(C,A)}$
  - $SplitInformation(C, A) = -\sum_{i=1}^{d} \frac{|E_i|}{|E|} \log_2 \frac{|E_i|}{|E|}$
- Attributes with differing costs
  - Attributes have a cost needed to get the data
  - Usually low-cost attributes tend to have more error / noise
  - Replace $Gain$ with:
    - $\frac{Gain^2(C,A)}{Cost(A)}$
    - $\frac{2^{Gain(C,A)}-1}{(Cost(A)+1)^w}$, where $w \in [0,1]$ determines importance of cost
- Missing attribute values
  - What if come examples are missing values for some attribute A
  - Use training examples anyway and sort through DT
    - If node n tests A, then assign most common value of A among other examples sorted to node n
    - Assign most common value of A among other examples sorted to node n with same value of output/target concept
    - Assign probability p_i to each possible value A
      - Assign fraction p_i of example to each descendant in DT
- Then classify new unobserved input instances with missing attributes values in the same manner

Topic 3 – Neural Network

	DT Learning	Neural Networks
Target function / concept	Discrete outputs	Discrete or real vector
Input instance	Discrete	Discrete or real high dimension
Training data	Robust to noise	Robust to noise
Hypothesis space	Complete, expressive	Restricted: #hidden unit (hard bias), expressive
Search strategy	Incomplete: prefer shorter trees (soft bias) Refine search using all examples No backtracking	Incomplete: prefer smaller weights (soft bias) Gradient ascent Batch node: all examples Stochastic: min-batches
Training time	Short	Long
Prediction time	Fast	Fast
Interpretability	White-box	Black-box

Perceptron Unit



Notice $(x_1, .., x_n)$ is all the input attributes. We have an extra $x_0 = 1$ and w_0 to act as biased input and biased weights. To make it more expressive

$$o(x_1, \dots, x_n) = \begin{cases} 1 \ if \ w_0 + w_1x_1 + \dots + w_nx_n > 0 \\ -1 \quad otherwise \end{cases}$$

Using vector notation:

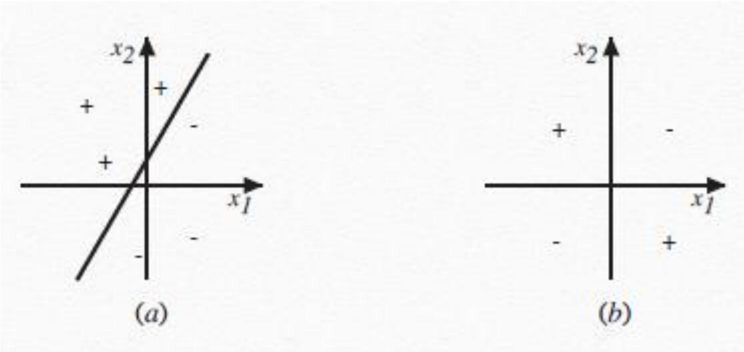
$$o(\boldsymbol{x}) = \begin{cases} 1 \ \ \boldsymbol{w} \cdot \boldsymbol{x} > 0 \\ -1 \ otherwise \end{cases}$$

$$where \ \boldsymbol{w} = (w_0, w_1, \dots w_n)^T \in H = \mathbb{R}^{n+1}$$

$$\boldsymbol{x} = (1, x_1, \dots, x_n)^T \in X = \mathbb{R}^n$$

We want to search for a hypothesis $\underline{w} \in H$

$$\underline{x} = \begin{pmatrix} 1 \\ x_1 \\ x_2 \end{pmatrix}, \underline{w} = \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix}$$



For graph (a), the line is $\underline{w} \cdot \underline{x} = 0$.

The weight vector, $\binom{w_1}{w_2}$ is the orthogonal vector point towards the positive examples.

For graph(a), it points to the negative direction of x_1 axis, then we know that w_1 is negative. Similarly, w_2 is positive.

We need w_0 and x_0 so that the line does not necessarily cross the origin.

If line is above origin, w_0 is negative. If line is below origin, w_q is positive

Figure (a) is **linearly separable,** there exists a horizontal line that can separate the examples. The line is not unique.

Figure (b) is **linearly non-separable,** no horizontal line can separate the examples

Perceptron Training Rule

Idea: Initialize w randomly, apply perceptron training rule to every training example, and iterate thru all training examples till w is consistent. Update weights if it is not consistent with a training example. We iterate through because we can stop at any time, so that we can stop the learning at any moment, if needed.

$$w_i \leftarrow w_i + \Delta w_i, \Delta w_i = \eta(t - o)x_i$$

For $i = 0,1, \dots, n$ where:

- $t = c(x)$ is target output for training example $\langle x, c(x) \rangle$
- $o = o(x)$ is perceptron output
- η is a small positive constant (eg.1) called learning rate.
    - This value is usually small and decreases over time.

This algorithm is guaranteed to converge if training examples are linearly separable and η is sufficiently small.

Gradient Descent

Can be used even if linearly non separable

Idea: Search H to find weight vector that "best fits" the (possibly linearly non-separable) training examples.

Usually need to search a very large space, possibly infinite H. Gradient Descent can do gradient climbing to find

$$o = \boldsymbol{w} \cdot \boldsymbol{x}$$

Learn $\boldsymbol{w}$ that minimizes squared error/loss. This function needs to be differentiable.

$$L_D(\boldsymbol{w}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

Where D is the set of training examples, t_d and o_d are target outputs and output of linear unit for training example d respectively.

Idea: Find $\boldsymbol{w}$ that minimizes L by first initializing it randomly and then repeatedly updating it in the direction of steepest descent

Gradient:

$$\nabla L_D(w) = \left[\frac{\delta L_D}{\delta w_0}, \frac{\delta L_D}{\delta w_1}, \dots, \frac{\delta L_D}{\delta w_n}\right]$$

Training rule:

$$\boldsymbol{w} \leftarrow \boldsymbol{w} + \Delta \boldsymbol{w} \qquad \Delta \boldsymbol{w} = -\eta \nabla L_D(\boldsymbol{w})$$

That is

$$w_i \leftarrow w_i + \Delta w_i \qquad \Delta w_i = -\eta \frac{\delta L_D}{\delta w_i}$$

We put negative because we want to find the steepest descent, $\frac{\delta L_D}{\delta w_i}$ gets us the greatest increase

Gradient Descent Algorithm

Idea: Initialize $\boldsymbol{w}$ randomly, apply linear unit training rule to all training examples and repeat

```
Gradient-Descent(D, η )
Initialize each w_i to some small random value
Until termination condition is met, do
    Initialize each Δw_i to zero.
    For each d ∈ D, do:
        Input instance x_d to linear unit and compute output o
        For each linear unit weight w_i, do
                    Δw_i ← Δw_i + η(t − o)x_id
    For each linear unit weight w_i, do
                    w_i ← w_i + Δw_i
```

Here, o is the output of a linear unit.

Gradient Descent vs Stochastic (incremental) Gradient Descent

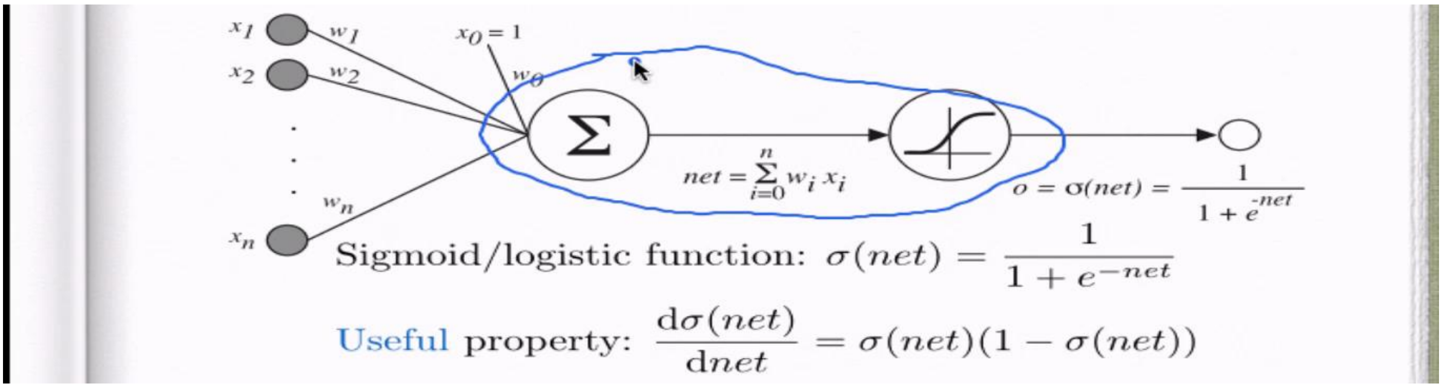
Batch Gradient Descent	Stochastic Gradient Descent
1. Compute gradient $\nabla L_D(\boldsymbol{w})$ 2. $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla L_D(\boldsymbol{w})$ where $L_d(\boldsymbol{w}) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$	For each training example $d \in D$ 1. Compute gradient $\nabla L_D(\boldsymbol{w})$ 2. $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \nabla L_d(\boldsymbol{w})$ where $L_d(\boldsymbol{w}) = \frac{1}{2}(t_d - o_d)^2$

Batch gradient is defined over all training examples, Stochastic Gradient Descent is defined for each example
SGD uses sampling, thus it is an unbiased estimator of true gradient
SGD can approximate batch GD arbitrarily close if learning rate η is sufficiently small
SGD usefulness:

- Computational cost on big data: Although it is linear time. Using SGD, we can decide how many data points to process at each iteration, resulting in constant time at each iteration.
- Any time performance: can get some performance at any time. Even though haven't process all data.
- Economical cost: can buy data based on how much we have or need.
- Can escape local minima: each iteration advises you to another direction to adjust to. Batch will point to one direction

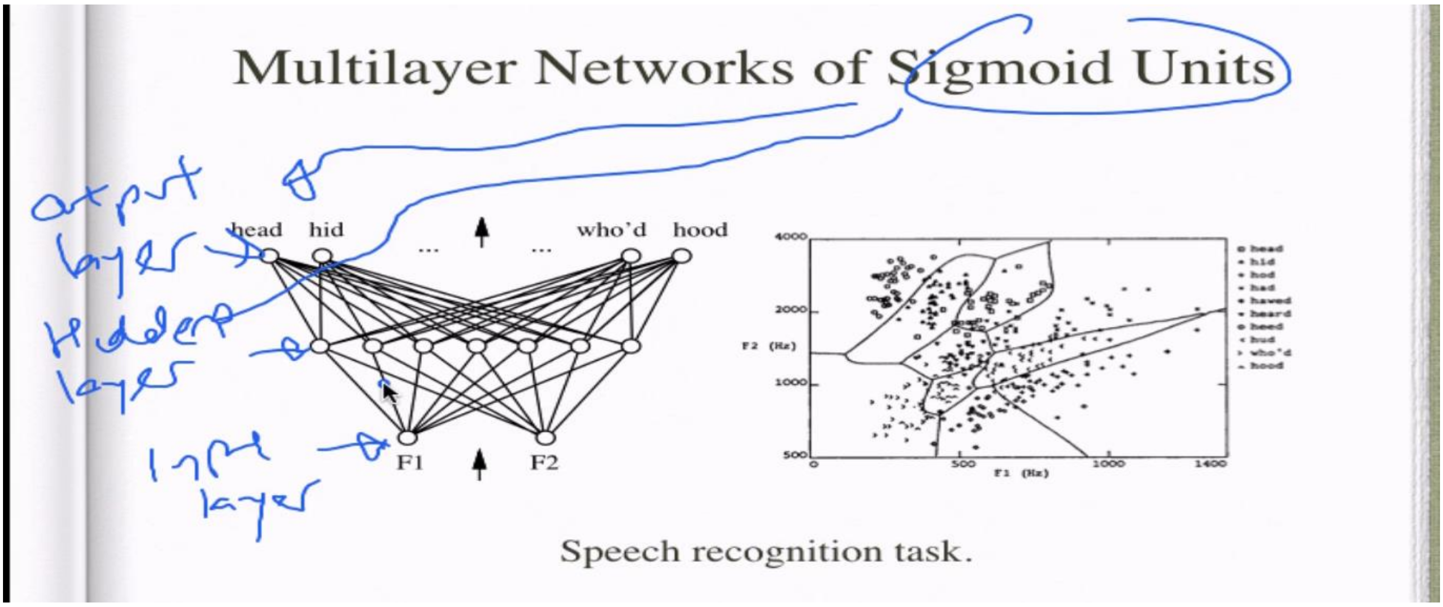
General idea: Objective function (differentiable wrt model parameters w) can be decomposed into a sum of terms, each depending on a subset of training examples. So that we can use Stochastic Gradient Descent. Can use other formula to find expected loss.

Sigmoid Unit



Input to activation function is the net/sum. Output ranges from 0 to 1.
Gradient descent can be derived to train it

Multilayer Networks of Sigmoid Units



Outputs for one hidden layer will be the input for next layer.
Sigmoid in all layers except input layer.
We choose the output with the highest value.

Gradient descent can be derived to train multilayer using backpropagation

Topic 4 – Bayesian Inference

Bayes' Theorem/Belief Update

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

- $P(h)$: prior belief of hypothesis h independent of D
- $P(D|h)$: likelihood of data D given h
- $P(D)$: $\sum_{h \in H} P(D|h)P(h)$: marginal likelihood/evidence of D
- $P(h|D)$: posterior belief of h given D (Law of total probability+)

Limitations:

- Requires specifying probabilities and underlying distributions for every hypothesis
- Often prohibitively expansive to compute evidence. To calculate need to do a lot of maths. To solve, use approximate inference use random sampling.

How to Choose Hypothesis

In normal circumstances, we generally want to pick the hypothesis that is most probable given the training examples, this is known as the *maximum a posteriori* hypothesis, denoted h_{MAP}:

$$h_{MAP} = \max_{h \in H} P(h|D) = \max_{h \in H} \frac{P(D|h)P(h)}{P(D)} = \max_{h \in H} P(D|h)P(h)$$

Since $P(D)$ is not dependent of h, it is just a constant, does not affect finding max

What is the "easiest" way to find MAP hypothesis h_{MAP}?

1. For each hypothesis h compute posterior belief $P(h|D)$
2. Then just find the max

But this is expensive as h becomes very large $(h \to \infty)$

In practice we find this h_{MAP} on top of our algorithms. For example, in FIND-S:

We can set $P(D|h) = \begin{cases} 1 \; if \; h \; is \; consitent \; with \; D \\ 0 \quad\quad otherwise \end{cases}$

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

We get the scenario where every hypothesis in FIND-S is a MAP hypothesis, because they all have same probability, which is the max

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

With more training examples, the version space would decrease, so the probability of each hypothesis increase (Belief Update)

But what if out data is noisy. Suppose we want to find some target function f and training examples $D = \{\langle x_d, t_d \rangle\}$, where t_d is a noisy target output for training example d

We can model is as $t_d = f(x_d) + \epsilon_d$.

- ϵ_d is a random noise variable drawn independently for each x_d according to $\epsilon_d \sim N(0, \sigma^2)$, or its normally distributed

In this case, we want to find the *maximum likelihood* hypothesis H_{ML}, which is the one that minimizes the sum of squared errors

$$h_{ML} = \min_{h \in H} \frac{1}{2} \sum_{d \in D} \left(t_d - h(x_d)\right)^2$$

The $\frac{1}{2}$ is there for mathematical reasons, see lecture notes. Look up probability density function for a normal distribution

Learning to predict hypothesis

Consider the target function/concept $c: X \to \{0,1\}$ and training examples $D = \{\langle x_d, t_d \rangle\}$ where $t_d = c(x_d)$.

X could be like symptoms of a person and $c(x)$ if 1 if the patient survives, 0 otherwise

Now, the hypothesis output the probability that $t_d = 1$ given an input instance. $h(x_d)$ is between 0 and 1

We want to learn a neural network to output $P(c(x) = 1)$ through the use of maximum likelihood hypothesis h_{ML}:

$$h_{ML} = \max_{h \in H} \sum_{d \in D} t_d \ln h(x_d) + (1 - t_d) \ln\left(1 - h(x_d)\right)$$

Useful information: $P(D|h) = \prod_{\mathrm{d \in D}} P(x_d, t_d|h) = \prod_{\mathrm{d \in D}} P(t_d|h, x_d) P(x_d)$

x_d is no longer fixed, it is unknown

To get the last equation use product rule, but is should $P(x_d|h)$. However, h and x_d are independent, so simplify to $P(x_d)$.

Minimum description Length:

Occam's razor states that we prefer short hypothesis that fits the data

$$\begin{aligned} h_{MAP} &= \max_{h \in H} P(D|h) P(h) \\ &= \max_{h \in H} \log_2 P(D|h) + log_2 P(h) \\ &= \min_{h \in H} - \log_2 P(D|h) - log_2 P(h) \end{aligned}$$

This is a result of information theory. The Optimal (shortest expected description length) code for a message with probability p is $-\log_2 p$ bits

- $-\log_2 P(h)$ is description length of h under optimal code
- $-\log_2\left(P(D|h)\right)$ is description length of D given h under optimal code for describing data D

We want to select hypothesis that minimizes

$$h_{MDL} = \min_{h \in H} L_{C_1}(h) + L_{C_2}(D|h)$$

Where $L_C(x)$ is the description length of x under encoding C

- $L_{C_1}(h)$: number of bits to describe h
- $L_{C_2}(D|h)$: number of bits to describe D given h
    - $L_{C_2}(D|h) = 0$ if examples classified perfectly by h. Otherwise, only misclassifications need to be described
    -

Most Probable Classifications of New instance

Suppose we have a new instance x, we want to classify it. What is the most probable classification given the training data D.

h_{MAP} is the most probable hypothesis, does not guarantee the most probable classification

Suppose we have 3 hypotheses $P(h_1|D) = 0.4, P(h_2|D) = 0.3, P(h_3|D) = 0.3$

h_{MAP} would pick h_1

But now suppose that for the new instance x: $h_1(x) = +, h_2(x) = -, h_3(x) = -$

The most probable classification is not + as stated by h_1

We use **Bayes-Optimal Classifier**

$$\max_{t \in T} P(t|D) = \max_{\mathrm{t \in T}} \sum_{h \in H} P(t|h) P(h|D)$$

For a Boolean output, we count the sum of all probabilities over all hypotheses that it is positive, then negative, then find the largest

We are summing all the hypothesis, which is very expensive as h is very large. To solve use Gibbs Classifier

- Sample a h from posterior belief $P(h|D)$
- Use h to classify new instance x

It is very and cheap yet very effective. Expected misclassification error of Gibbs classifier is at most twice of Bayes-optimal classifier

Naïve Bayes Classifier

A very practical Machine Learning models like decision trees and neural networks
Limitations:

- Moderate or Large training data is needed
- Input attributes are conditionally independent given classification (this is a strong assumption we make)

Suppose we have an input instance $x = (x_1, x_2, \dots, x_n)^T$
The most probable classification of new instance x is

$$t_{MAP} = \max_{t \in T} P(t|x_1, \dots, x_n) = \max_{t \in T} P(x_1, \dots, x_n|t)P(t)$$

Using Naïve Bayes Assumption $P(x_1, \dots, x_n|t) = \prod_{\mathrm{i}=1}^{\mathrm{n}} P(x_i|t)$

$$t_{NB} = \max_{t \in T} P(t) \prod_{i=1}^{n} P(x_i|t)$$

Algorithm is surprisingly simple

```
Naïve-Bayes-Learn(D)
   For each value of target output t
     P̂(t) <- estimate P(t) from D
     For each value of attribute x_i
      P̂(x_i|t) ← estimate P(x_i|t) from D
Classify-new-instance(x):
   t_NB = max_{t∈T} P̂(t) ∏_{i=1}^{n} P̂(x_i|t)
```

To put it simply for each of the possible target value, for each of the input: what is the probability that the corresponding value corresponds to the target value. Then find the max of all target values.

Even though our dataset is conditionally dependent, we often just don't care and assume conditional independence. This is because we only need to ensure that

$$\max_{t \in T} \hat{P}(t) \prod_{i=1}^{n} \hat{P}(x_i|t) = \max_{t \in T} P(t)P(x_1, \dots, x_n|t)$$

Issues that arise if what if none of the training instances with target output value t have attribute x_i:

$$\hat{P}(x_i|t) = 0 \rightarrow \hat{P}(t) \prod_{i=1}^{n} (x_i|t) = 0$$

This is very bad, so this target value is never picked
So we use a Bayesian Estimate

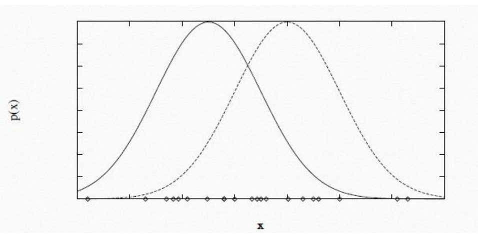
$$\hat{P}(x_i|t) \leftarrow \frac{|D_{tx_i}| + mp}{|D_t| + m}$$

- $|D_t|$ is the number of training examples with target output value t
- $|D_{tx_i}|$ is number of training examples with target output value t and attribute x_i
- p is prior estimate for $\hat{P}(x_i|t)$
- m is weight given to prior p (number of virtual weight)
  - $m \rightarrow 0$ rely more on training example/ frequency counting
  - $m \rightarrow \infty$ tend to p

Set m and p to a value we like. Fine-tune this value

Expectation Maximization

Often times there are latent/hidden variables that we do not capture in our training data. We would like to use our data D to infer this latent/hidden variable.



Given a distribution of our observable data x_d, we would like to know which of these Gaussian Distributions (different means but same variance) would most likely generate these data.

- We do not know the means $\langle \mu_1, \ldots, \mu_M \rangle$ of the M gaussian distributions present
- We also don't know which instance x_d is generated by which gaussian

We want to find the maximum likelihood (*ML*) estimates of $\langle \mu_1, \ldots, \mu_M \rangle$

Consider the full description of each instance as $d = \langle x_d, z_{d1}, \ldots, Z_{dM} \rangle$

- Z_{dm} is unobservable and has value 1 if the m-th gaussian is selected to generate x_d and 0 otherwise
- x_d is observable

The EM algorithm for the case of 2 possible gaussian distributions has 2 steps.

First pick random initial $h = \langle \mu_1, \mu_2 \rangle$. Then iterate

1. E step
   - Calculate expected value $\mathbb{E}[Z_{dm}]$ of each hidden/latent variable z_{dm}, assuming the current hypothesis is correct

   $$\mathbb{E}[Z_{dm}] = \mathbb{E}[Z_{dm}|h, x_d] = 1 \times P(Z_{dm} = 1|h, x_d) + 0 \times P(Z_{dm} = 0|h, x_d) = 1 \times P(Z_{dm} = 1|h, x_d)$$

   $$= \frac{P(x_d|h, z_{dm} = 1)P(Z_{dm} = 1|h)}{\sum_{l=1}^{2} P(x_d|h, z_{dl} = 1)P(z_{dl}|h)}, \text{ by Bayes' theorem}$$

   We assume that each gaussian has equal probability to be chosen, to this simplifies to

   $$\mathbb{E}[Z_{dm}] = \frac{P(x_d|h, z_{dm} = 1)P(Z_{dm} = 1|h)}{\sum_{l=1}^{M} P(x_d|h, z_{dl} = 1)P(z_{dl}|h)} = \frac{P(x_d|\mu_m)}{\sum_{l=1}^{M} P(x_d|\mu_l)} = \frac{\exp\left(-\frac{1}{2\sigma^2}(x_d - \mu_m)^2\right)}{\sum_{l=1}^{2} \exp\left(-\frac{1}{2\sigma^2}(x_d - \mu_l)^2\right)}$$

2. M step
   - We calculate a new *ML* hypothesis $h' = \langle \mu_1', \mu_2' \rangle$, assuming the value taken on by each latent variable z_{dm} is its expected value $\mathbb{E}[Z_{dm}]$ computed above, Replace h with $h' = \langle \mu_1', \mu_2' \rangle$

   $$\mu_m' \leftarrow \frac{\sum_{d \in D} \mathbb{E}[Z_{dm}]\, x_d}{\sum_{d \in D} \mathbb{E}}$$

This EM algorithm provides an estimate of hidden/latent variable Z_{dm}. It converges to the local maximum in $\mathbb{E}[\ln p(D|h')]$

- D is complete containing the observable and unobservable variables
- Expectation is with respect to unobserved variables in D

In a general EM Problem, given:

- Observed data $\{x_d\}_{d \in D}$
- Unobserved data $\{z_d\}_{d \in D}$ where $z_d = \langle z_{d1}, \ldots, z_{dm} \rangle$
- Parameterized probability distribution $p(D|h)$ where
  - $D = \{d\}$ is the complete data where $d = \langle x_d, z_d \rangle$
  - h comprises the parameters

We want to find the *ML* hypothesis h' that maximizes $\mathbb{E}[\ln p(D|h')]$

General EM Algorithm

Define function $Q(h'|h) = \mathbb{E}[\ln p(D|h')|h, \{x_d\}_{d \in D}]$ given current parameters h and observed data $\{x_d\}_{d \in D}$ to estimate the latent variables $\{z_d\}_{d \in D}$

The algorithm is similar to the above

Pick random initial h. Then iterate,

- E step:
  - Calculate $Q(h'|h)$ using current hypothesis h and observe data $\{x_d\}_{d \in D}$ to estimate the latent variables $\{z_d\}_{d \in D}$ and then $Q(h'|h) \leftarrow \mathbb{E}[\ln p(D|h')|h, \{x_d\}_{d \in D}]$
- M step
  - Replace hypothesis h by the hypothesis h' that maximizes this Q function: $h \leftarrow \max_{h'} Q(h'|h)$

Why do we find the expectation of the log likely-hood of $p(D|h)$, its because if we do not, we will get a $\ln \sum_h \ldots$, which is computationally hard. So, by using this Q value, we get $\sum_h \ln \ldots$

Glossary

- H denotes the Hypotheses space, which is the set of all semantically distinct hypotheses
- D denotes the set of training data
- *A priori:* A given proposition is knowable *a priori* if it can be known independent of any experience other than the experience of learning the language in which the proposition is expressed
- *A Posteriori:* a proposition that is knowable *a posteriori* is known on the basis of experience