



Week 12

Machine Learning

Dr Ammar Masood
Department of Cyber Security,
Air University Islamabad

Table of Contents

- Machine learning
- Decision trees
- Practical challenges and Solutions
- Hyperparameter tuning
- Linear and Logistic Regression

Learning

What is Learning?

- An agent learns by improving performance based on past observations.
- Learning ranges from simple tasks (e.g., note-taking) to complex discoveries (e.g., Einstein's theories).
- In computers, this is called **machine learning** using data to build models that solve problems.

Machine Learning

Why Use Machine Learning?

- **Unpredictable Environments:** Pre-programming can't cover all future scenarios.
 - Example: A robot must learn new maze layouts.
- **Unknown Solutions:** Humans may not know how to explicitly program the task.
 - Example: Face recognition is intuitive for humans but hard to code without ML.

Forms of Learning

What Can Be Learned in an Agent

- Any component of an agent (e.g., state-action mappings, world models, utility functions) can be improved through learning.
- **Example:** A self-driving car can learn:
 - When to brake (Condition \rightarrow Action)
 - To recognize objects from images
 - Effects of actions (e.g., skidding)
 - Passenger comfort as a utility metric

Types of learning problems

Type	Description	Output Examples
Classification	Learn discrete categories	e.g., bus / car / person
Regression	Predict continuous values	e.g., stopping distance

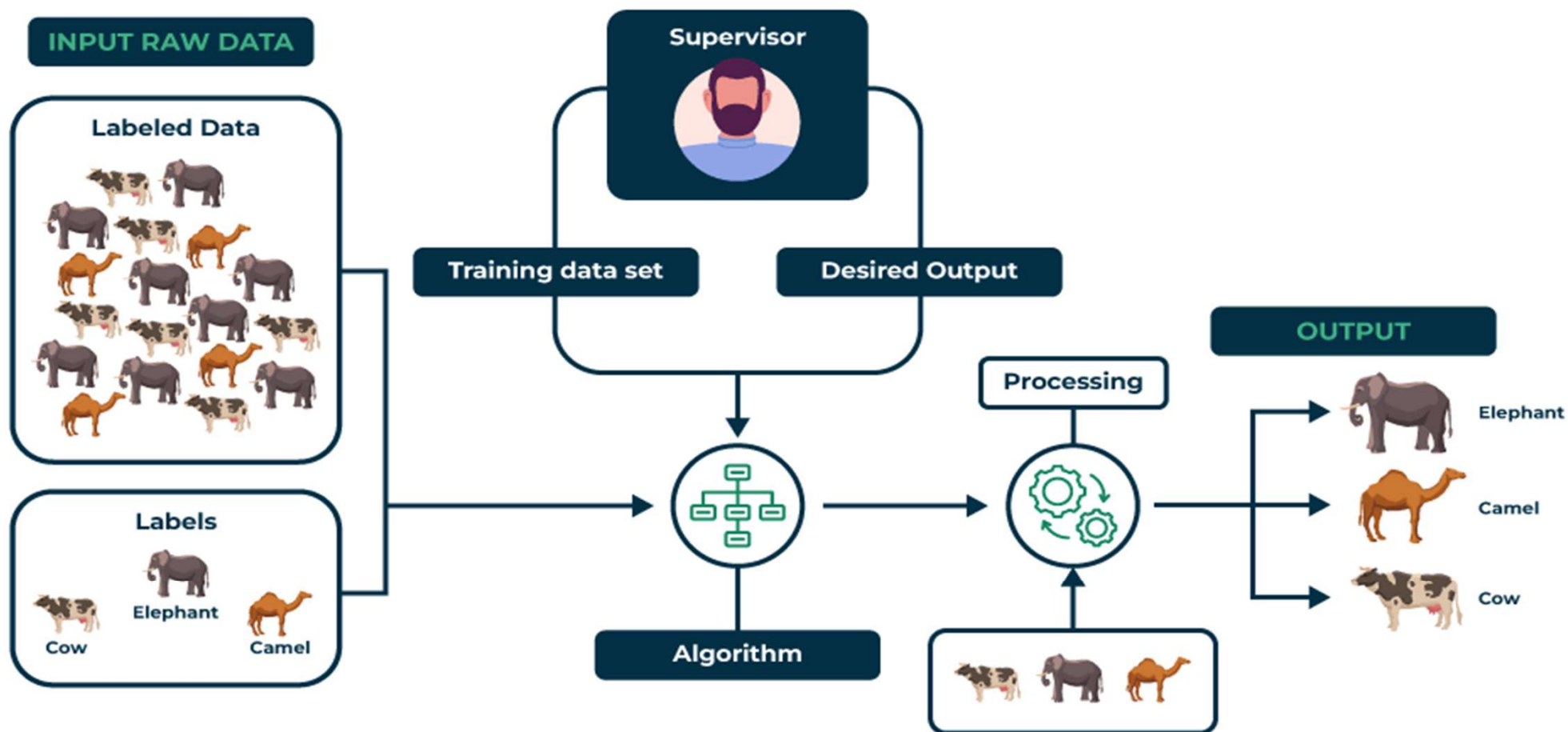
Three learning paradigms

Supervised Learning: Learns from labeled input-output pairs
e.g., Images + "bus" labels → object detector

Unsupervised Learning: Finds patterns without labels
e.g., Cluster millions of images → "cats" emerge

Reinforcement Learning: Learns from reward/punishment signals
e.g., Win/loss in chess → adjusts future strategy

Supervised Learning



Goal

Goal of Supervised Learning

- **Given:** Training set of input–output pairs:
 $(x_1, y_1), (x_2, y_2), \dots, (x_N, y_N)$

where each pair was generated by an unknown function $y = f(x)$

- **Find:** Function h that approximates unknown function f
- **Terminology:**
 - h : Hypothesis (candidate function)
 - H : Hypothesis space
 - y : Ground truth output
 - $h \in H$: Selected model from a class of possible models

Hypothesis space

Choosing a Hypothesis Space

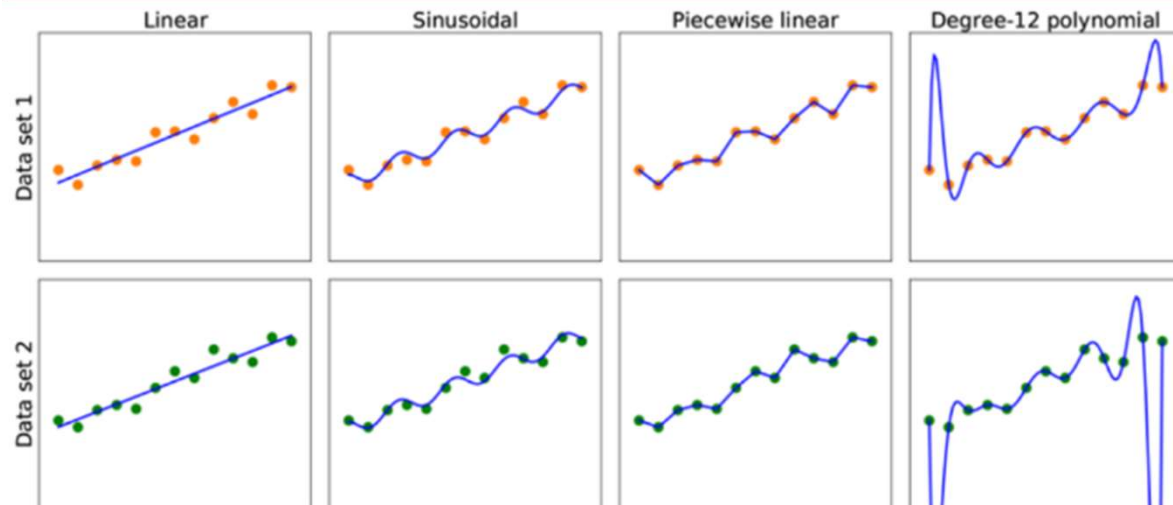
- **Based on:**
 - Prior knowledge about data generation
 - **Exploratory Data Analysis (EDA):** visualizations (scatter plots, box plots, etc.)
- **Approaches:**
 - Try different hypothesis spaces
 - Evaluate performance on training and test sets

Evaluating a Hypothesis

- **Consistent Hypothesis:** $h(x_i)=y_i$ for all training points (possible only with discrete or low-noise data)
- **Generalization:** Ability to perform well on unseen **test data**
- **Best-fit:** Chosen when exact match is not possible (e.g., regression tasks)

Model Comparison

Hypothesis Space	Description	Properties
Straight Line	$h(x)=w_1x+w_0$	Simple, underfits data
Sinusoidal	$h(x)=w_1x+\sin(w_0x)$	Good generalization
Piecewise Linear	Connects data points directly	Always consistent
Degree-12 Poly	$h(x)=\sum w_ix_i$	High variance, overfits



Bias Variance Tradeoff

- **Bias:** Error from overly simplistic models (e.g., linear → underfitting)
- **Variance:** A high-variance model learns the noise in the training data and may overfit, performing poorly on unseen data.
- Error from too much sensitivity to training data (e.g., high-degree polynomials → overfitting)
- **Tradeoff:** Simpler models = high bias, low variance; Complex models = low bias, high variance

Model Selection Considerations

Simplicity vs Appropriateness:

- Fewer parameters seem simpler but may not always generalize better.
- Deep models (e.g., neural nets) can generalize well despite complexity.

Expressiveness vs Complexity:

- Rich hypothesis spaces (e.g., all Turing machines) are expressive but computationally intractable.
- Practical learning uses bounded, efficient models.

Probabilistic Hypothesis Selection

- Use **Bayesian approach**:
 - Choose hypothesis h^* with highest posterior $P(h|data)$

$$h^* = \operatorname{argmax}_{h \in \mathcal{H}} P(h|data) .$$

By Bayes' rule this is equivalent to

$$h^* = \operatorname{argmax}_{h \in \mathcal{H}} P(data|h) P(h) .$$

- Incorporate **priors**: smoother functions get higher prior probabilities
- Penalize complex or unlikely models unless strongly supported by data

Example Restaurant waiting problem

Problem Setup

- **Goal:** Predict whether a person will wait for a table at a restaurant.

- **Output variable:**

$\text{WillWait} \in \{\text{True}, \text{False}\}$

- **Input:**

A vector of **10 discrete attributes**, including:

Attributes

1. **ALTERNATE** – Is there another suitable restaurant nearby?
2. **BAR** – Does the restaurant have a waiting bar?
3. **FRI/SAT** – Is it Friday or Saturday?
4. **HUNGRY** – Are we currently hungry?
5. **PATRONS** – Number of people in the restaurant (None, Some, Full)
6. **PRICE** – Restaurant price range
7. **RAINING** – Is it raining outside?
8. **RESERVATION** – Have we made a reservation?
9. **TYPE** – Type of restaurant (French, Italian, Thai, Burger)
10. **WAITESTIMATE** – Host's estimated wait time

Core Learning Task

- Generalize from 12 examples to unseen combinations.
- **Induction:** Estimate unknown output values (WillWait) based on limited training data.

Restaurant Waiting Dataset (12 Examples)

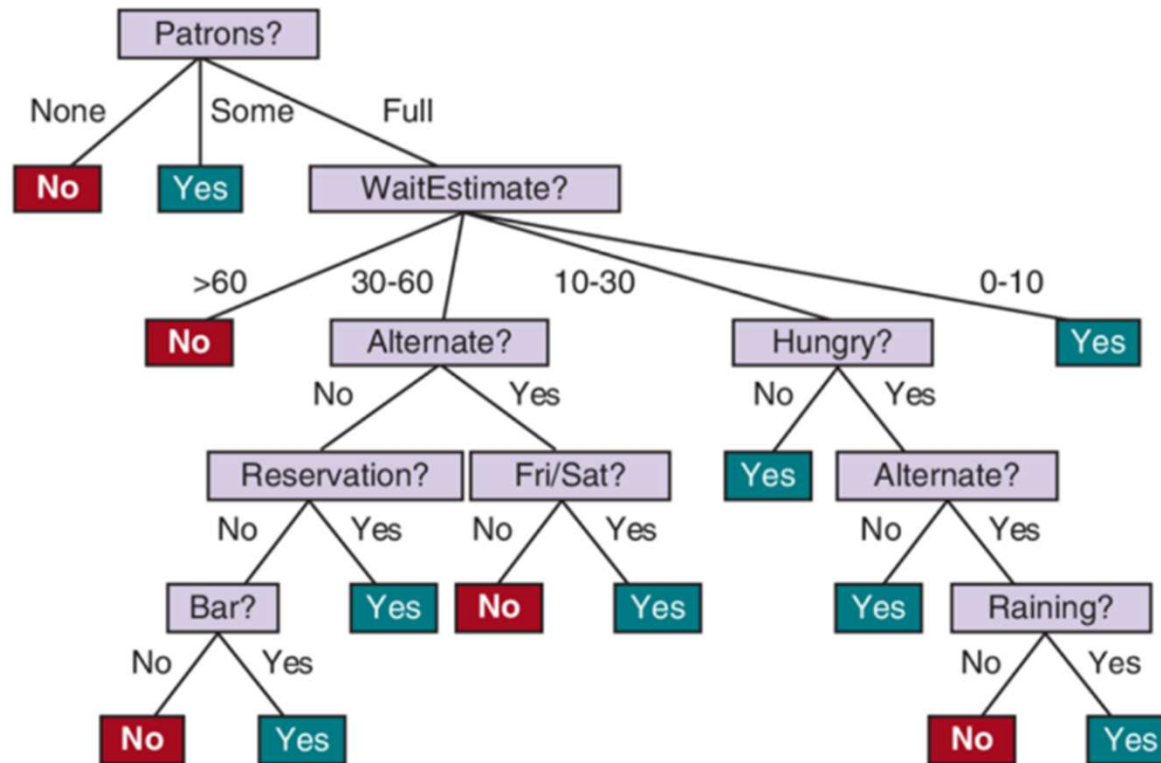
#	Alt	Bar	Fri/Sat	Hungry	Patrons	Price	Rain	Res	Type	WaitEst	WillWait
1	No	Yes	No	Yes	Some	\$\$\$	No	Yes	French	0–10	Yes
2	No	Yes	No	Yes	Full	\$\$	No	No	Thai	30–60	No
3	No	No	Yes	No	Some	\$	Yes	No	Burger	0–10	Yes
4	Yes	No	Yes	Yes	Full	\$	No	No	Thai	10–30	Yes
5	Yes	No	No	Yes	None	\$	No	No	Burger	0–10	No
6	No	Yes	Yes	Yes	Some	\$\$\$	Yes	Yes	French	>60	No
7	Yes	Yes	Yes	Yes	Full	\$\$	No	Yes	Italian	10–30	Yes
8	No	No	No	No	None	\$	Yes	No	Thai	0–10	No
9	Yes	Yes	Yes	Yes	Full	\$	Yes	No	Burger	>60	No
10	No	No	No	Yes	Some	\$\$	Yes	Yes	Thai	10–30	Yes
11	Yes	Yes	Yes	No	Full	\$\$\$	No	Yes	French	30–60	Yes
12	No	No	No	No	None	\$\$	No	No	Burger	0–10	No

Key Observations

- Only 12 data points for a huge input space (~9,216 possible combinations).
- Output (WillWait) varies across combinations of attributes.
- Used to test generalization ability of different learning models.

What Is a Decision Tree?

- A decision tree maps input attribute vectors to a single output (decision).
- **Structure:**
 - **Root & internal nodes:** Test on attributes
 - **Branches:** Possible values of attributes
 - **Leaves:** Output values (True/False)
- Example: Tree for restaurant problem



A decision tree for deciding whether to wait for a table.

Discrete inputs, binary output
 (WillWait = True/False) Follows
 paths like:
 If Patrons = Full → check
 WaitEstimate = 0–10 → then return
 True

Expressiveness of Decision Trees

- Equivalent to **disjunctive normal form (DNF)** logic:
- $\text{Output} \Leftrightarrow (\text{Path1} \vee \text{Path2} \vee \dots)$

Pros:

- Easy to understand and visualize
- Good for rule-based decisions

Limitations:

- Struggles with functions like **parity** or **majority**
- Can't easily represent diagonal or non-axis-aligned boundaries

Learning Decision Trees from Examples

- **Goal:** Learn a **small**, consistent decision tree from training data
- **Challenge:** Finding the smallest tree is **intractable**
- **Solution:** Use **greedy** divide-and-conquer with **important attributes first**

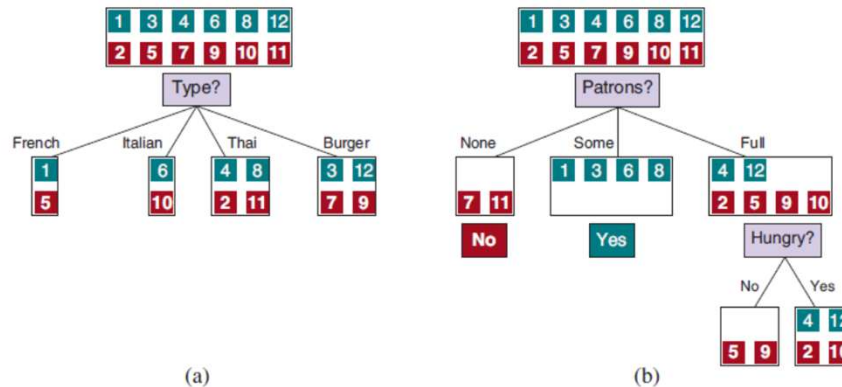
```

function LEARN-DECISION-TREE(examples, attributes, parent_examples) returns a 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
         $A \leftarrow \operatorname{argmax}_{a \in \text{attributes}} \text{IMPORTANCE}(a, \text{examples})$ 
        tree  $\leftarrow$  a new decision tree with root test A
        for each value v of A do
            exs  $\leftarrow \{e : e \in \text{examples} \text{ and } e.A = v\}$ 
            subtree  $\leftarrow$  LEARN-DECISION-TREE(exs, attributes − A, examples)
            add a branch to tree with label (A = v) and subtree subtree
        return tree

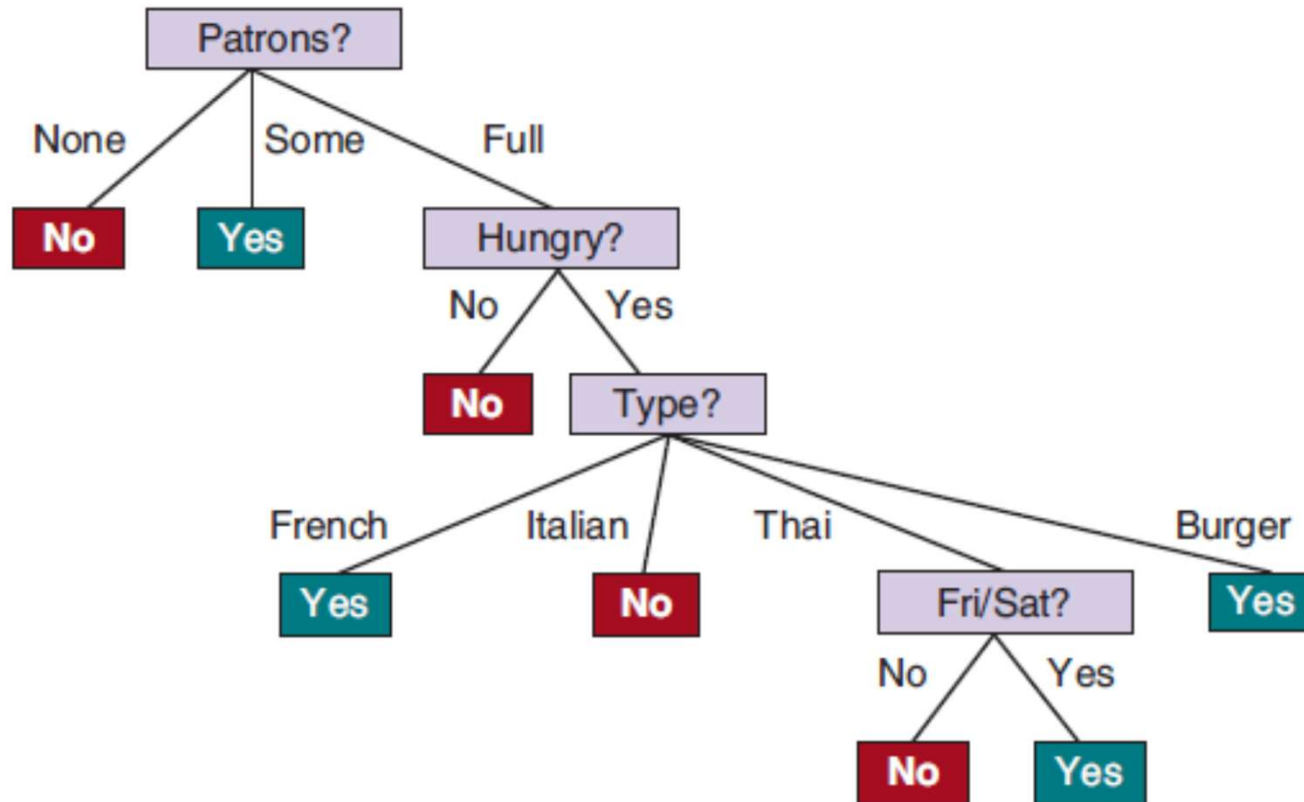
```

Figure 19.5 The decision tree learning algorithm. The function IMPORTANCE is described in Section 19.3.3. The function PLURALITY-VALUE selects the most common output value among a set of examples, breaking ties randomly.

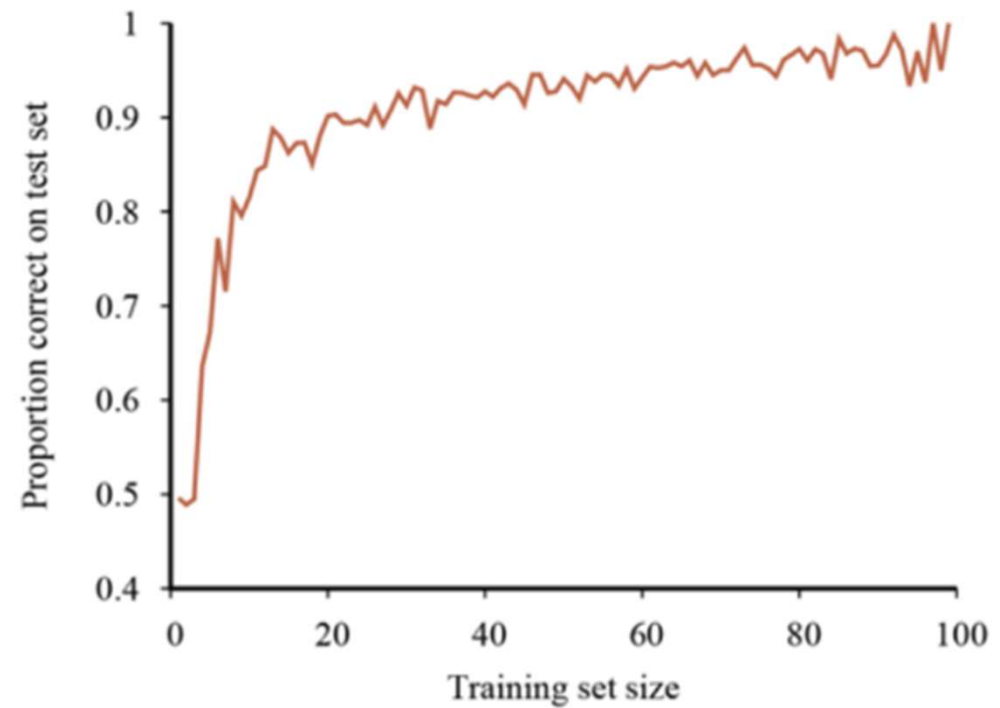
Four Cases When Building the Tree



1. All examples same label → Return that label
2. Mixed labels → Split using best attribute
3. No examples left → Return **plurality label** from parent
4. No attributes left → Conflict due to **noise** or **missing features**



Accuracy improves with
more training data



How to Measure Importance of Attribute?

- Using the notion of information gain, which is defined in terms of entropy.
- The entropy of a random variable V with values v_k having probability $P(v_k)$ is defined as

$$\text{Entropy: } H(V) = \sum_k P(v_k) \log_2 \frac{1}{P(v_k)} = - \sum_k P(v_k) \log_2 P(v_k).$$

We can check that the entropy of a fair coin flip is indeed 1 bit:

$$H(\text{Fair}) = -(0.5 \log_2 0.5 + 0.5 \log_2 0.5) = 1.$$

And of a four-sided die is 2 bits:

$$H(\text{Die4}) = -(0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25 + 0.25 \log_2 0.25) = 2$$

For the loaded coin with 99% heads, we get

$$H(\text{Loaded}) = -(0.99 \log_2 0.99 + 0.01 \log_2 0.01) \approx 0.08 \text{ bits.}$$

It will help to define $B(q)$ as the entropy of a Boolean random variable that is true with probability q :

$$B(q) = -(q \log_2 q + (1 - q) \log_2 (1 - q)).$$

Thus, $H(\text{Loaded}) = B(0.99) \approx 0.08$. Now let's get back to decision tree learning. If a training set contains p positive examples and n negative examples, then the entropy of the output variable on the whole set is

$$H(\text{Output}) = B\left(\frac{p}{p+n}\right).$$

Information Gain

- For the restaurant training set $p = n = 6$, so the corresponding entropy is **$B(0.5)$ or exactly 1 bit**.
- The result of a test on an attribute A will give us some information, thus reducing the overall entropy by some amount. We can measure this reduction by looking at **the entropy remaining after the attribute test**.
- An attribute A with d distinct values divides the training set E into subsets E_1, \dots, E_d . Each subset E_k has p_k positive examples and n_k negative examples

$$Remainder(A) = \sum_{k=1}^d \frac{p_k + n_k}{p + n} B\left(\frac{p_k}{p_k + n_k}\right).$$

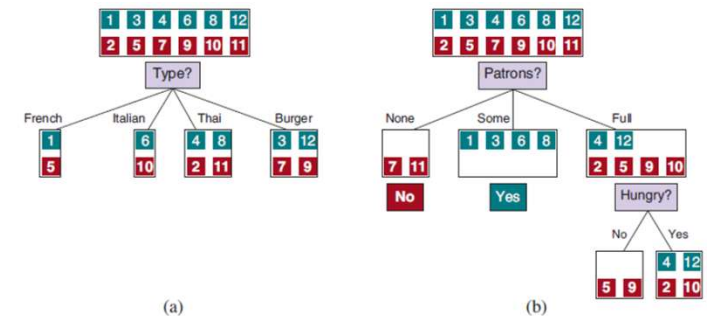
The **information gain** from the attribute test on A is the expected reduction in entropy:

$$Gain(A) = B\left(\frac{p}{p+n}\right) - Remainder(A).$$

In fact $Gain(A)$ is just what we need to implement the IMPORTANCE function. Returning to the attributes considered in Figure 19.4, we have

$$Gain(Patrons) = 1 - \left[\frac{2}{12} B\left(\frac{0}{2}\right) + \frac{4}{12} B\left(\frac{4}{4}\right) + \frac{6}{12} B\left(\frac{2}{6}\right) \right] \approx 0.541 \text{ bits},$$

$$Gain(Type) = 1 - \left[\frac{2}{12} B\left(\frac{1}{2}\right) + \frac{2}{12} B\left(\frac{1}{2}\right) + \frac{4}{12} B\left(\frac{2}{4}\right) + \frac{4}{12} B\left(\frac{2}{4}\right) \right] = 0 \text{ bits},$$



Objective of Learning

- Fit the training data well
- **More importantly:** generalize to unseen data

Overfitting in Decision Trees

- Complex trees may **fit noise** instead of signal
- More likely with:
 - More attributes (high dimensionality)
 - Fewer training examples
 - Larger hypothesis spaces (deep trees, high-degree polynomials)

Decision Tree Pruning

- Solution: **Prune** irrelevant or noisy branches
- Steps:
 - Start with fully-grown tree (from LEARN-DECISION-TREE)
 - Inspect internal nodes with leaf descendants
 - If a **test appears irrelevant** → **replace with majority label leaf**
 - Repeat recursively

Detecting Irrelevance

- Use **Information Gain**:
- $\text{Gain}(A) = B(p/p+n) - \text{Remainder}(A)$
- Low gain \rightarrow likely irrelevant attribute

Practical Challenges & Solutions

- **Missing Data**
- **Problem:**
 - How to classify examples missing attribute values?
 - How to compute information gain if some values are unknown?
- **Solution:** Addressed via specialized strategies

Continuous Output (Regression Trees)

- **Problem:** Predicting numeric values (e.g., apartment price)
- **Solution:**
 - Use **Regression Trees** instead of classification
 - Leaf node contains a **linear function** over numeric attributes
 - Known as **CART**: Classification and Regression Trees

Real-World Readiness

- A robust decision tree system must:
 - Handle missing data
 - Process continuous and categorical features
 - Support classification **and** regression tasks

Error Rate and Evaluation

- **Training Set:** Build the model
- **Test Set:** Unbiased evaluation of final model
- **Goal:** Minimize error rate
 - Error=Proportion where $h(x) \neq y$
 - $\text{Accuracy} = \frac{\text{\# inputs correctly classified}}{\text{\# inputs total}}$
 - $\text{Error Rate} = \frac{\text{\# inputs incorrectly classified}}{\text{\# inputs total}}$
 - Error Rate is thus = **1- Accuracy**

The problem with accuracy

- In most real-world problems, there is one class label that is much more frequent than all others.
 - Words: most words are nouns
 - Animals: most animals are insects
 - Disease: most people are healthy
- It is therefore easy to get a very high accuracy. All you need to do is write a program that completely ignores its input, and always guesses the majority class. The accuracy of this classifier is called the “chance accuracy.”
- It is sometimes very hard to beat the chance accuracy. If chance=90%, and your classifier gets 89% accuracy, is that good, or bad?

The solution: Confusion Matrix

title: Consonant Confusions in CV utterances, for V=/a/, for S/N = +12db and
Phones involved: 16, namely p t k f T (th) s S (sh) b d g v D (dh) z Z (zh) m

Confusion Matrix =

- $(m, n)^{\text{th}}$ element is
- the number of tokens of the m^{th} class
- that were labeled, by the classifier, as belonging to the n^{th} class.

	p	t	k	f	T	s	S	b	d	g	v	D	z	Z	m	n	Total
p	228	7	7	1	0	0	1	0	0	0	0	0	0	0	0	0	p 244
t	0	236	8	0	0	0	0	0	0	0	0	0	0	0	0	0	t 244
k	26	5	213	0	0	0	0	0	0	0	0	0	0	0	0	0	k 244
f	6	1	1	194	35	0	0	3	0	0	1	3	0	0	0	0	f 244
T	0	2	2	96	146	2	0	2	1	0	1	8	0	0	0	0	T 260
s	0	2	0	1	31	204	1	1	9	4	0	7	0	0	0	0	s 260
S	0	0	0	0	0	1	243	0	0	0	0	0	0	0	0	0	S 244
b	0	0	0	13	12	0	0	207	2	3	19	8	0	0	0	0	b 264
d	0	0	0	0	0	0	0	0	240	9	0	0	0	3	0	0	d 252
g	0	0	0	0	0	0	0	1	41	199	0	0	2	1	0	0	g 244
v	0	0	0	3	3	0	0	20	0	2	182	47	2	0	0	1	v 260
D	0	0	0	0	7	0	0	10	3	22	49	170	19	0	0	0	D 280
z	0	0	0	0	1	0	0	3	8	24	2	22	145	3	0	0	z 208
Z	0	0	0	0	0	0	1	0	2	0	0	0	13	264	0	0	Z 280
m	0	0	0	0	0	0	0	0	0	0	0	0	0	0	213	11	m 224
n	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	248	n 248

Plaintext versions of the Miller & Nicely matrices, posted by Dinoj Surendran, <http://people.cs.uchicago.edu/~dinoj/research/nicely.html>

Confusion matrix for a binary classifier

Suppose that the correct label is either 0 or 1. Then the confusion matrix is just 2x2.

For example, in this box, you would write the # inputs of class 1 that were misclassified as class 0

Classified As:

		0	1
Correct Label:	0		
	1		



False Positives & False Negatives

- TP (True Positives) = inputs that were correctly labeled as “1”
- FN (False Negatives) = inputs that should have been “1”, but were mislabeled as “0”
- FP (False Positives) = inputs that should have been “0”, but were mislabeled as “1”
- TN (True Negative) = inputs that were correctly labeled as “0”

		Classified As:	
Correct Label:		0	1
	0	TN	FP
	1	FN	TP

False Alarms and Missed Detections

The **false alarm rate** of a classifier is the fraction of inputs that should have been 0, but were misclassified as 1:

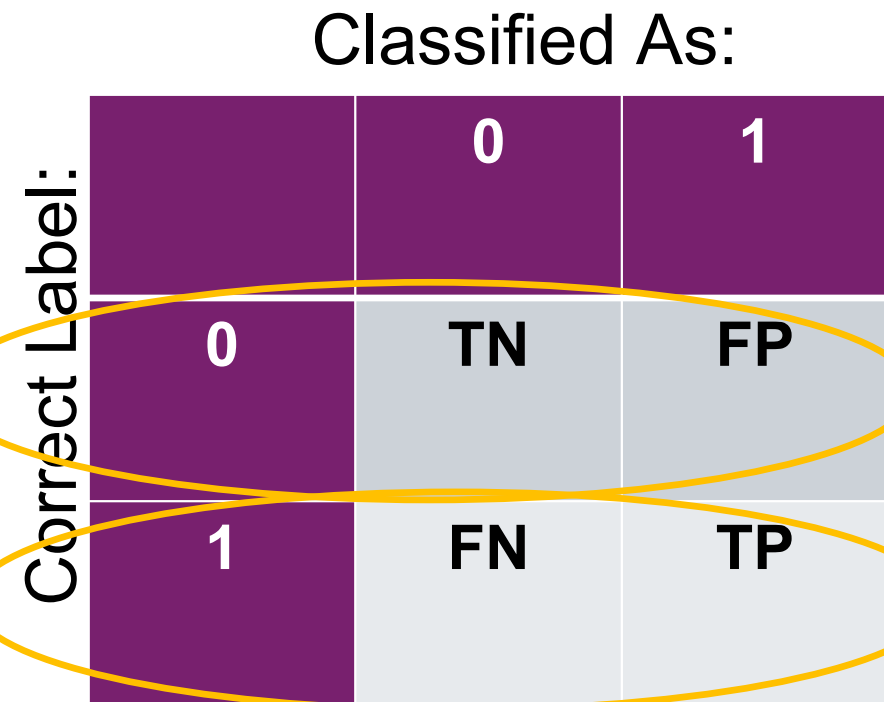
$$\text{False Alarm Rate} = \frac{FP}{TN + FP}$$

The **missed detection rate** of a classifier is the fraction of inputs that should have been 1, but were misclassified as 0:

$$\text{Missed Detection Rate} = \frac{FN}{TP + FN}$$

Classified As:

		0	1
Correct Label:	0	TN	FP
	1	FN	TP



Loss Function

Absolute-value loss: $L_1(y, \hat{y}) = |y - \hat{y}|$

Squared-error loss: $L_2(y, \hat{y}) = (y - \hat{y})^2$

0/1 loss: $L_{0/1}(y, \hat{y}) = 0$ if $y = \hat{y}$, else 1

$$EmpLoss_{L,E}(h) = \sum_{(x,y) \in E} L(y, h(x)) \frac{1}{N}.$$

The estimated best hypothesis \hat{h}^* is then the one with minimum empirical loss:

$$\hat{h}^* = \operatorname{argmin}_{h \in \mathcal{H}} EmpLoss_{L,E}(h).$$

Hyperparameters etc

Hyperparameters and Multiple Models

- **Hyperparameters:** Model-level settings (e.g., degree of polynomial, tree depth)
- Testing different hyperparameters on the same test set is **indirect peeking**

Three-Way Data Split

1. **Training Set** – Build candidate models
2. **Validation Set** – Tune hyperparameters, compare models
3. **Test Set** – Final, unbiased accuracy evaluation

K-Fold Cross-Validation

- Split data into **K equal parts**
- Run **K rounds**:
 - Each round uses $K-1$ parts for training, 1 part for validation
- **Average results** for reliable estimate
- Popular: **K = 5 or 10**
Extreme case: **Leave-One-Out (LOOCV)**

Algorithm

function MODEL-SELECTION(*Learner*, *examples*, *k*) **returns** a (hypothesis, error rate) pair

err \leftarrow an array, indexed by *size*, storing validation-set error rates

training_set, *test_set* \leftarrow a partition of *examples* into two sets

for *size* = 1 **to** ∞ **do**

err[*size*] \leftarrow CROSS-VALIDATION(*Learner*, *size*, *training_set*, *k*)

if *err* is starting to increase significantly **then**

best_size \leftarrow the value of *size* with minimum *err*[*size*]

h \leftarrow *Learner*(*best_size*, *training_set*)

return *h*, ERROR-RATE(*h*, *test_set*)

function CROSS-VALIDATION(*Learner*, *size*, *examples*, *k*) **returns** error rate

N \leftarrow the number of *examples*

errs \leftarrow 0

for *i* = 1 **to** *k* **do**

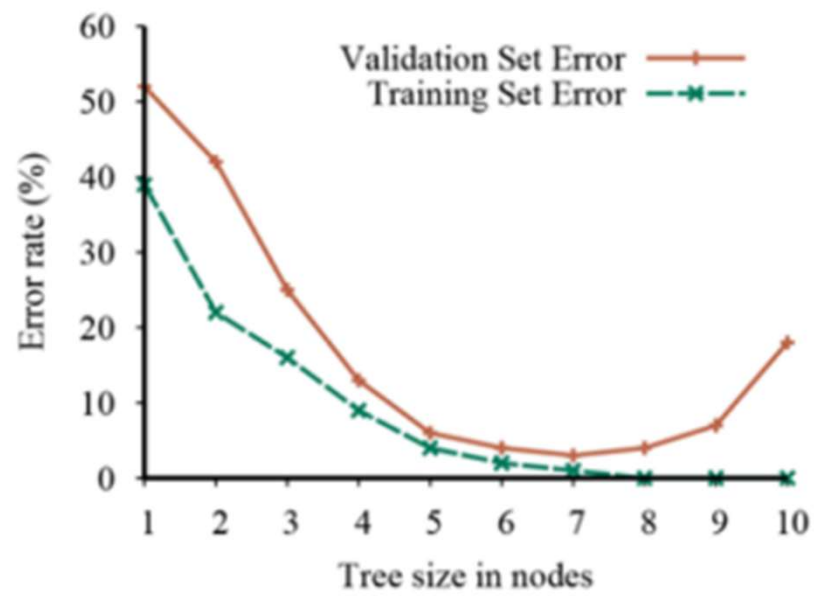
validation_set \leftarrow *examples*[(*i* - 1) \times *N*/*k*:*i* \times *N*/*k*]

training_set \leftarrow *examples* - *validation_set*

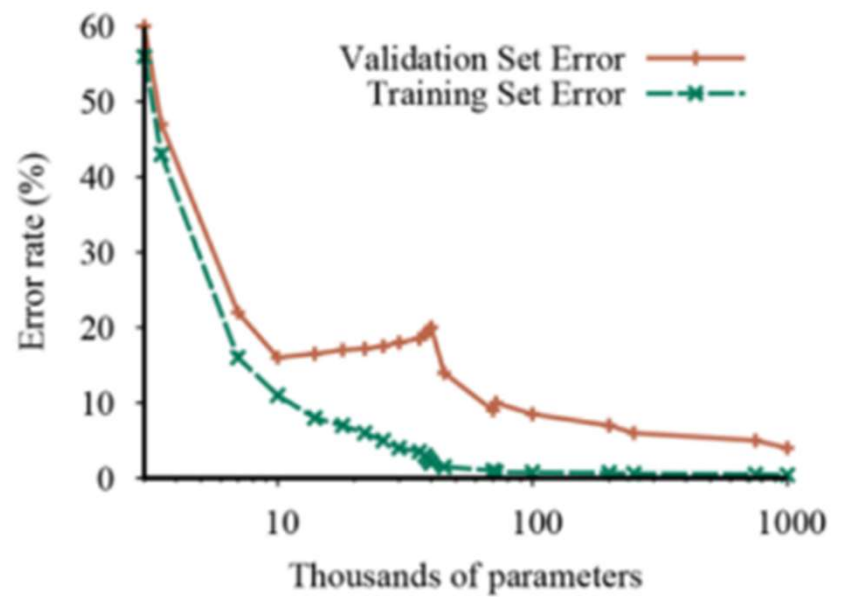
h \leftarrow *Learner*(*size*, *training_set*)

errs \leftarrow *errs* + ERROR-RATE(*h*, *validation_set*)

return *errs* / *k* // average error rate on validation sets, across *k*-fold cross-validation



(a)



(b)

What Is Regularization?

- Goal: **Prevent overfitting** by penalizing complexity
- **Total Cost Function:**
 - $\text{Cost}(h) = \text{EmpLoss}(h) + \lambda \cdot \text{Complexity}(h)$
- λ : Hyperparameter that balances loss vs. complexity
- **Regularization Function Choices**
- Depends on the model:
 - **Polynomials:** Sum of squared coefficients (L2 norm)
 - Other options: L1 norm, tree size, etc.

Feature Selection

- Remove irrelevant input attributes
- Helps simplify model and reduce overfitting(e.g., attribute pruning in decision trees)
- Chi Square Pruning

Minimum Description Length (MDL)

- Encode both model + data in bits
- Choose the hypothesis that minimizes total bits:
 - **Shorter programs = simpler models**
 - Accurate predictions = fewer bits to encode errors

How to do Hyperparameter Tuning?

Adjusting model-level parameters to improve validation performance(e.g., tree depth, regularization strength, degree of polynomial)

Tuning Methods

1. **Hand-Tuning**
 1. Based on intuition & trial-error
2. **Grid Search**
 1. Try all combinations of parameters
 2. Run in parallel for efficiency
3. **Random Search**
 1. Uniform sampling over parameter space
 2. Good for high dimensions & continuous values

Bayesian Optimization

- Treat hyperparameter tuning as its own ML problem:
 - Inputs: Hyperparameter values
 - Output: Validation loss
- Learn function **f** that maps inputs to loss
- Trade-off:
 - **Exploration**: Try new parameters
 - **Exploitation**: Refine promising ones

Population-Based Training (PBT)

- **Step 1:** Start by using random search to train (in parallel) a population of models, each with different hyperparameter values.
- **Step 2:** Then a second generation of models are trained, but they can choose hyperparameter values based on the successful values from the previous generation, as well as by random mutation, as in genetic algorithms

Linear Regression

- Linear models represent outputs as weighted sums of inputs.
- **Applications:**
 - Predicting continuous values (regression)
 - Making decisions via thresholds (classification)

Hypothesis Function

- Linear form:
- $h_w(x) = w_1x + w_0$
- w_0, w_1 : Learnable weights (bias + slope)

$$Loss(h_w) = \sum_{j=1}^N L_2(y_j, h_w(x_j)) = \sum_{j=1}^N (y_j - h_w(x_j))^2 = \sum_{j=1}^N (y_j - (w_1x_j + w_0))^2.$$

We would like to find $w^* = \operatorname{argmin}_w Loss(h_w)$. The sum $\sum_{j=1}^N (y_j - (w_1x_j + w_0))^2$ is minimized when its partial derivatives with respect to w_0 and w_1 are zero:

$$\frac{\partial}{\partial w_0} \sum_{j=1}^N (y_j - (w_1x_j + w_0))^2 = 0 \text{ and } \frac{\partial}{\partial w_1} \sum_{j=1}^N (y_j - (w_1x_j + w_0))^2 = 0. \quad (19.2)$$

These equations have a unique solution:

$$w_1 = \frac{N(\sum x_j y_j) - (\sum x_j)(\sum y_j)}{N(\sum x_j^2) - (\sum x_j)^2}; \quad w_0 = (\sum y_j - w_1(\sum x_j))/N. \quad (19.3)$$

Weight Space Insight

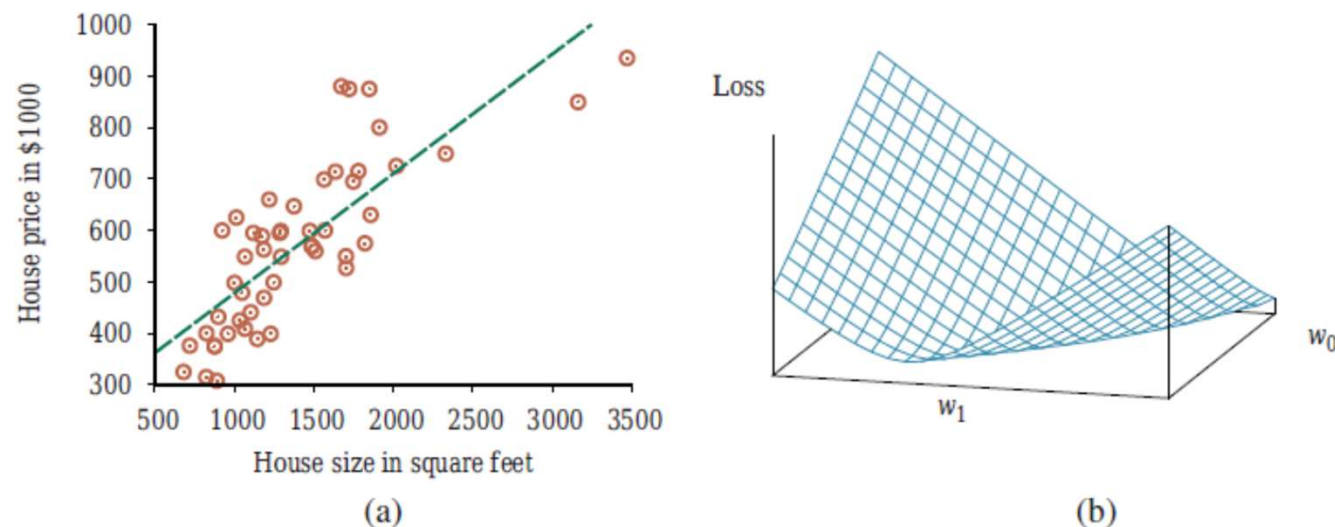


Figure 19.13 (a) Data points of price versus floor space of houses for sale in Berkeley, CA, in July 2009, along with the linear function hypothesis that minimizes squared-error loss: $y = 0.232x + 246$. (b) Plot of the loss function $\sum_j (y_j - w_1 x_j + w_0)^2$ for various values of w_0, w_1 . Note that the loss function is convex, with a single global minimum.

Gradient Descent

The algorithm is as follows:

$\mathbf{w} \leftarrow$ any point in the parameter space
while not converged do
 for each w_i in \mathbf{w} do

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) \quad (19.4)$$

The parameter α , which we called the **step size** in Section 4.2, is usually called the **learning rate** when we are trying to minimize loss in a learning problem. It can be a fixed constant, or it can decay over time as the learning process proceeds.

For univariate regression, the loss is quadratic, so the partial derivative will be linear. (The only calculus you need to know is the **chain rule**: $\partial g(f(x))/\partial x = g'(f(x)) \partial f(x)/\partial x$, plus the facts that $\frac{\partial}{\partial x} x^2 = 2x$ and $\frac{\partial}{\partial x} x = 1$.) Let's first work out the partial derivatives—the slopes—in the simplified case of only one training example, (x, y) :

$$\begin{aligned} \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x))^2 = 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(x)) \\ &= 2(y - h_{\mathbf{w}}(x)) \times \frac{\partial}{\partial w_i} (y - (w_1 x + w_0)). \end{aligned} \quad (19.5)$$

Batch Gradient Descent

Applying this to both w_0 and w_1 we get:

$$\frac{\partial}{\partial w_0} \text{Loss}(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)); \quad \frac{\partial}{\partial w_1} \text{Loss}(\mathbf{w}) = -2(y - h_{\mathbf{w}}(x)) \times x.$$

Plugging this into Equation (19.4), and folding the 2 into the unspecified learning rate α , we get the following learning rule for the weights:

$$w_0 \leftarrow w_0 + \alpha(y - h_{\mathbf{w}}(x)); \quad w_1 \leftarrow w_1 + \alpha(y - h_{\mathbf{w}}(x)) \times x.$$

The preceding equations cover one training example. For N training examples, we want to minimize the sum of the individual losses for each example. The derivative of a sum is the sum of the derivatives, so we have:

$$w_0 \leftarrow w_0 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)); \quad w_1 \leftarrow w_1 + \alpha \sum_j (y_j - h_{\mathbf{w}}(x_j)) \times x_j.$$

These updates constitute the batch gradient descent learning rule for univariate linear regression (also called deterministic gradient descent).

We have to sum over all N training examples for every step, and there may be many steps. The problem is compounded if N is larger than the processor's memory size. A step that covers all the training examples is called an **epoch.**

Stochastic Gradient Descent

- It randomly selects a small number of training examples at each step, and updates
- The original version of SGD selected only one training example for each step, but it is now more common to select a minibatch of m out of the N examples.
- Suppose we have $N = 10,000$ examples and choose a minibatch of size $m = 100$. Then on each step we have reduced the amount of computation by a factor of 100

Multivariable linear regression

$$h_{\mathbf{w}}(\mathbf{x}_j) = w_0 + w_1x_{j,1} + \cdots + w_nx_{j,n} = w_0 + \sum_i w_ix_{j,i}.$$

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^T \mathbf{x}_j = \sum_i w_ix_{j,i}.$$

The best vector of weights, \mathbf{w}^* , minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

Multivariable linear regression

$$h_{\mathbf{w}}(\mathbf{x}_j) = w_0 + w_1x_{j,1} + \cdots + w_nx_{j,n} = w_0 + \sum_i w_ix_{j,i}.$$

The w_0 term, the intercept, stands out as different from the others. We can fix that by inventing a dummy input attribute, $x_{j,0}$, which is defined as always equal to 1. Then h is simply the dot product of the weights and the input vector (or equivalently, the matrix product of the transpose of the weights and the input vector):

$$h_{\mathbf{w}}(\mathbf{x}_j) = \mathbf{w} \cdot \mathbf{x}_j = \mathbf{w}^T \mathbf{x}_j = \sum_i w_ix_{j,i}.$$

The best vector of weights, \mathbf{w}^* , minimizes squared-error loss over the examples:

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \sum_j L_2(y_j, \mathbf{w} \cdot \mathbf{x}_j).$$

Multivariable linear regression

the update equation for each weight w_i is

$$w_i \leftarrow w_i + \alpha \sum_j (y_j - h_{\mathbf{w}}(\mathbf{x}_j)) \times x_{j,i}. \quad (19.6)$$

With the tools of linear algebra and vector calculus, it is also possible to solve analytically for the \mathbf{w} that minimizes loss. Let \mathbf{y} be the vector of outputs for the training examples, and \mathbf{X} be the **data matrix**—that is, the matrix of inputs with one n -dimensional example per row. Then the vector of predicted outputs is $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w}$ and the squared-error loss over all the training data is

$$L(\mathbf{w}) = \|\hat{\mathbf{y}} - \mathbf{y}\|^2 = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2.$$

We set the gradient to zero:

$$\nabla_{\mathbf{w}} L(\mathbf{w}) = 2\mathbf{X}^T(\mathbf{X}\mathbf{w} - \mathbf{y}) = 0.$$

Rearranging, we find that the minimum-loss weight vector is given by

$$\mathbf{w}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y}. \quad (19.7)$$

We call the expression $(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$ the **pseudoinverse** of the data matrix, and Equation (19.7) is called the **normal equation**.

Regularization for Multivariable Linear Functions

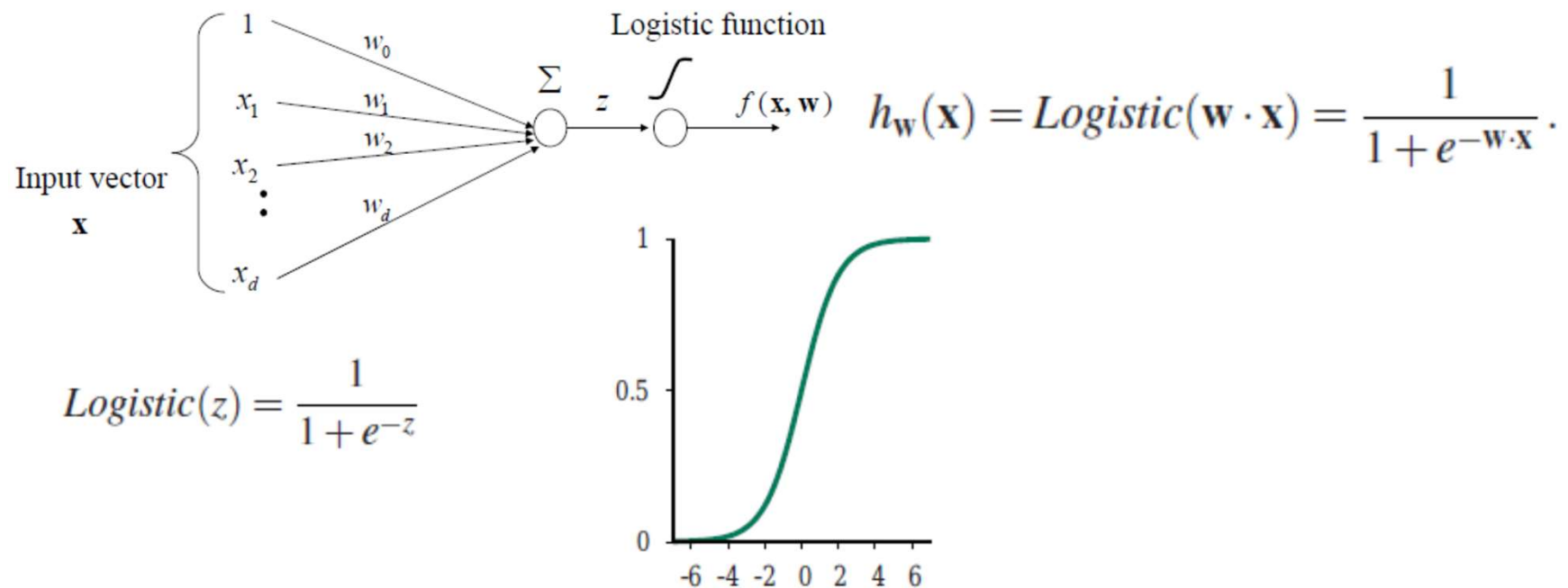
$$Cost(h) = EmpLoss(h) + \lambda Complexity(h).$$

For linear functions the complexity can be specified as a function of the weights. We can consider a family of regularization functions:

$$Complexity(h_{\mathbf{w}}) = L_q(\mathbf{w}) = \sum_i |w_i|^q.$$

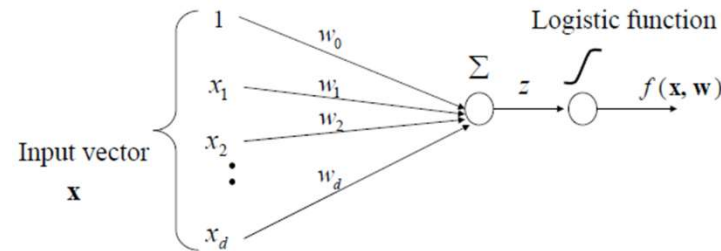
- With $q=1$ we have **L1 regularization**, which minimizes the sum of the absolute values; with $q=2$, **L2 regularization** minimizes the sum of squares.
- Which function should you pick? That depends on the specific problem, but **L1** regularization has an important advantage: it tends to produce a **sparse model**.

Logistic Regression



Output, being a number between 0 and 1, can be interpreted as a probability of belonging to the class labeled 1.

Logistic Regression



- The process of fitting the weights of this model to minimize loss on a data set is called **logistic regression**.
- There is no easy closed-form solution to find the optimal value of \mathbf{w} with this model, but the gradient descent computation is straightforward.

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w})$$

$$\frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) = \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(\mathbf{x}))^2$$

$$= 2(y - h_{\mathbf{w}}(\mathbf{x})) \times \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(\mathbf{x})) \quad h_{\mathbf{w}}(\mathbf{x}) = \text{Logistic}(\mathbf{w} \cdot \mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w} \cdot \mathbf{x}}}$$

$$= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times \frac{\partial}{\partial w_i} \mathbf{w} \cdot \mathbf{x}$$

$$= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i.$$

g stands for the logistic function, with g' its derivative.

The derivative g' of the logistic function satisfies $g'(z) = g(z)(1 - g(z))$, so we have

$$g'(\mathbf{w} \cdot \mathbf{x}) = g(\mathbf{w} \cdot \mathbf{x})(1 - g(\mathbf{w} \cdot \mathbf{x})) = h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x}))$$

so the weight update for minimizing the loss takes a step in the direction of the difference between input and prediction, $(y - h_{\mathbf{w}}(\mathbf{x}))$, and the length of that step depends on the constant α and g' :

$$w_i \leftarrow w_i + \alpha (y - h_{\mathbf{w}}(\mathbf{x})) \times h_{\mathbf{w}}(\mathbf{x})(1 - h_{\mathbf{w}}(\mathbf{x})) \times x_i. \quad (19.9)$$

$$w_i \leftarrow w_i - \alpha \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w})$$

$$\begin{aligned} \frac{\partial}{\partial w_i} \text{Loss}(\mathbf{w}) &= \frac{\partial}{\partial w_i} (y - h_{\mathbf{w}}(\mathbf{x}))^2 \\ &= -2(y - h_{\mathbf{w}}(\mathbf{x})) \times g'(\mathbf{w} \cdot \mathbf{x}) \times x_i. \end{aligned}$$