What's on our menu today?

Supervised Learning

- Classification
  - Decision trees
  - Cross validation
  - K-nearest neighbor
  - Neural networks
    - Perceptrons
  - Support Vector Machines (SVMs)
- Regression
  - Backpropagation networks
Why Learning?

Learning is essential for unknown environments
  e.g., when designer lacks omniscience

Learning is necessary in dynamic environments
  Agent can adapt to changes in environment not
  foreseen at design time

Learning is useful as a system construction method
  Expose the agent to reality rather than trying
  to approximate it through equations etc.

Learning modifies the agent's decision mechanisms to
  improve performance

Types of Learning

Supervised learning: correct answers for each input is
  provided
  E.g., decision trees, backpropagation neural networks

Unsupervised learning: correct answers not given, must
  discover patterns in input data
  E.g., clustering, principal component analysis

Reinforcement learning: occasional rewards (or
  punishments) given to guide behavior
**Inductive learning**

We will focus on one form of supervised learning called Inductive Learning:
Learn a function from examples

\( f \) is the target function. Examples are pairs \((x, f(x))\)

Problem: learn a function ("hypothesis") \( h \)
such that \( h \approx f \) (\( h \) approximates \( f \) as best as possible)
given a training set of examples

(This is a highly simplified model of real learning:
- Ignores prior knowledge
- Assumes examples are given)

---

**Inductive learning example**

Construct \( h \) to agree with \( f \) on training set

- \( h \) is consistent if it agrees with \( f \) on all training examples

E.g., curve fitting (regression):

\[
f(x) \]

\[ x \]

\( x \) = Input data point (training example)
Inductive learning example

$h = \text{Straight line?}$

What about a quadratic function?

What about this little fella?
Inductive learning example

Finally, a function that satisfies all!

But so does this one...
Ockham’s Razor Principle

Ockham’s razor: prefer the simplest hypothesis consistent with data
Related to KISS principle ("keep it simple stupid")
Smooth blue function preferable over wiggly yellow one
If noise known to exist in this data, even linear might be better (the lowest x might be due to noise)

Supervised Learning Technique I:
Decision Trees

To play or not to play?
Example data for learning the concept “Good day for tennis”

Day | Outlook | Humid | Wind | PlayTennis?
---|---------|-------|------|--------
 d1 | s       | h     | w    | n      
 d2 | s       | h     | s    | n      
 d3 | o       | h     | w    | y      
 d4 | r       | h     | w    | y      
 d5 | r       | n     | w    | y      
 d6 | r       | n     | s    | y      
 d7 | o       | n     | s    | y      
 d8 | s       | h     | w    | n      
 d9 | s       | n     | w    | y      
 d10| r       | n     | w    | y      
 d11| s       | n     | s    | y      
 d12| o       | h     | s    | y      
 d13| o       | n     | w    | y      
 d14| r       | h     | s    | n      

• Outlook = sunny, overcast, rain
• Humidity = high, normal
• Wind = weak, strong

A Decision Tree for the Same Data

Decision Tree for “PlayTennis?”

Leaves = classification output
Arcs = choice of value for parent attribute

Decision tree is equivalent to logic in disjunctive normal form

\[
\text{PlayTennis} \iff (\text{Sunny} \land \text{Normal}) \lor \text{Overcast} \lor (\text{Rain} \land \text{Weak})
\]
**Decision Trees**

**Input**: Description of an object or a situation through a set of attributes

**Output**: a decision that is the predicted output value for the input

Both input and output can be discrete or continuous

Discrete-valued functions lead to classification problems

---

**Example: Decision Tree for Continuous Valued Features and Discrete Output**

Input real number attributes \((x_1, x_2)\), Classification output: 0 or 1

How do we branch using attribute values \(x_1\) and \(x_2\) to partition the space correctly?
Example: Classification of Continuous Valued Inputs

Decision trees divide the feature space into axis-parallel rectangles, and label each rectangle with one of the $K$ classes.

Expressiveness of Decision Trees

Decision trees can express any function of the input attributes.

E.g., for Boolean functions, truth table row = path to leaf:

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>A xor B</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>F</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>F</td>
</tr>
</tbody>
</table>

Trivially, there is a consistent decision tree for any training set with one path to leaf for each example

- But most likely won’t generalize to new examples

Prefer to find more compact decision trees
Learning Decision Trees

Example: When should I wait for a table at a restaurant?

Attributes (features) relevant to *Wait?* decision:
1. Alternate: is there an alternative restaurant nearby?
2. Bar: is there a comfortable bar area to wait in?
3. Fri/Sat: is today Friday or Saturday?
4. Hungry: are we hungry?
5. Patrons: number of people in the restaurant (None, Some, Full)
6. Price: price range ($, $$, $$$)
7. Raining: is it raining outside?
8. Reservation: have we made a reservation?
9. Type: kind of restaurant (French, Italian, Thai, Burger)
10. WaitEstimate: estimated waiting time (0-10, 10-30, 30-60, >60)

Example Decision tree

A decision tree for *Wait?* based on personal “rules of thumb”:
Input Data for Learning

Past examples when I did/did not wait for a table:

<table>
<thead>
<tr>
<th>Example</th>
<th>Att</th>
<th>Bar</th>
<th>Fri</th>
<th>Hun</th>
<th>Pat</th>
<th>Price</th>
<th>Rain</th>
<th>Res</th>
<th>Type</th>
<th>Est</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₁</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>French</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>X₂</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>30–60</td>
<td>F</td>
</tr>
<tr>
<td>X₃</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Burger</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>X₄</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>10–30</td>
<td>T</td>
</tr>
<tr>
<td>X₅</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Full</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>French</td>
<td>&gt;60</td>
<td>F</td>
</tr>
<tr>
<td>X₆</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$</td>
<td>T</td>
<td>T</td>
<td>Italian</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>X₇</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>None</td>
<td>$</td>
<td>T</td>
<td>F</td>
<td>Burger</td>
<td>0–10</td>
<td>F</td>
</tr>
<tr>
<td>X₈</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$$</td>
<td>T</td>
<td>T</td>
<td>Thai</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>X₉</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>Full</td>
<td>$</td>
<td>T</td>
<td>F</td>
<td>Burger</td>
<td>&gt;60</td>
<td>F</td>
</tr>
<tr>
<td>X₁₀</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>Italian</td>
<td>10–30</td>
<td>F</td>
</tr>
<tr>
<td>X₁₁</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>None</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>0–10</td>
<td>F</td>
</tr>
<tr>
<td>X₁₂</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Burger</td>
<td>30–60</td>
<td>T</td>
</tr>
</tbody>
</table>

Classification of examples is positive (T) or negative (F)

Decision Tree Learning

Aim: find a small tree consistent with training examples
Idea: (recursively) choose "most significant" attribute as root of (sub)tree

```python
function DTL(examples, attributes, default) returns a decision tree
    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value vᵢ of best do
            examples ← {elements of examples with best = vᵢ}
            subtree ← DTL(examples, attributes − best, MODE(examples))
            add a branch to tree with label vᵢ and subtree subtree
        return tree
```
Choosing an attribute to split on

Idea: a good attribute should reduce uncertainty
- E.g., splits the examples into subsets that are (ideally) "all positive" or "all negative"

Patrons? is a better choice

How do we quantify uncertainty?
Using information theory to quantify uncertainty

Entropy measures the amount of uncertainty in a probability distribution.

**Entropy** (or Information Content) of an answer to a question with possible answers $v_1, \ldots, v_n$:

$$I(P(v_1), \ldots, P(v_n)) = \sum_{i=1}^{n} -P(v_i) \log_2 P(v_i)$$

Using information theory

Imagine we have $p$ examples with $\text{Wait} = \text{True}$ (positive) and $n$ examples with $\text{Wait} = \text{false}$ (negative).

Our best estimate of the probability of $\text{Wait} = \text{true}$ or $\text{false}$ is given by:

$$P(\text{true}) = \frac{p}{p+n}$$
$$P(\text{false}) = \frac{n}{p+n}$$

Hence the entropy of $\text{Wait}$ is given by:

$$I(\frac{p}{p+n}, \frac{n}{p+n}) = -\frac{p}{p+n} \log_2 \frac{p}{p+n} - \frac{n}{p+n} \log_2 \frac{n}{p+n}$$
Entropy is highest when uncertainty is greatest.

Choosing an attribute to split on

Idea: a good attribute should reduce uncertainty and result in “gain in information”

How much information do we gain if we disclose the value of some attribute?

Answer:

uncertainty before - uncertainty after
Before choosing an attribute:
Entropy = \(-\frac{6}{12} \log(\frac{6}{12}) - \frac{6}{12} \log(\frac{6}{12})\)
= \(-\log(\frac{1}{2}) = \log(2) = 1\) bit
There is “1 bit of information to be discovered”

If we choose Type: Go along branch “French”: we have entropy = 1 bit; similarly for the others.
Information gain = 1-1 = 0 along any branch

If we choose Patrons:
In branch “None” and “Some”, entropy = 0
For “Full”, entropy = \(-\frac{2}{6} \log(\frac{2}{6}) - \frac{4}{6} \log(\frac{4}{6})\) = 0.92
Info gain = (1-0) or (1-0.92) bits > 0 in both cases
So choosing Patrons gains more information!
Entropy across branches

- How do we combine entropy of different branches?
- Answer: Compute average entropy
- Weight entropies according to probabilities of branches
  - 2/12 times we enter “None”, so weight for “None” = 1/6
  - “Some” has weight: 4/12 = 1/3
  - “Full” has weight 6/12 = ½

\[
\text{AvgEntropy} = \sum_{i=1}^{n} \frac{p_i + n_i}{p + n} \text{Entropy}
\]

Information gain

Information Gain (IG) or reduction in entropy from using attribute A:

\[
IG(A) = \text{Entropy before} - \text{AvgEntropy after choosing } A
\]

Choose the attribute with the largest IG
Information gain in our example

Patrons has the highest IG of all attributes
⇒ DTL algorithm chooses Patrons as the root

\[
IG(\text{Patrons}) = 1 - \left[ \frac{2}{12} I(0,1) + \frac{4}{12} I(1,0) + \frac{6}{12} I\left(\frac{2}{6}, \frac{4}{6}\right) \right] = .541 \text{ bits}
\]

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

Should I stay or should I go?
Learned Decision Tree

Decision tree learned from the 12 examples:

Substantially simpler than “rules-of-thumb” tree
• more complex hypothesis not justified by small amount of data
**Performance Evaluation**

How do we know that the learned tree \( h \approx f \)?

Answer: Try \( h \) on a new test set of examples

Learning curve = \% correct on test set as a function of training set size

---

**Generalization**

How do we know the classifier function we have learned is good?

- Look at generalization error on test data
  - Method 1: Split data in training vs test set (the “hold out” method)
  - Method 2: Cross-Validation
Cross-validation

K-fold cross-validation:
- Divide data into k subsets of equal size
- Train learning algorithm K times, leaving out one of the subsets. Compute error on left-out subset
- Report average error over all subsets

Leave-1-out cross-validation:
- Train on all but 1 data point, test on that data point; repeat for each point
- Report average error over all points

Decision trees are for girlie men – let’s move on to more powerful learning algorithms

http://www.ipsnet.com/schwarzenegger2/pages/arnold_01.htm
Example Problem: Face Detection

How do we build a classifier to distinguish between faces and other objects?

Images as Vectors

Treat an image as a high-dimensional vector (e.g., by reading pixel values left to right, top to bottom row)

\[
\mathbf{I} = \begin{bmatrix}
    p_1 \\
    p_2 \\
    \vdots \\
    p_{N-2} \\
    p_N
\end{bmatrix}
\]

Pixel value \( p \) can be 0 or 1 (binary image) or 0 to 255 (greyscale)

<table>
<thead>
<tr>
<th>Images as Vectors</th>
<th>Binary handwritten characters</th>
<th>Greyscale images</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Binary Image]</td>
<td>![Greyscale Image]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The human brain is extremely good at classifying images

Can we develop classification methods by emulating the brain?

Brains

$10^{11}$ neurons of $>20$ types, $10^{14}$ synapses, $1ms$–$10ms$ cycle time
Signals are noisy “spike trains” of electrical potential
Neurons communicate via spikes

Output spike roughly dependent on whether sum of all inputs reaches a threshold

Neurons as "Threshold Units"

Artificial neuron:
- $m$ binary inputs (-1 or 1), 1 output (-1 or 1)
- Synaptic weights $w_{ji}$
- Threshold $\mu_i$

$$v_i = \Theta(\sum_j w_{ji}u_j - \mu_i)$$

$\Theta(x) = 1$ if $x > 0$ and -1 if $x \leq 0$
"Perceptrons" for Classification

Fancy name for a type of layered “feed-forward” networks (no loops)

Uses artificial neurons ("units") with binary inputs and outputs

Perceptrons and Classification

Consider a single-layer perceptron

- Weighted sum forms a linear hyperplane

\[ \sum_j w_{ji} u_j - \mu_i = 0 \]

- Everything on one side of this hyperplane is in class 1 (output = +1) and everything on other side is class 2 (output = -1)

Any function that is linearly separable can be computed by a perceptron
Linear Separability

Example: AND is linearly separable

<table>
<thead>
<tr>
<th>u₁</th>
<th>u₂</th>
<th>AND</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Linear hyperplane

\[ v = 1 \text{ iff } u₁ + u₂ - 1.5 > 0 \]

Similarly for OR and NOT

How do we learn the appropriate weights given only examples of (input, output)?

Idea: Change the weights to decrease the error in output
**Perceptron Learning Rule**

Given input pair \((u, v^d)\) where \(v^d \in \{+1, -1\}\) is the desired output, adjust \(w\) and \(\mu\) as follows:

1. Calculate current output \(v\) of neuron

\[
v = \Theta\left(\sum_j w_j u_j - \mu\right) = \Theta(w^T u - \mu)
\]

2. Compute error signal \(e = (v^d - v)\)

3. Change \(w\) and \(\mu\) according to error \((v^d - v)\):
   - If input is positive and error is positive, then \(w\) not large enough \(\Rightarrow\) increase \(w\)
   - If input is positive and error is negative, then \(w\) too large \(\Rightarrow\) decrease \(w\)
   - Similar reasoning for other cases yields:

\[
\begin{align*}
w &\rightarrow w + \alpha(v^d - v)u \\
\mu &\rightarrow \mu - \alpha(v^d - v)
\end{align*}
\]

\(\alpha\) is the "learning rate" (a small positive number, e.g., 0.05)
What about the XOR function?

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Can a perceptron separate the +1 outputs from the -1 outputs?

Linear Inseparability

Perceptron with threshold units fails if classification task is not linearly separable

- Example: XOR
- No single line can separate the “yes” (+1) outputs from the “no” (-1) outputs!

Minsky and Papert’s book showing such negative results put a damper on neural networks research for over a decade!
How do we deal with linear inseparability?

Idea 1: Multilayer Perceptrons

Removes limitations of single-layer networks
  · Can solve XOR

Example: Two-layer perceptron that computes XOR

Output is +1 if and only if $x + y - 2(\theta(x + y - 1.5) - 0.5) > 0$
Multilayer Perceptron: What does it do?

\[ 1 + \frac{1}{2}x - y < 0 \]

\[ 1 + \frac{1}{2}x - y > 0 \]

\[ y = 1 + \frac{1}{2}x \]
Multilayer Perceptron: What does it do?
**Perceptrons as Constraint Satisfaction Networks**

![Diagram showing a network with inputs x and y, an output node labeled 'out', and equations defining linear separability]

**Back to Linear Separability**

- Recall: Weighted sum in perceptron forms a *linear hyperplane*

\[ \sum_{i} w_i x_i + b = 0 \]

- Due to threshold function, everything on *one side* of this hyperplane is labeled as class 1 (output = +1) and everything on *other side* is labeled as class 2 (output = -1)
Separating Hyperplane

\[ \sum w_i x_i + b = 0 \]

- denotes +1 output
- denotes -1 output

Need to choose \( w \) and \( b \) based on training data

Separating Hyperplanes

Different choices of \( w \) and \( b \) give different hyperplanes

(This and next few slides adapted from Andrew Moore’s)
Which hyperplane is best?

Class 1

Class 2

- denotes +1 output
- denotes -1 output

How about the one right in the middle?

Intuitively, this boundary seems good

Avoids misclassification of new test points if they are generated from the same distribution as training points
Define the margin of a linear classifier as the width that the boundary could be increased by before hitting a datapoint.

The maximum margin classifier is called a Support Vector Machine (in this case, a Linear SVM or LSVM). Support Vectors are those datapoints that the margin pushes up against.
Why Maximum Margin?

- Robust to small perturbations of data points near boundary
- There exists theory showing this is best for generalization to new points
- Empirically works great

Support Vector Machines: The Math

Suppose the training data points \((x_i, y_i)\) satisfy:

\[
\begin{align*}
\mathbf{w} \cdot \mathbf{x}_i + b &\geq +1 & y_i = +1 \\
\mathbf{w} \cdot \mathbf{x}_i + b &\leq -1 & y_i = -1
\end{align*}
\]

This can be rewritten as

\[
y_i (\mathbf{w} \cdot \mathbf{x}_i + b) \geq +1
\]

We can always do this by rescaling \(\mathbf{w}\) and \(b\), without affecting the separating hyperplane:

\[\mathbf{w} \cdot \mathbf{x} + b = 0\]
**Estimating the Margin**

The margin is given by (see Burges tutorial online):

\[ m = \frac{2}{||w||} \]

Margin can be calculated based on expression for distance from a point to a line, see, e.g., [http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html](http://mathworld.wolfram.com/Point-LineDistance2-Dimensional.html)

---

**Learning the Maximum Margin Classifier**

Want to maximize margin:

\[ \frac{2}{||w||} \text{ subject to } y_i (w \cdot x_i + b) \geq +1, \forall i \]

Equivalent to finding \( w \) and \( b \) that minimize:

\[ \frac{1}{2} ||w||^2 \text{ subject to } y_i (w \cdot x_i + b) \geq +1, \forall i \]

Constrained optimization problem that can be solved using **Lagrange multiplier method**
Learning the Maximum Margin Classifier

Using Lagrange formulation and Lagrangian multipliers $\alpha_i$, we get (see Burges tutorial online):

$$w = \sum \alpha_i y_i x_i$$

where the $\alpha_i$ are obtained by maximizing:

$$\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j (x_i \cdot x_j)$$

subject to $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$

This is a quadratic programming (QP) problem.

- A global maximum can always be found.

Geometrical Interpretation

$x_i$ with non-zero $\alpha_i$ are called support vectors.

$w^T x + b = 1$

$w^T x + b = -1$
What if data is not linearly separable?

Outliers (due to noise)

Approach 1: Soft Margin SVMs

Allow errors $\xi_i$ (deviations from margin)

Trade off margin with errors.

Minimize: $\frac{1}{2}\|w\|^2 + C \sum \xi_i$, subject to:

$y_i(w \cdot x_i + b) \geq 1 - \xi_i$ and $\xi_i \geq 0, \forall i$
What if data is not linearly separable: Other ideas?

<table>
<thead>
<tr>
<th>$u_1$</th>
<th>$u_2$</th>
<th>XOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>-1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>-1</td>
<td>-1</td>
</tr>
<tr>
<td>-1</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Can we do something to the inputs?

Another Example

Not linearly separable
What if data is not linearly separable?

**Approach 2:** Map original input space to higher-dimensional feature space; use linear classifier in higher-dim. space

\[ x \to \phi(x) \]

\[ \Phi : \mathbb{R}^2 \to \mathbb{R}^3 \]

\[ (x_1, x_2) \mapsto (z_1, z_2, z_3) := (x_1^2, \sqrt{2}x_1x_2, x_2^2) \]

Problem with high dimensional spaces

Computation in high-dimensional feature space can be costly

The high dimensional projection function \( \phi(x) \) may be too complicated to compute

Kernel trick to the rescue!
The Kernel Trick

Recall: SVM maximizes the quadratic function:

$$\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \langle x_i, x_j \rangle$$

subject to $\alpha_i \geq 0$ and $\sum_i \alpha_i y_i = 0$

Insight:
The data points only appear as inner product
- No need to compute high-dimensional $\phi(x)$ explicitly! Just replace inner product $x_i \cdot x_j$ with a kernel function $K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j)$
- E.g., Gaussian kernel
  $$K(x_i, x_j) = \exp(-||x_i-x_j||^2/2\sigma^2)$$
- E.g., Polynomial kernel
  $$K(x_i, x_j) = (x_i \cdot x_j + 1)^d$$

An Example for $\phi(.)$ and $K(.,.)$

Suppose $\phi(.)$ is given as follows

$$\phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) = (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

An inner product in the feature space is

$$\langle \phi\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right), \phi\left(\begin{bmatrix} y_1 \\ y_2 \end{bmatrix}\right)\rangle = (1 + x_1y_1 + x_2y_2)^2$$

So, if we define the kernel function as follows,
there is no need to compute $\phi(.)$ explicitly

$$K(x, y) = (1 + x_1y_1 + x_2y_2)^2$$

This use of kernel function to avoid computing $\phi(.)$ explicitly is known as the kernel trick
Summary: Steps for Classification using SVMs

- Prepare the data matrix
- Select the kernel function to use
- Select parameters of the kernel function
  - You can use the values suggested by the SVM software, or use cross-validation
- Execute the training algorithm and obtain the parameters $\alpha_i$
- Classify new data using the learned parameters

---

Face Detection using SVMs

Kernel used: Polynomial of degree 2

<table>
<thead>
<tr>
<th>Test Set</th>
<th>Detect Rate</th>
<th>False Alarms</th>
</tr>
</thead>
<tbody>
<tr>
<td>SVM</td>
<td>97.1%</td>
<td>4</td>
</tr>
<tr>
<td>Sung et al.</td>
<td>94.6%</td>
<td>2</td>
</tr>
</tbody>
</table>

(Osuna, Freund, Girosi, 1998)
K-Nearest Neighbors

A simple non-parametric classification algorithm

Idea:

- Look around you to see how your neighbors classify data
- Classify a new data-point according to a majority vote of your $k$ nearest neighbors
### Distance Metric

How do we measure what it means to be a neighbor (what is “close“)?

Appropriate distance metric depends on the problem

Examples:

- **x discrete (e.g., strings): Hamming distance**
  \[ d(x_1, x_2) = \# \text{ features on which } x_1 \text{ and } x_2 \text{ differ} \]

- **x continuous (e.g., vectors over reals): Euclidean distance**
  \[ d(x_1, x_2) = || x_1 - x_2 || = \text{square root of sum of squared differences between corresponding elements of data vectors} \]

---

### Example

Input Data: 2-D points \((x_1, x_2)\)

Two classes: \(C_1\) and \(C_2\).

New Data Point +

\[ K = 4 \]

K = 4: Look at 4 nearest neighbors of + 3 are in \(C_1\), so classify + as \(C_1\)
Decision Boundary using K-NN

Some points near the boundary may be misclassified (but maybe noise)

What if we want to learn continuous-valued functions?
Example: Learning to Drive

Can you use a neural network to drive?

Regression using Networks

We want networks that can learn a function
• Network maps real-valued inputs to real-valued output
• Idea: Given data, minimize errors between network’s output and desired output by changing weights

Continuous output values → Can’t use binary threshold units anymore
To minimize errors, a differentiable output function is desirable
**Sigmoidal Networks**

The most common activation function:

Sigmoid function:

$$g(a) = \frac{1}{1 + e^{-\beta a}}$$

Non-linear “squashing” function: Squashes input to be between 0 and 1. The parameter $\beta$ controls the slope.

---

**Gradient-Descent Learning (“Hill-Climbing”)**

Given training examples $(u^m, d^m)$ ($m = 1, \ldots, N$), define an error function (cost function or “energy” function)

$$E(w) = \frac{1}{2} \sum_{m} (d^m - v^m)^2$$

where $v^m = g(w^T u^m)$
Gradient-Descent Learning
("Hill-Climbing")

Would like to change $w$ so that $E(w)$ is minimized

- **Gradient Descent**: Change $w$ in proportion to $-dE/dw$ (why?)

$$w \rightarrow w - \varepsilon \frac{dE}{dw}$$

$$\frac{dE}{dw} = -\sum_m (d^m - v^m) \frac{dv^m}{dw} = -\sum_m (d^m - v^m) g'(w^T u^m) u^m$$

Derivative of sigmoid

"Stochastic" Gradient Descent

What if the inputs only arrive one-by-one?

Stochastic gradient descent approximates sum over all inputs with an "on-line" running sum:

$$w \rightarrow w - \varepsilon \frac{dE_i}{dw}$$

$$\frac{dE_i}{dw} = -(d^m - v^m) g'(w^T u^m) u^m$$

delta = error

Also known as the "delta rule" or "LMS (least mean square) rule"
But wait....

Delta rule tells us how to modify the connections from input to output (one layer network)
  - One layer networks are not that interesting (remember XOR?)
What if we have multiple layers?

Learning Multilayer Networks

$v_i = g(\sum_j W_{ji} g(\sum_k w_{kj} u_k))$

Start with random weights $W, w$

Given input $u$, network produces output $v$

Find $W$ and $w$ that minimize total squared output error over all output units (labeled $i$):

$E(W, w) = \frac{1}{2} \sum_i (d_i - v_i)^2$
Backpropagation: Output Weights

\[ E(W, w) = \frac{1}{2} \sum_i (d_i - v_i)^2 \]

\[ v_i = g(\sum_j W_{ji} x_j) \]

Learning rule for hidden-output weights \( W \):

\[ W_{ji} \rightarrow W_{ji} - \varepsilon \frac{dE}{dW_{ji}} \]  \{gradient descent\}

\[ \frac{dE}{dW_{ji}} = -(d_i - v_i) g'(\sum_j W_{ji} x_j) x_j \]  \{delta rule\}

Backpropagation: Hidden Weights

\[ E(W, w) = \frac{1}{2} \sum_i (d_i - v_i)^2 \]

\[ v_i^m = g(\sum_j W_{ji} x_j) \]

\[ x_j^m = g(\sum_k w_{kj} u_k^m) \]

Learning rule for input-hidden weights \( w \):

\[ w_{kj} \rightarrow w_{kj} - \varepsilon \frac{dE}{dw_{kj}} \]  \{chain rule\}

\[ \frac{dE}{dw_{kj}} = \left[ - \sum_m (d_i^m - v_i^m) g'(\sum_j W_{ji} x_j^m) W_{ji} \right] \cdot \left[ g'(\sum_k w_{kj} u_k^m) u_k^m \right] \]
Learning to Drive using Backprop

ALVINN (Autonomous Land Vehicle in a Neural Network)
(Albus, 1985)

Trained using human driver + camera images
After learning:
- Drove up to 70 mph on highway
- Drove cross-country largely autonomously
- Up to 22 miles without intervention

"Road features" with
One of the learned
"road features" w,
Another Example: Face Detection

Output between -1 (no face) and +1 (face present)

(Rowley, Baluja & Kanade, 1998)

Face Detection Results

(Rowley, Baluja & Kanade, 1998)
Demos: Pole Balancing and Backing up a Truck

(courtesy of Keith Grochow, CSE 599)

- Neural network learns to balance a pole on a cart
  - System:
    - 4 state variables: \( x_{\text{cart}} \), \( v_{\text{cart}} \), \( \theta_{\text{pole}} \), \( v_{\text{pole}} \)
    - 1 input: Force on cart
  - Backprop Network:
    - Input: State variables
    - Output: New force on cart

- NN learns to back a truck into a loading dock
  - System (Nyugen and Widrow, 1989):
    - State variables: \( x_{\text{cab}} \), \( y_{\text{cab}} \), \( \theta_{\text{cab}} \)
    - 1 input: new \( \theta_{\text{steering}} \)
  - Backprop Network:
    - Input: State variables
    - Output: Steering angle \( \theta_{\text{steering}} \)