**Why Use Machine Learning?**

- **advantages:**
  - often much more **accurate** than human-crafted rules (since data driven)
  - humans often incapable of expressing what they know (e.g., rules of English, or how to recognize letters), but can easily classify examples
  - don’t need a human expert or programmer
  - **flexible** — can apply to any learning task
  - **cheap** — can use in applications requiring **many** classifiers (e.g., one per customer, one per product, one per web page, ...)

- **disadvantages**
  - need a lot of **labeled** data
  - **error prone** — usually impossible to get perfect accuracy

**Machine Learning**

- studies how to **automatically learn** to make accurate **predictions** based on past observations
- **classification** problems:
  - classify examples into given set of categories

```
          labeled training examples
```

```
 new example
```

```
  machine learning algorithm
```

```
classification rule
```

```
predicted classification
```

**Machine Learning Algorithms**

- **this talk:**
  - decision trees
  - boosting
  - support-vector machines

- **others not covered:**
  - neural networks
  - nearest neighbor algorithms
  - Naive Bayes
  - bagging

**Examples of Classification Problems**

- text categorization
  - e.g.: spam filtering
  - e.g.: categorize news articles by topic
- fraud detection
- optical character recognition
- natural-language processing
  - e.g.: part-of-speech tagging
  - e.g.: spoken language understanding
- market segmentation
  - e.g.: predict if customer will respond to promotion
  - e.g.: predict if customer will switch to competitor
- medical diagnosis

**Decision Trees**
Example: Good versus Evil

- **Problem:** identify people as good or bad from their appearance

<table>
<thead>
<tr>
<th></th>
<th>sex</th>
<th>mask</th>
<th>cape</th>
<th>tie</th>
<th>ears</th>
<th>smokes</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>batman</td>
<td>male</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>Good</td>
</tr>
<tr>
<td>robin</td>
<td>male</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Good</td>
</tr>
<tr>
<td>alfred</td>
<td>male</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>Good</td>
</tr>
<tr>
<td>penguin</td>
<td>male</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>Bad</td>
</tr>
<tr>
<td>catwoman</td>
<td>female</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>Bad</td>
</tr>
<tr>
<td>joker</td>
<td>male</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>Bad</td>
</tr>
<tr>
<td>batgirl</td>
<td>female</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>??</td>
</tr>
<tr>
<td>riddler</td>
<td>male</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>??</td>
</tr>
</tbody>
</table>

Choosing the Splitting Rule

- Choose rule that leads to greatest increase in “purity”:

![Decision Tree Diagram]

Example (cont.)

- (in)purity measures:
  - entropy: \(-p_+ \ln p_+ - p_- \ln p_-\)
  - Gini index: \(p_+ p_-\)
  - where \(p_+ / p_-\) = fraction of positive / negative examples

Choosing the Splitting Rule (cont.)

- [Symmetry Graph]
  - \(p_+ = 1 - p_-\)

How to Build Decision Trees

- Choose rule to split on
- Divide data using splitting rule into disjoint subsets
- Repeat recursively for each subset
- Stop when leaves are (almost) “pure”

Kinds of Error Rates

- **Training error** = fraction of training examples misclassified
- **Test error** = fraction of test examples misclassified
- **Generalization error** = probability of misclassifying new random example
Tree Size versus Accuracy

- trees must be big enough to fit training data (so that “true” patterns are fully captured)
- BUT: trees that are too big may overfit (capture noise or spurious patterns in the data)
- significant problem: can’t tell best tree size from training error

Example

Training data:

Good and Bad Classifiers

- sufficient data
  - low training error
  - simple classifier

- insufficient data
  - training error too high
  - classifier too complex

Overfitting Example

- fitting points with a polynomial

underfit (degree = 1) ideal fit (degree = 3) overfit (degree = 20)

Building an Accurate Classifier

- for good test performance, need:
  - enough training examples
  - good performance on training set
  - classifier that is not too “complex” (“Occam’s razor”)
- measure “complexity” by:
  - number of bits needed to write down
  - number of parameters
  - VC-dimension

Theory

- can prove:

\[
\text{(generalization error)} \leq \text{(training error)} + O \left( \frac{d}{\sqrt{m}} \right)
\]

with high probability
- \(d = \text{VC-dimension}\)
- \(m = \text{number training examples}\)
Controlling Tree Size

- typical approach: build very large tree that fully fits training data, then prune back

- pruning strategies:
  * grow on just part of training data, then find pruning with minimum error on held out part
  * find pruning that minimizes
    \[(\text{training error}) + \text{constant} \times (\text{tree size})\]

Example: Spam Filtering

- problem: filter out spam (junk email)
- gather large collection of examples of spam and non-spam:
  
  From: yoav@att.com   Rob, can you review a paper... non-spam
  From: xa412@hotmail.com   Earn money without working!!! spam

- main observation:
  * easy to find “rules of thumb” that are “often” correct
  * If “buy now” occurs in message, then predict ‘spam’
  * hard to find single rule that is very highly accurate

Decision Trees

- best known:
  * C4.5 (Quinlan)
  * CART (Breiman, Friedman, Olshen & Stone)
- very fast to train and evaluate
- relatively easy to interpret
- but: accuracy often not state-of-the-art

The Boosting Approach

- devise computer program for deriving rough rules of thumb
- apply procedure to subset of emails
- obtain rule of thumb
- apply to 2nd subset of emails
- obtain 2nd rule of thumb
- repeat $T$ times

Details

- how to choose examples on each round?
  * concentrate on “hardest” examples
  * (those most often misclassified by previous rules of thumb)
- how to combine rules of thumb into single prediction rule?
  * take (weighted) majority vote of rules of thumb
Boosting

- Boosting = general method of converting rough rules of thumb into highly accurate prediction rule
- Technically:
  - Assume given "weak" learning algorithm that can consistently find classifiers ("rules of thumb") at least slightly better than random, say, accuracy ≥ 55%
  - Given sufficient data, a boosting algorithm can provably construct single classifier with very high accuracy, say, 99%

AdaBoost

- Given training examples \((x_i, y_i)\) where \(y_i \in \{-1, +1\}\)
- Initialize \(D_1 = \text{uniform distribution on training examples}\)
- For \(t = 1, \ldots, T\):
  - Train weak classifier ("rule of thumb") \(h_t\) on \(D_t\)
  - Choose \(\alpha_t > 0\)
  - Compute new distribution \(D_{t+1}\):
    - For each example \(i\):
      - Multiply \(D_t(x_i)\) by \(\frac{1}{\text{if } y_i = h_t(x_i)} \frac{1}{\text{if } y_i \neq h_t(x_i)}\)
    - Renormalize
- Output final classifier \(H_{\text{final}}(x) = \text{sign} \left( \sum_t \alpha_t h_t(x) \right)\)

Toy Example

- Weak classifiers = vertical or horizontal half-planes
**Final Classifier**

$$H_{\text{final}} = \text{sign} \left( 0.42 + 0.65 + 0.92 \right)$$

**Actual Typical Run**

- Test error does not increase, even after 1000 rounds
  - (total size > 2,000,000 nodes)
- Test error continues to drop even after training error is zero

<table>
<thead>
<tr>
<th># of rounds</th>
<th>Train error</th>
<th>Test error</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0.0</td>
<td>8.4</td>
</tr>
<tr>
<td>100</td>
<td>0.0</td>
<td>3.3</td>
</tr>
<tr>
<td>1000</td>
<td>0.0</td>
<td>3.1</td>
</tr>
</tbody>
</table>

**Theory: Training Error**

- Weak learning assumption: each weak classifier at least slightly better than random
  - i.e., (error of $h_t$ on $D_t$) $\leq 1/2 - \gamma$ for some $\gamma > 0$
- Given this assumption, can prove:
  $$\text{training error}(H_{\text{final}}) \leq e^{-2\gamma^2T}$$

**The Margins Explanation**

- Key idea:
  - Training error only measures whether classifications are right or wrong
  - Should also consider confidence of classifications
- Recall: $H_{\text{final}}$ is weighted majority vote of weak classifiers
- Measure confidence by margin = strength of the vote
- Empirical evidence and mathematical proof that:
  - Large margins $\Rightarrow$ better generalization error (regardless of number of rounds)
  - Boosting tends to increase margins of training examples (given weak learning assumption)

**How Will Test Error Behave? (A First Guess)**

- Expect:
  - Training error to continue to drop (or reach zero)
  - Test error to increase when $H_{\text{final}}$ becomes “too complex” (overfitting)

**Boosting**

- Fast (but not quite as fast as other methods)
- Simple and easy to program
- Flexible: can combine with any learning algorithm, e.g., C4.5
  - Very simple rules of thumb
- Provable guarantees
- State-of-the-art accuracy
- Tends not to overfit (but occasionally does)
- Many applications
**Support-Vector Machines**

**Finding the Maximum Margin Hyperplane**

- examples \( x_i, y_i \) where \( y_i \in \{-1, +1\} \)
- find hyperplane \( w \cdot x = 0 \) with \( \| w \| = 1 \)
- margin = \( y(w \cdot x) \)
- maximize: \( \gamma \)
  subject to: \( y_i(w \cdot x_i) \geq \gamma \) and \( \| w \| = 1 \)
- set \( w \leftarrow \frac{w}{\gamma} \Rightarrow \gamma = 1/\| w \| \)
- minimize \( \frac{1}{2} \| w \|^2 \)
  subject to: \( y_i(w \cdot x_i) \geq 1 \)

**Geometry of SVM's**

- given linearly separable data
- margin = distance to separating hyperplane
- choose hyperplane that maximizes minimum margin
- intuitively:
  - want to separate +’s from −’s as much as possible
  - margin = measure of confidence

**Convex Dual**

- form Lagrangian, set \( \partial / \partial w = 0 \)
- get quadratic program:
  maximize \( \sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j x_i \cdot x_j \)
  subject to: \( \alpha_i \geq 0 \)
- \( w = \sum_i \alpha_i y_i x_i \)
- \( \alpha_i = \) Lagrange multiplier
  \( > 0 \) ⇔ support vector
- key points:
  - optimal \( w \) is linear combination of support vectors
  - dependence on \( x_i \)'s only through inner products
  - maximization problem is convex with no local maxima

**Theoretical Justification**

- let \( \gamma = \) minimum margin
  \( R = \) radius of enclosing sphere
- then
  \[ \text{VC-dim} \leq \left( \frac{R}{\gamma} \right)^2 \]
- so larger margins ⇒ lower “complexity”
- independent of number of dimensions
- in contrast, unconstrained hyperplanes in \( \mathbb{R}^n \) have
  \[ \text{VC-dim} = (\# \text{parameters}) = n + 1 \]

**What If Not Linearly Separable?**

- answer #1: penalize each point by distance from margin 1, i.e., minimize:
  \[ \frac{1}{2} \| w \|^2 + \text{constant} \cdot \sum_i \max\{0, 1 - y_i(w \cdot x_i)\} \]
- answer #2: map into higher dimensional space in which data becomes linearly separable
Example

- not linearly separable
- map \( \mathbf{x} = (x_1, x_2) \mapsto \Phi(\mathbf{x}) = (1, x_1, x_2, x_1^2, x_2^2) \)
- hyperplane in mapped space has form
  \[
  a + bx_1 + cx_2 + dx_1x_2 + ex_1^2 + fx_2^2 = 0
  \]
  = conic in original space
- linearly separable in mapped space

Kernels

- kernel = function \( K \) for computing
  \[
  K(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x}) \cdot \Phi(\mathbf{z})
  \]
- permits efficient computation of SVM's in very high dimensions
- \( K \) can be any symmetric, positive semi-definite function
  (Mercer's theorem)
- some kernels:
  - polynomials
  - Gaussian \( \exp(-||\mathbf{x} - \mathbf{z}||^2/2\sigma) \)
  - defined over structures (trees, strings, sequences, etc.)
- evaluation:
  \[
  \mathbf{w} \cdot \Phi(\mathbf{x}) = \sum \alpha_i y_i \Phi(\mathbf{x}_i) \cdot \Phi(\mathbf{x}) = \sum \alpha_i y_i K(\mathbf{x}_i, \mathbf{x})
  \]
  - time depends on # support vectors

Higher Dimensions Don’t (Necessarily) Hurt

- may project to very high dimensional space
- statistically, may not hurt since VC-dimension independent of
  number of dimensions \((R/\gamma)^2\)
- computationally, only need to be able to compute inner products
  \( \Phi(\mathbf{x}) \cdot \Phi(\mathbf{z}) \)
  - sometimes can do very efficiently using kernels

SVM's versus boosting

- both are large-margin classifiers
  (although with slightly different definitions of margin)
- both work in very high dimensional spaces
  (in boosting, dimensions correspond to weak classifiers)
- but different tricks are used:
  - SVM’s use kernel trick
  - boosting relies on weak learner to select one dimension (i.e., weak
    classifier) to add to combined classifier

Example (cont.)

- modify \( \Phi \) slightly:
  \[
  \Phi(\mathbf{x}) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2)
  \]
- then
  \[
  \Phi(\mathbf{x}) \cdot \Phi(\mathbf{z}) = 1 + 2x_1z_1 + 2x_2z_2 + 2x_1x_2z_1z_2 + x_1^2z_1^2 + x_2^2z_2^2 + x_1^2 + x_2^2
  = (1 + x_1z_1 + x_2z_2)^2
  = (1 + \mathbf{x} \cdot \mathbf{z})^2
  \]
- in general, for polynomial of degree \( d \), use \((1 + \mathbf{x} \cdot \mathbf{z})^d\)
- very efficient, even though finding hyperplane in \( O(n^d) \) dimensions

SVM’s

- fast algorithms now available, but not so simple to program
  (but good packages available)
- state-of-the-art accuracy
- power and flexibility from kernels
- theoretical justification
- many applications
Further reading on machine learning in general:

Decision trees:

Boosting:

Support-vector machines:

Many more papers, tutorials, etc. available at www.boosting.org.