Machine Learning Algorithms for Classification

Rob Schapire
Princeton University

www.cs.princeton.edu/~schapire
Machine Learning

- studies how to **automatically learn** to make accurate **predictions** based on past observations

- **classification** problems:
  - classify examples into given set of categories
Examples of Classification Problems

- text categorization (e.g., spam filtering)
- fraud detection
- optical character recognition
- machine vision (e.g., face detection)
- natural-language processing (e.g., spoken language understanding)
- market segmentation (e.g., predict if customer will respond to promotion)
- bioinformatics (e.g., classify proteins according to their function)
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 Algorithms**

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

- **others not covered:**
  - nearest neighbor algorithms
  - Naive Bayes
  - bagging
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>
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”
Choosing the Splitting Rule

- choose rule that leads to greatest increase in “purity”:
Choosing the Splitting Rule (cont.)

- (im)purity measures:
  - entropy: $-p \ln p - q \ln q$
  - Gini index: $pq$

where $p$ ($q$) = fraction of positive (negative) examples

\[ p = 1 - q \]
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
A Possible Classifier

- perfectly classifies training data
- BUT: intuitively, overly complex
Another Possible Classifier

- overly simple
- doesn’t even fit available data
significant problem: can’t tell best tree size from training error

• overfit
• 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
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 bits needed to write down
      - number of parameters
      - VC-dimension
Example

Training data:
Good and Bad Classifiers

Good:
- sufficient data
- low training error
- simple classifier

Bad:
- insufficient data
- training error too high
- classifier too complex
Theory

- can prove:

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

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
    \[(\text{training error}) + \text{constant} \cdot (\text{tree size})\]
**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
Boosting
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
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 $\geq 55\%$
  - 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 \textbf{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 \(e^{\alpha_t} \begin{cases} < 1 & \text{if } y_i = h_t(x_i) \\ > 1 & \text{if } y_i \neq h_t(x_i) \end{cases}\)
    - renormalize
- output \textbf{final classifier} \(H_{\text{final}}(x) = \text{sign} \left( \sum_t \alpha_t h_t(x) \right)\)
**Toy Example**

$D_1$

weak classifiers = vertical or horizontal half-planes
Round 1

$h_1$

$\varepsilon_1 = 0.30$
$\alpha_1 = 0.42$

$D_2$
Round 2

\[ \varepsilon_2 = 0.21 \]
\[ \alpha_2 = 0.65 \]
Round 3

\[ \varepsilon_3 = 0.14 \]
\[ \alpha_3 = 0.92 \]
Final Classifier

\[ H_{\text{final}} = \text{sign} \left( \begin{array}{ccc} +0.42 & +0.65 & +0.92 \\ +0.42 & +0.65 & +0.92 \\ +0.42 & +0.65 & +0.92 \\ +0.42 & +0.65 & +0.92 \end{array} \right) \]
Theory: Training Error

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

\[
\text{training error}(H_{\text{final}}) \leq e^{-2\gamma^2 T}
\]
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)
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># rounds</th>
<th>5</th>
<th>100</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>train error</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>test error</td>
<td>8.4</td>
<td>3.3</td>
<td>3.1</td>
</tr>
</tbody>
</table>

(Boosting C4.5 on “letter” dataset)
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)
**Application: Detecting Faces**

- Problem: find faces in photograph or movie
- Weak classifiers: detect light/dark rectangles in image

- Many clever tricks to make extremely fast and accurate
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
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
Theoretical Justification

- let $\gamma = \text{minimum margin}$
  
  $R = \text{radius of enclosing sphere}$

- then

  $$\text{VC-dim} \leq \left( \frac{R}{\gamma} \right)^2$$

- so larger margins $\Rightarrow$ lower “complexity”
  - independent of number of dimensions

- in contrast, unconstrained hyperplanes in $\mathbb{R}^n$ have

  $$\text{VC-dim} = (\# \text{ parameters}) = n + 1$$
Finding the Maximum Margin Hyperplane

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

- form Lagrangian, set $\partial / \partial \mathbf{w} = 0$
- get quadratic program:
  - maximize $\sum_i \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \cdot \mathbf{x}_j$
    subject to: $\alpha_i \geq 0$
  - $\mathbf{w} = \sum_i \alpha_i y_i \mathbf{x}_i$
  - $\alpha_i =$ Lagrange multiplier
    $> 0 \Rightarrow$ support vector
- key points:
  - optimal $\mathbf{w}$ is linear combination of support vectors
  - dependence on $\mathbf{x}_i$’s only through inner products
  - maximization problem is convex with no local maxima
What If Not Linearly Separable?

- **answer #1**: penalize each point by distance from margin 1, i.e., minimize:
  \[
  \frac{1}{2} \| \mathbf{w} \|^2 + \text{constant} \cdot \sum_i \max\{0, 1 - y_i (\mathbf{w} \cdot \mathbf{x}_i)\}
  \]

- **answer #2**: map into higher dimensional space in which data becomes linearly separable
Example

- not linearly separable
- map \( \mathbf{x} = (x_1, x_2) \mapsto \Phi(\mathbf{x}) = (1, x_1, x_2, x_1x_2, x_1^2, x_2^2) \)
- 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
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 $(\frac{R}{\gamma})^2$
- computationally, only need to be able to compute inner products $\Phi(x) \cdot \Phi(z)$

- sometimes can do very efficiently using kernels
Example (cont.)

• modify $\Phi$ slightly:

$$\Phi(x) = (1, \sqrt{2}x_1, \sqrt{2}x_2, \sqrt{2}x_1x_2, x_1^2, x_2^2)$$

• then

$$\Phi(x) \cdot \Phi(z) = 1 + 2x_1z_1 + 2x_2z_2 + 2x_1x_2z_1z_2 + x_1^2z_1^2 + x_2^2 + z_2^2$$

$$= (1 + x_1z_1 + x_2z_2)^2$$

$$= (1 + x \cdot z)^2$$

• in general, for polynomial of degree $d$, use $(1 + x \cdot z)^d$

• very efficient, even though finding hyperplane in $O(n^d)$ dimensions
Kernels

- kernel = function $K$ for computing
  \[ K(x, z) = \Phi(x) \cdot \Phi(z) \]

- 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 $\exp \left( - \| x - z \|^2 / 2\sigma \right)$
  - defined over structures (trees, strings, sequences, etc.)

- evaluation:
  \[ w \cdot \Phi(x) = \sum \alpha_i y_i \Phi(x_i) \cdot \Phi(x) = \sum \alpha_i y_i K(x_i, x) \]

- time depends on # support vectors
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
Application: Text Categorization

• **goal**: classify text documents
  * e.g.: spam filtering
  * e.g.: categorize news articles by topic

• need to represent text documents as vectors in $\mathbb{R}^n$:
  * one dimension for each word in vocabulary
  * value = # times word occurred in particular document
  * (many variations)

• kernels don’t help much

• performance state of the art
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
Neural Networks
The Neural Analogy

- perceptron (= linear threshold function) looks a lot like a neuron

- other neurons fire (inputs)
- when electrical potential exceeds threshold, fires (output)

- inputs: $a_1, \ldots, a_n \in \{0, 1\}$
- weights: $w_1, \ldots, w_n \in \mathbb{R}$
- "activation" = \[ 1 \text{ if } \sum w_i a_i > \theta \]
  \[ 0 \text{ else} \]
A Network of Neurons

- **idea**: put perceptrons in network

- weights on every edge
- each unit = perceptron
- dramatic increase in representation power (not necessarily a good thing for learning)
- great flexibility in choice of architecture
Perceptron Units

- problem: overall network computation is horribly discontinuous because of $g$
  - optimizing network weights easier when everything continuous
Smoothed Threshold Functions

- **idea**: approximate $g$ with **smoothed** threshold function

- e.g., use $g(x) = \frac{1}{1 + e^{-x}}$

- now $h_w(x)$ is continuous and differentiable in both inputs $x$ and weights $w$
Finding Weights

• given \((x_1, y_1), \ldots, (x_m, y_m)\) where \(y_i \in \{0, 1\}\)
• how to find weights \(w\)?
• want network output \(h_w(x_i)\) “close” to \(y_i\)
• typical measure of closeness:

“energy” \(E(w) = \sum_i (h_w(x_i) - y_i)^2\)
Minimizing Energy

- $E$ is a continuous and differentiable function of $w$
- minimize using gradient descent:
  - start with any $w$
  - repeatedly adjust $w$ by taking tiny steps in direction of steepest descent
- easy to compute gradients
  - turns out to have simple recursive form in which error signal is backpropagated from output to inputs
Implementation Details

- often do gradient descent step based just on single example (and repeat for all examples in training set)
- often slow to converge
  - speed up using techniques like conjugate gradient descent
- can get stuck in local minima or large flat regions
- can overfit
  - use regularization to keep weights from getting too large

\[
E(w) = \sum_i (h_w(x_i) - y_i)^2 + \beta \|w\|^2
\]
Application: Optical Character Recognition

[LeCun, Bottou, Bengio & Haffner]

- problem: recognize handwritten characters
- LeNet-5:
  - 7 layers (plus inputs) specially designed for OCR
  - extended for segmentation
  - very high accuracy
Neural Nets

- can be slow to converge
- can be difficult to get right architecture, and difficult to tune parameters
- not state-of-the-art as a general method
- with proper care, can do very well on particular problems, often with specialized architecture
Further reading on machine learning in general:


Decision trees:


Boosting:


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

Support-vector machines:


Many more papers, tutorials, etc. available at www.kernel-machines.org.

Neural nets: