Combining Classifiers

- Generic methods of generating and combining multiple classifiers
  - Bagging
  - Boosting

- References:
  - Duda, Hart & Stork, pg 475-480.
  - Hastie, Tibshirani, Friedman, pg 246-256 and Chapter 10.
  - http://www.boosting.org/
    - Bulletin Board
      - "Is there a book available on boosting?"

- Stacking
  - "meta-learn" which classifier does well where

- Error-correcting codes
  - going from binary to multi-class problems
Why Combine Classifiers?

- Combine several classifiers to produce a more accurate single classifier.
- If $C_2$ and $C_3$ are correct where $C_1$ is wrong, etc, majority vote will do better than each $C_i$ individually.
- Suppose:
  - each $C_i$ has error rate $p < 0.5$
  - errors of different $C_i$ are uncorrelated.
- Then $\Pr(r \text{ out of } n \text{ classifiers are wrong}) = \binom{n}{r} p^r (1-p)^{n-r}$

Pr(majority of $n$ classifiers are wrong) is small if:
- $n$ is large
- $p$ is small
Bagging

- "Bootstrap aggregation"
- Bootstrap estimation - generate data set by randomly selecting from training set:
  - with replacement (some points may repeat)
  - repeat $B$ times
  - use as estimate the average of individual estimates

- Bagging
  - generate $B$ equal size training sets
  - each training set:
    - is drawn randomly, with replacement, from the data
    - is used to generate a different component classifier $f_i$
      - usually using same algorithm (e.g. decision tree)
  - final classifier decides by voting among component classifiers

- Leo Breiman, 1996.
Suppose there are \( k \) classes

- Each \( f_i(x) \) predicts 1 of the classes
- Equivalently, \( f_i(x) = (0, 0, ..., 0, 1, 0, ..., 0) \)

Define \( f_{bag}(x) = \frac{1}{B} \sum_{i=1}^{B} f_i(x) \)

\[
= (p_1(x), ..., p_k(x)), \\
p_j(x) = \text{proportion of } f_i \text{ predicting class } j \text{ at } x
\]

Bagged prediction is

- \( \arg \max_k f_{bag}(x) \)

Reduces variance

- always (provable) for squared-error
- not always for classification (0/1 loss)
- In practice usually most effective if classifiers are "unstable" - depend sensitively on training points.

However may lose interpretability

- a bagged decision tree is not a single decision tree
Boosting

- Generate the component classifiers so that each does well where the previous ones do badly
  - Train classifier $C_1$ using (some part of) the training data
  - Train classifier $C_2$ so that it performs well on points where $C_1$ performs badly
  - Train classifier $C_3$ to perform well on data classified badly by $C_1$ and $C_2$, etc.

- Overall classifier $C$ classifies by weighted voting among the component classifiers $C_i$

- The same algorithm is used to generate each $C_i$ - only the data used for training changes
AdaBoost

- "Adaptive Boosting"
- Give each training point \((x_i, y_i = \pm 1)\) in \(D\) a weight \(w_i\) (initialized uniformly)
- Repeat:
  - Draw a training set \(D_m\) at random from \(D\) according to the weights \(w_i\)
  - Generate classifier \(C_m\) using training set \(D_m\)
  - Measure error of \(C_m\) on \(D\)
    - Increase weights of misclassified training points
    - Decrease weights of correctly classified points
- Overall classification is determined by
  - \(C_{boost}(x) = \text{Sign}(\sum_m \alpha_m C_m(x))\), where
  - \(\alpha_m\) measures the "quality" of \(C_m\)
- Terminate when \(C_{boost}(x)\) has low error
AdaBoost (Details)

- Initialize weights uniformly: $w_i^1 = 1/N \ (N=\text{training set size})$
- Repeat for $m=1,2, \ldots, M$
  - Draw random training set $D_m$ from $D$ according to weights $w_i^m$
  - Train classifier $C_m$ using training set $D_m$
  - Compute $err_m = \Pr_{i \sim D_m} [C_m(x_i) \neq y_i]$
    - error rate of $C_m$ on (weighted) training points
  - Compute $a_m = 0.5 \log((1-err_m)/err_m)$
    - $a_m = 0$ when $err_m = 0.5$
    - $a_m \to \infty$ as $err_m \to 0$
  - $w_i^{m^*} = w_i^m \exp(a_m) = w_i^m \sqrt{(1-err_m)/\text{err}_m}$ if $x_i$ is incorrectly classified
    - $w_i^{m^*} \exp(-a_m) = w_i^m \sqrt{\text{err}_m/(1-err_m)}$ if $x_i$ is correctly classified
  - $w_i^{m+1} = w_i^{m^*}/Z_m$
    - $Z_m = \Sigma_i w_i^{m^*}$ is a normalization factor so that $\Sigma_i w_i^{m+1} = 1$
- Overall classification is determined by
  - $C_{\text{boost}}(x) = \text{Sign}(\Sigma_m a_m C_m(x))$
Theory

If:
- each component classifier $C_m$ is a "weak learner"
  - performs better than random chance ($\text{err}_m < 0.5$)

Then:
- the TRAINING SET ERROR of $C_{\text{boost}}$ can be made arbitrarily small as $M$ (the number of boosting rounds) $\to \infty$

Proof (see Later)
- Probabilistic bounds on the TEST SET ERROR can be obtained as a function of training set error, sample size, number of boosting rounds, and "complexity" of the classifiers $C_m$
- If Bayes Risk is high, it may become impossible to continually find $C_m$ which perform better than chance.
- "In theory theory and practice are the same, but in practice they are different"
Practice

- Use an independent test set to determine stopping point
- Boosting performs very well in practice
  - Fast
  - Boosting decision "stumps" is competitive with decision trees
  - Test set error may continue to fall even after training set error=0
  - Does not (usually) overfit
  - Sometimes vulnerable to outliers/noise
  - Result may be difficult to interpret
- "AdaBoost with trees is the best off-the-shelf classifier in the world" - Breiman, 1996.
History

- Robert Schapire, 1989
  - Weak classifier could be boosted

- Yoav Freund, 1995
  - Boost by combining many weak classifiers
  - Required bound on error rate of weak classifier

- Freund & Schapire, 1996
  - AdaBoost - adapts weights based on error rate of weak classifier

- Many extensions since then
  - Boosting Decision Trees, Naive Bayes, ...
  - More robust to noise
  - Improving interpretability of boosted classifier
  - Incorporating prior knowledge
  - Extending to multi-class case
  - "Balancing between Boosting and Bagging using Bumping"
  - ......
Proof

- **Claim:** If $\text{err}_m < 0.5$ for all $m$, then Training Set Error of $C_{\text{boost}}$ ~$0$ as $M$~$\infty$
- **Note:** $y_i C_m(x_i) = 1$ if $x_i$ is correctly classified by $C_m$
  = -1 if $x_i$ is incorrectly classified by $C_m$,
  similarly for $C_{\text{boost}}(x) = \text{sign}(\Sigma_m a_m C_m(x))$
- **Training Set Error of classifier $C_{\text{boost}}(x)$ is**
  \[\text{err}_{\text{boost}} = \frac{|\{i:C_{\text{boost}}(x_i) \neq y_i\}|}{N}\]
- $C_{\text{boost}}(x_i) \neq y_i$ if and only if $y_i \Sigma_m a_m C_m(x) < 0$
  if and only if $-y_i \Sigma_m a_m C_m(x) > 0$
- **Hence** $C_{\text{boost}}(x_i) \neq y_i \Rightarrow \exp(-y_i \Sigma_m a_m C_m(x)) > 1$
  so $\text{err}_{\text{boost}} < \frac{\left[\Sigma_i \exp(-y_i \Sigma_m a_m C_m(x))\right]}{N}$
- **By definition,** $w_{i}^{m+1} = w_i^m \exp(-y_i a_m C_m(x)) / Z_m$
- **So** $\exp(-y_i a_m C_m(x)) = Z_m w_{i}^{m+1} / w_i^m$
- **Now insert the "sum" into the exponential:**
  \[
  \exp(-y_i \Sigma_m a_m C_m(x)) = \prod_m \exp(-y_i a_m C_m(x))
  = \prod_m Z_m w_{i}^{m+1} / w_i^m
  = w_i^{M+1} / w_i^1 \prod_m Z_m
  = N w_i^{M+1} \prod_m Z_m
  \]
Proof (continued)

- Thus \[ \frac{\sum_i \exp(-y_i \sum_m a_m C_m(x))}{N} = \sum_i w_i^{M+1} \Pi_m Z_m \]
  \[= \Pi_m Z_m \]
  because \( \sum_i w_i^{M+1} = 1 \) (having been normalized by \( Z_m \))

- Nothing has been said so far about the choice of \( a_m \)

- Set \( a_m = 0.5 \log((1-\text{err}_m)/\text{err}_m) \)

- Then \( w_i^{m^*} = w_i^m \frac{\sqrt{1-\text{err}_m}}{\text{err}_m} \) if \( x_i \) is incorrectly classified
  \[w_i^m \sqrt{\frac{\text{err}_m}{1-\text{err}_m}}\] if \( x_i \) is correctly classified

- To normalize, set \( Z_m = \sum_i w_i^{m^*} \)
  \[= \sum_i w_i^m [\text{err}_m (\sqrt{1-\text{err}_m})/\text{err}_m] + (1-\text{err}_m) \sqrt{\text{err}_m/(1-\text{err}_m)}] \]
  \[= \sum_i w_i^m [\sqrt{\text{err}_m (1-\text{err}_m)} + \sqrt{\text{err}_m (1-\text{err}_m}]) \]
  \[= 2 \sqrt{\text{err}_m (1-\text{err}_m)} \]
  because \( \sum_i w_i^m = 1 \)

- So \( \text{err}_{\text{boost}} < \frac{\sum_i \exp(-y_i \sum_m a_m C_m(x))}{N} = \Pi_m Z_m = \Pi_m 2 \sqrt{\text{err}_m (1-\text{err}_m)} \)

NOTE: D, H & S, pg 479, says \( \text{err}_{\text{boost}} = \Pi_m 2 \sqrt{\text{err}_m (1-\text{err}_m)} \)
Proof (continued)

- Let $\gamma_m = 0.5 - \text{err}_m > 0$ for all $m$
  - $\gamma_m$ is the "edge" of $C_m$ over random guessing
- Then $2\sqrt{\text{err}_m(1-\text{err}_m)} = 2\sqrt{(0.5-\gamma_m)(0.5+\gamma_m)}$
  - $= \sqrt{1-4\gamma_m^2}$
- So $\text{err}_{\text{boost}} < \Pi_m \sqrt{1-4\gamma_m^2}$
  - $< \Pi_m(1-2\gamma_m^2)$ since $(1-x)^{0.5} = 1-0.5x-...$
  - $< \Pi_m \exp(-2\gamma_m^2)$ since $1+x < \exp(x)$
  - $= \exp(-2\sum_m \gamma_m^2)$

- If:
  - $\gamma_m > \gamma > 0$ for all $m$
- Then
  - $\text{err}_{\text{boost}} < \exp(-2\sum_m \gamma^2)$
    - $= \exp(-2M\gamma^2)$
    - which tends to zero exponentially fast as $M->\infty$
Why Boosting Works


Additive models:
- \( f(x) = \sum_m a_m b(x; \theta_m) \)
  - Classify using Sign(\( f(x) \))
- \( b \) = "basis" function parametrized by \( \theta \)
- \( a_m \) are weights

Examples:
- neural networks
  - \( b \) = activation function, \( \theta \) = input-to-hidden weights
- support vector machines
  - \( b \) = kernel function, appropriately parametrized
- boosting
  - \( b \) = weak classifier, appropriately parametrized
Fitting Additive Models

- To fit $f(x) = \sum a_m b(x; \theta_m)$, usually $a_m, \theta_m$ are found by minimizing a loss function (e.g. squared error) over the training set.

- Forward Stagewise fitting:
  - Add new basis functions to the expansion one-by-one
  - Do not modify previous terms

- Algorithm:
  - $f_0(x) = 0$
  - For $m=1$ to $M$:
    - Find $a_m, \theta_m$ by $\min_{a,\theta} \sum_i L(y_i, f_{m-1}(x) + ab(x_i; \theta))$
    - Set $f_m(x) = f_{m-1}(x) + a_m b(x; \theta_m)$

- AdaBoost is Forward Stagewise fitting applied to the weak classifier with an EXPONENTIAL loss function.
AdaBoost (Derivation)

- $L(y,f(x)) = \exp(-yf(x))$ exponential loss
- $a_m, C_m = \arg \min_{a,c} \sum_i \exp(-y_i(f_{m-1}(x_i)+aC(x_i)))$
  
  $= \arg \min_{a,c} \sum_i \exp(-y_i(f_{m-1}(x_i)))\exp(-ay_iC(x_i))$
  
  $= \arg \min_{a,c} \sum_i w_i^m \exp(-ay_iC(x_i))$
  
  where $w_i^m = \exp(-y_i(f_{m-1}(x_i)))$

  $w_i^m$ depends on neither $\alpha$ nor $C$.

- Note: $\sum_i w_i^m \exp(-ay_iC(x_i))$
  
  $= e^{-a\sum_{y_i=C(x_i)} w_i^m} + e^{a\sum_{y_i\neq C(x_i)} w_i^m}$
  
  $= e^{-a\sum_i w_i^m} + (e^a - e^{-a})\sum_i w_i^m \text{Ind}(y_i\neq C(x_i))$

- For $\alpha>0$, pick $C_m = \arg \min_C \sum_i w_i^m \text{Ind}(y_i\neq C(x_i))$

  $= \arg \min_C \text{err}_m$
AdaBoost (Derivation) (continued)

Substitute back:
- yields $e^{-a \sum_i w_i^m} + (e^a - e^{-a}) \text{err}_m$
  - a function of $a$ only
- $\arg \min_a e^{-a \sum_i w_i^m} + (e^a - e^{-a}) \text{err}_m$ can be found
  - differentiate, etc - Exercise!
- giving $a_m = 0.5 \log((1 - \text{err}_m)/\text{err}_m$

The model update is: $f_m(x) = f_{m-1}(x) + a_mC_m(x_i)$

$w_i^{m+1} = \exp(-y_i(f_m(x_i)))$
  $= \exp(-y_i(f_{m-1}(x_i) + a_mC_m(x_i)))$
  $= \exp(-y_i(f_{m-1}(x_i))) \exp(-y_ia_mC_m(x_i))$
  $= w_i^m \exp(-a_ymC_m(x_i))$

deriving the weight update rule.
**Exponential Loss**

- $L_1(y,f(x)) = \exp(-yf(x))$  
  - Exponential loss
- $L_2(y,f(x)) = \text{Ind}(yf(x)<0)$  
  - 0/1 loss
- $L_3(y,f(x)) = (y-f(x))^2$  
  - Squared error

Exponential loss puts heavy weight on examples with large negative margin. These are difficult, atypical, training points - boosting is sensitive to outliers.
Boosting and SVMs

The margin of $(x_i, y_i)$ is
$$y_i \frac{\sum a_m C_m(x_i)}{\sum |a_m|} = y_i \frac{\alpha \cdot C(x_i)}{\|\alpha\|}$$
- lies between -1 and 1
- >0 if and only if $x_i$ is classified correctly

Large margins on the training set yield better bounds on generalization error

It can be argued that boosting attempts to (approximately) maximize the minimum margin
- $\max_a \min_i y_i (\alpha \cdot C(x_i))/\|\alpha\|$
- same expression as SVM, but 1-norm instead of 2-norm
Stacking

Stacking = "stacked generalization"

Usually used to combine models $l_1, \ldots, l_r$ of different types

- e.g. $l_1$ = neural network,
- $l_2$ = decision tree,
- $l_3$ = Naive Bayes,
- ...

Use a "meta-learner" $L$ to learn which classifier is best where

Let $x$ be an instance for the component learners

Training instance for $L$ is of the form

- $(l_1(x), \ldots, l_r(x))$,
  - $l_i(x)$ = class predicted by classifier $l_i$

OR

- $(l_{11}(x), \ldots, l_{1k}(x), \ldots, l_{r1}(x), \ldots, l_{rk}(x))$,
  - $l_{ij}(x)$ = probability $x$ is in class $j$ according to classifier $l_i$
What should class label for L be?
- actual label from data
  - may prefer classifiers that overfit
- use a "hold-out" data set which is not used to train the $l_1, \ldots, l_r$
  - wastes data
- use cross-validation
  - when x occurs in the test set, use it as a training instance for L
  - computationally expensive

Use simple linear models for L

Error-correcting Codes

- Using binary classifiers to predict multi-class problem
- Generate one binary classifier $C_i$ for each class vs every other class

<table>
<thead>
<tr>
<th>class</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>class</th>
<th>$C_1$</th>
<th>$C_2$</th>
<th>$C_3$</th>
<th>$C_4$</th>
<th>$C_5$</th>
<th>$C_6$</th>
<th>$C_7$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>b</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>c</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>d</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- Each binary classifier $C_i$ predicts the $i^{th}$ bit
  - LHS: Predictions like "1 0 1 0" cannot be "decoded"
  - RHS: Predictions like "1 0 1 1 1 1 1" are class "a" ($C_2$ made a mistake)
Hamming Distance

- Hamming distance $H$ between codewords = number of single-bit corrections needed to convert one into the other
  - $H(1000,0100) = 2$
  - $H(1111111,0000111) = 4$

- $(d-1)/2$ single-bit errors can be corrected if $d=\text{minimumum}$ Hamming distance between any pair of code-words
  - LHS: $d=2$
    - No error-correction
  - RHS: $d=4$
    - Corrects all single-bit errors