Ensemble Methods for Machine Learning
COMBINING CLASSIFIERS: ENSEMBLE APPROACHES
Common Ensemble classifiers

- Bagging/Random Forests
- “Bucket of models”
- Stacking
- Boosting
Ensemble classifiers we’ve studied so far

- Bagging
  - Build many bootstrap replicates of the data
  - Train many classifiers
  - Vote the results
Bagged trees to random forests

• Bagged decision trees:
  – Build many bootstrap replicates of the data
  – Train many tree classifiers
  – Vote the results

• Random forest:
  – Build many bootstrap replicates of the data
  – Train many tree classifiers (no pruning)
    • For each split, restrict to a random subset of the original feature set. (Usually a random subset of size $\sqrt{d}$ where there are $d$ original features.)
  – Vote the results

  – Like bagged decision trees, fast and easy to use
A “bucket of models”

- **Scenario:**
  - I have three learners
    - Logistic Regression, Naïve Bayes, Backprop
  - I’m going to embed a learning machine in some system (eg, “genius”-like music recommender)
    - Each system has a different user \(\rightarrow\) different dataset \(D\)
    - In pilot studies there’s no single best system
    - Which learner do embed to get the best result?
  - Can I combine all of them somehow and do better?
Simple learner combination: A “bucket of models”

• Input:
  – your top \( T \) favorite learners (or tunings)
    • \( L_1, \ldots, L_T \)
  – A dataset \( D \)

• Learning algorithm:
  – Use 10-CV to estimate the error of \( L_1, \ldots, L_T \)
  – Pick the best (lowest 10-CV error) learner \( L^* \)
  – Train \( L^* \) on \( D \) and return its hypothesis \( h^* \)
Pros and cons of a “bucket of models”

• Pros:
  – Simple
  – Will give results not much worse than the best of the “base learners”

• Cons:
  – What if there’s not a single best learner?

• Other approaches:
  – Vote the hypotheses (how would you weight them?)
  – Combine them some other way?
  – How about learning to combine the hypotheses?
Common Ensemble classifiers

- Bagging/Random Forests
- “Bucket of models”
- Stacking
- Boosting
Stacked learners: first attempt

• Input:
  – your top $T$ favorite learners (or tunings)
  • $L_1,\ldots,L_T$
  – A dataset $D$ containing $(x,y),\ldots$

• Learning algorithm:
  – Train $L_1,\ldots,L_T$ on $D$ to get $h_1,\ldots,h_T$
  – Create a new dataset $D'$ containing $(x',y'),\ldots$
    • $x'$ is a vector of the $T$ predictions $h_1(x),\ldots,h_T(x)$
    • $y$ is the label $y$ for $x$
  – Train YFCL on $D'$ to get $h'$--- which combines the predictions!

• To predict on a new $x$:
  – Construct $x'$ as before and predict $h'(x')$

Problem: if $L_i$ overfits the data $D$, then $h_i(x)$ could be *almost always* the same as $y$ in $D'$.

But that won’t be the case on an out-of-sample-the test example $x$.

The fix: make an $x'$ in $D'$ look more like the *out-of-sample test cases*. 
Stacked learners: the right way

- **Input:**
  - your top $T$ favorite learners (or tunings): $L_1, \ldots, L_T$
  - A dataset $D$ containing $(x, y), \ldots$

- **Learning algorithm:**
  - Train $L_1, \ldots, L_T$ on $D$ to get $h_1, \ldots, h_T$
  - Also run 10-CV and collect the CV test predictions $(x, L_i(D^*, x))$ for each example $x$ and each learner $L_i$
  - Create a new dataset $D'$ containing $(x', y'), \ldots$
    - $x'$ is the CV test predictions $(L_1(D_x, x), L_2(D_x, x), \ldots)$
    - $y$ is the label $y$ for $x$
  - Train YFCL on $D'$ to get $h'$ --- which combines the predictions!

- **To predict on a new $x$:**
  - Construct $x'$ using $h_1, \ldots, h_T$ (as before) and predict $h'(x')$
Pros and cons of stacking

• Pros:
  – Fairly simple
  – Slow, but easy to parallelize

• Cons:
  – What if there’s not a single best combination scheme?
  – E.g.: for movie recommendation sometimes L1 is best for users with many ratings and L2 is best for users with few ratings.
Multi-level stacking/blending

- Learning algorithm:
  - Train $L_1, \ldots, L_T$ on $D$ to get $h_1, \ldots, h_T$
  - Also run 10-CV and collect the CV test predictions $(x, L_i(D-x, x))$ for each example $x$ and each learner $L_i$
  - Create a new dataset $D'$ containing $(x', y'), \ldots$

  - $x'$ is the CV test predictions ($L_1(D-x, x), L_2(D-x, x), \ldots$ combined with additional features from $x$ (e.g., numRatings, userAge, ...))
  - $y$ is the label $y$ for $x$
  - Train YFCL on $D'$ to get $h'$ --- which combines the predictions!

- To predict on a new $x$:
  - Construct $x'$ using $h_1, \ldots, h_T$ (as before) and predict $h'(x')$

where the choice of classifier to rely on depends on meta-features of $x$
Comments

• Ensembles based on blending/stacking were key approaches used in the Netflix competition
  – Winning entries blended many types of classifiers
• Ensembles based on stacking are the main architecture used in Watson
  – Not all of the base classifiers/rankers are learned, however; some are hand-programmed.
Common Ensemble classifiers

- Bagging/Random Forests
- “Bucket of models”
- Stacking
- Boosting
Boosting
Valiant CACM 1984 and PAC-learning: partly inspired by Turing

<table>
<thead>
<tr>
<th>AI</th>
<th>Formal</th>
<th>Informal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Valiant (1984)</td>
<td>Turing test (1950)</td>
<td></td>
</tr>
<tr>
<td>Complexity</td>
<td>Turing machine (1936)</td>
<td></td>
</tr>
</tbody>
</table>

Question: what sort of AI questions can we formalize and study with formal methods?
"Weak" pac-learning (Kearns & Valiant 88)

(PAC learning)

<table>
<thead>
<tr>
<th>Strong Learning</th>
<th>Weak Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>∃ algorithm $A$</td>
<td>∃ algorithm $A$</td>
</tr>
<tr>
<td>$\forall c \in C$</td>
<td>$\exists \gamma &gt; 0$</td>
</tr>
<tr>
<td>$\forall D$</td>
<td>$\forall c \in C$</td>
</tr>
<tr>
<td>$\forall \epsilon &gt; 0$</td>
<td>$\forall D$</td>
</tr>
<tr>
<td>$\forall \delta &gt; 0$</td>
<td>$\forall \epsilon \geq \frac{1}{2} - \gamma$</td>
</tr>
</tbody>
</table>

$A$ produces $h \in \mathcal{H}$: $Pr[err(h) > \epsilon] \leq \delta$

say, $\epsilon = 0.49$
“Weak” PAC-learning is equivalent to “strong” PAC-learning (!) (Schapire 89)

(PAC learning)

<table>
<thead>
<tr>
<th>Strong Learning</th>
<th>Weak Learning</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists \text{ algorithm } A )</td>
<td>( \exists \text{ algorithm } A )</td>
</tr>
<tr>
<td>( \forall c \in C )</td>
<td>( \exists \gamma &gt; 0 )</td>
</tr>
<tr>
<td>( \forall D )</td>
<td>( \forall c \in C )</td>
</tr>
<tr>
<td>( \forall \epsilon &gt; 0 )</td>
<td>( \forall D )</td>
</tr>
<tr>
<td>( \forall \delta &gt; 0 )</td>
<td>( \forall \epsilon \geq \frac{1}{2} - \gamma )</td>
</tr>
<tr>
<td>( A \text{ produces } h \in \mathcal{H} : Pr[err(h) &gt; \epsilon] \leq \delta )</td>
<td>( A \text{ produces } h \in \mathcal{H} : Pr[err(h) &gt; \epsilon] \leq \delta )</td>
</tr>
</tbody>
</table>

say, \( \epsilon = 0.49 \)
“Weak” PAC-learning is equivalent to “strong” PAC-learning (!) (Schapire 89)

- The basic idea exploits the fact that you can learn a little on every distribution:
  - Learn $h1$ from $D0$ with error < 49%
  - Modify $D0$ so that $h1$ has error 50% (call this $D1$)
    - If heads wait for an example $x$ where $h1(x)=f(x)$, otherwise wait for an example where $h1(x)! = f(x)$.
  - Learn $h2$ from $D1$ with error < 49%
  - Modify $D1$ so that $h1$ and $h2$ always disagree (call this $D2$)
  - Learn $h3$ from $D2$ with error <49%.
  - Now vote $h1$, $h2$, and $h3$. This has error better than any of the “weak” hypotheses $h1$, $h2$ or $h3$.
  - Repeat this as needed to lower the error rate more....
Boosting can actually help experimentally… but…
(Drucker, Schapire, Simard)

- The basic idea exploits the fact that you can learn a little on every distribution:
  - Learn $h_1$ from $D_0$ with error < 49%
  - Modify $D_0$ so that $h_1$ has error 50% (call this $D_1$)
    - If heads **wait** for an example $x$ where $h_1(x)=f(x)$, otherwise **wait** for an example where $h_1(x)! = f(x)$.
  - Learn $h_2$ from $D_1$ with error < 49%
  - Modify $D_1$ so that $h_1$ and $h_2$ always disagree (call this $D_2$)
    - **_wait for examples where they disagree !?**
  - Learn $h_3$ from $D_2$ with error < 49%
  - Now vote $h_1$, $h_2$, and $h_3$. This has error better than any of the “**weak” hypotheses** $h_1$, $h_2$, or $h_3$.
  - Repeat this as needed to lower the error rate more….

- Very wasteful of examples
AdaBoost (Freund and Schapire)

Given: \((x_1, y_1), \ldots, (x_m, y_m)\) where \(x_i \in X, y_i \in Y = \{-1, +1\}\)
Initialize \(D_1(i) = 1/m.\)
For \(t = 1, \ldots, T:\)

- Train base learner using distribution \(D_t.\)
- Get base classifier \(h_t : X \to \mathbb{R}.\)
- Choose \(\alpha_t \in \mathbb{R}.\)
- Update:

\[
D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}
\]

where \(Z_t\) is a normalization factor (chosen so that \(D_{t+1}\) will be a distribution).

Output the final classifier:

\[
H(x) = \text{sign} \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right).
\]

Figure 1: The boosting algorithm AdaBoost.
Given: \((x_1, y_1), \ldots, (x_m, y_m)\) where \(x_i \in X, y_i \in Y = \{-1, +1\}\)
Initialize \(D_1(i) = 1/m\).
For \(t = 1, \ldots, T\):

- Train base learner using distribution \(D_t\).
- Get base classifier \(h_t : X \to \mathbb{R}\).
- Choose \(\alpha_t \in \mathbb{R}\).
- Update:
\[
D_{t+1}(i) = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}
\]
where \(Z_t\) is a normalization factor (chosen so that \(D_{t+1}\) will be a distribution).

Output the final classifier:
\[
H(x) = \text{sign} \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right).
\]

Figure 1: The boosting algorithm AdaBoost.
AdaBoost: Adaptive Boosting (Freund & Schapire, 1995)

Given: \((x_1, y_1), \ldots, (x_m, y_m)\) where \(x_i \in X, y_i \in Y = \{-1, +1\}\)

Initialize \(D_1(i) = 1/m\).

For \(t = 1, \ldots, T\):

- Train weak learner using distribution \(D_t\).
- Get weak hypothesis \(h_t : X \rightarrow \{-1, +1\}\) with error
  \[
  \epsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i].
  \]
- Choose \(\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)\).
- Update:
  \[
  D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} 
  e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\
  e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i
  \end{cases}
  = \frac{D_t(i) \exp(-\alpha_t y_t h_t(x_i))}{Z_t}
  \]
  where \(Z_t\) is a normalization factor (chosen so that \(D_{t+1}\) will be a distribution).

Output the final hypothesis:
\[
H(x) = \text{sign} \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right).
\]

Figure 1: The boosting algorithm AdaBoost.
Given: \((x_1, y_1), \ldots, (x_m, y_m)\) where \(x_i \in X, y_i \in Y = \{-1, +1\}\)

Initialize \(D_1(i) = 1/m\).

For \(t = 1, \ldots, T\):

- Train weak learner using distribution \(D_t\).
- Get weak hypothesis \(h_t : X \to \{-1, +1\}\) with error
  \[
  \epsilon_t = \Pr_{i \sim D_t} [h_t(x_i) \neq y_i].
  \]
- Choose \(\alpha_t = \frac{1}{2} \ln \left( \frac{1 - \epsilon_t}{\epsilon_t} \right)\).
- Update:
  \[
  D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} 
  e^{-\alpha_t} & \text{if } h_t(x_i) = y_i \\
  e^{\alpha_t} & \text{if } h_t(x_i) \neq y_i
  \end{cases}
  \]
  \[
  = \frac{D_t(i) \exp(-\alpha_t y_i h_t(x_i))}{Z_t}
  \]
  where \(Z_t\) is a normalization factor (chosen so that \(D_{t+1}\) will be a distribution).

Output the final hypothesis:

\[
H(x) = \text{sign} \left( \sum_{t=1}^{T} \alpha_t h_t(x) \right).
\]
Boosting: A toy example

Thanks, Rob Schapire
Boosting: A toy example

Thanks, Rob Schapire

Round 1

\[ h_1 \]

\[ \varepsilon_1 = 0.30 \]
\[ \alpha_1 = 0.42 \]
Boosting: A toy example

Round 2

\[ \varepsilon_2 = 0.21 \]
\[ \alpha_2 = 0.65 \]
Boosting: A toy example

Round 3

\( h_3 \)

\( \varepsilon_3 = 0.14 \)

\( \alpha_3 = 0.92 \)
Boosting: A toy example

Final Classifier

\[ H_{\text{final}} = \text{sign} \left( \begin{array}{c} 0.42 \\ +0.65 \\ +0.92 \end{array} \right) \]
Boosting improved decision trees…

![Graph showing boosting stumps vs. C4.5](image)

- 1950 - T
- 1984 - V
- 1988 - KV
- 1989 - S
- 1993 - DSS
- 1995 - FS
Boosting: Analysis

Theorem: if the error at round $t$ of the base classifier is $\varepsilon_t = \frac{1}{2} - \gamma_t$, then the \textit{training error} of the boosted classifier is bounded by

$$\prod_t \left[ 2 \sqrt{\varepsilon_t(1-\varepsilon_t)} \right] = \prod_t \sqrt{1 - 4\gamma_t^2} \leq \exp \left( -2 \sum_t \gamma_t^2 \right)$$

The algorithm doesn’t need to know any bound on $\gamma_t$ in advance though -- it \textit{adapts} to the actual sequence of errors.
BOOSTING AS OPTIMIZATION
Even boosting *single features* worked well...
Coordinate descent optimization to minimize $f(w)$

- For $t=1,\ldots,T$ or till convergence:
  - For $i=1,\ldots,N$ where $w=<w_1,\ldots,w_N>$
    - Pick $w^*$ to minimize
      \[ f(<w_1,\ldots,w_{i-1},w^*,w_{i+1},\ldots,w_N>) \]
    - Set $w_i = w^*$
Boosting as optimization using coordinate descent

With a small number of possible $h$’s, you can think of boosting as finding a linear combination of these:

$$H(x) = \text{sign} \left( \sum_i w_i h_i(x) \right)$$

So boosting is sort of like stacking:

$$\mathbf{h}(x) \equiv \langle h_1(x), \ldots, h_N(x) \rangle \quad \text{(stacked) instance vector}$$

$$\mathbf{w} \equiv \langle \alpha_1, \ldots, \alpha_N \rangle \quad \text{weight vector}$$

Boosting uses coordinate descent to minimize an upper bound on error rate:

$$\sum_{t=i} \exp \left( y_i \sum_i w_i h_i(x) \right)$$
Boosting and optimization


Compared using AdaBoost to set feature weights vs direct optimization of feature weights to minimize log-likelihood, squared error, …
BOOSTING AS MARGIN LEARNING
Boosting didn’t seem to overfit…(!)
...because it turned out to be increasing the *margin* of the classifier
Boosting movie
Some background facts

**Coordinate descent** optimization to minimize $f(w)$

- For $t=1,\ldots,T$ or till convergence:
  - For $i=1,\ldots,N$ where $w=<w_1,\ldots,w_N>$
    - Pick $w^*$ to minimize
      \[ f(<w_1,\ldots,w_{i-1},w^*,w_{i+1},\ldots,w_N>) \]
    - Set $w_i = w^*$

\[
\| w \|_k = \sqrt[k]{w_1^k + \ldots + w_T^k} \quad \| w \|_2 = \sqrt{w_1^2 + \ldots + w_T^2} \quad \| w \|_1 = w_1 + \ldots + w_T \quad \| w \|_\infty = \max(w_1,\ldots,w_T)
\]
Boosting is closely related to margin classifiers like SVM, voted perceptron, … (!)

\[
\mathbf{h}(x) \equiv \langle h_1(x), \ldots, h_T(x) \rangle \quad \text{(stacked) instance vector}
\]
\[
\mathbf{w} \equiv \langle \alpha_1, \ldots, \alpha_N \rangle \quad \text{weight vector}
\]

Boosting:

\[
\max_w \min_x \frac{\mathbf{w} \cdot \mathbf{h}(x) \cdot y}{\|\mathbf{w}\|_1 \cdot \|\mathbf{h}(x)\|_\infty}
\]

optimized by coordinate descent

here \(\|\mathbf{w}\|_1 = \sum_t \alpha_t\) and \(\|\mathbf{h}(x)\|_\infty = \max_t h_t(x)\)

The “coordinates” are being extended by one in each round of boosting --- usually, unless you happen to generate the same tree twice
Boosting is closely related to margin classifiers like SVM, voted perceptron, … (!)

\[ h(x) = \langle h_1(x), ..., h_T(x) \rangle \text{ (stacked) instance vector} \]
\[ w = \langle \alpha_1, ..., \alpha_N \rangle \text{ weight vector} \]

**Boosting:**
\[
\max_w \min_x \frac{w \cdot h(x) \cdot y}{\| w \|_1 \cdot \| h(x) \|_\infty}
\]
optimized by coordinate descent

here \( \| w \|_1 = \sum_t \alpha_t \) and \( \| h(x) \|_\infty = \max_t h_t(x) \)

**Linear SVMs:**
\[
\max_w \min_x \frac{w \cdot h(x) \cdot y}{\| w \|_2 \cdot \| h(x) \|_2}
\]
optimized by QP,…

where \( \| w \|_2 = \sqrt{\sum_t \alpha_t^2} \) and \( \| h(x) \|_2 = \sqrt{h_t(x)^2} \)
BOOSTING VS BAGGING
Boosting vs Bagging

- Both build weighted combinations of classifiers, each classifier being learned on a sample of the original data
- Boosting reweights the distribution in each iteration
- Bagging doesn’t
Boosting vs Bagging

- Boosting finds a linear combination of weak hypotheses
  - Theory says it reduces *bias*
  - In practice is also reduces variance, especially in later iterations
Boosting vs Bagging
WRAPUP ON BOOSTING
Boosting in the real world

- William’s wrap up:
  - Boosting is not discussed much in the ML research community any more
    - It’s much too well understood
  - It’s really useful in practice as a meta-learning method
    - Eg, boosted Naïve Bayes usually beats Naïve Bayes
  - Boosted decision trees are
    - almost always competitive with respect to accuracy
    - very robust against rescaling numeric features, extra features, non-linearities, …
    - somewhat slower to learn and use than many linear classifiers
    - But getting probabilities out of them is a little less reliable.