Ensembles

• An ensemble is a set of classifiers whose combined results give the final decision.
MODEL* ENSEMBLES

- Basic Idea
  - Instead of learning one model
  - Learn several and combine them

- Often this improves accuracy by a lot

- Many Methods
  - Bagging
  - Boosting
  - Stacking

*A model is the learned decision rule. It can be as simple as a hyperplane in n-space (ie. a line in 2D or plane in 3D) or in the form of a decision tree or other modern classifier.
Bagging

• Generate bootstrap replicates of the training set by sampling with replacement

• Learn one model on each replicate

• Combine by uniform voting
Boosting

• Maintain a vector of weights for samples
• Initialize with uniform weights
• Loop
  – Apply learner to weighted samples
  – Increase weights of misclassified ones
• Combine models by weighted voting
Idea of Boosting

\[
h = h_1, h_2, h_3, h_4
\]
1. Set all $E$ weights to 1, and learn $H_1$.
2. Repeat $m$ times: increase the weights of misclassified $E$s, and learn $H_2, \ldots H_m$.
3. $H_1 \ldots H_m$ have “weighted majority” vote when classifying each test:
   $\text{Weight}(H) =$ accuracy of $H$ on the training data.
ADABoost

• ADABoost boosts the accuracy of the original learning algorithm.

• If the original learning algorithm does slightly better than 50% accuracy, ADABoost with a large enough number of classifiers is guaranteed to classify the training data perfectly.
ADABoost Weight Updating
(from Fig 18.34 text)

/* First find the sum of the weights of the misclassified samples */
for j = 1 to N do /* go through training samples */
    if h[m](x_j) <> y_j then error <- error + w_j

/* Now use the ratio of error to 1-error to change the weights of the correctly classified samples */
for j = 1 to N do
    if h[m](x_j) = y_j then w[j] <- w[j] * error/(1-error)
Example

- Start with 4 samples of equal weight .25.
- Suppose 1 is misclassified. So error = .25.
- The ratio comes out .25/.75 = .33
- The correctly classified samples get weight of .25*.33 = .0825

.2500
.0825
.0825
.0825

What’s wrong? What should we do?

We want them to add up to 1, not .4975.

Answer: To normalize, divide each one by their sum (.4975).
Sample Application: Insect Recognition

Using circular regions of interest selected by an interest operator, train a classifier to recognize the different classes of insects.
Boosting Comparison

- **ADTree classifier only** *(alternating decision tree)*

- Correctly Classified Instances: 268 (70.1571 %)
- Incorrectly Classified Instances: 114 (29.8429 %)
- Mean absolute error: 0.3855
- Relative absolute error: 77.2229 %

<table>
<thead>
<tr>
<th>Classified as -&gt;</th>
<th>Hesperperla</th>
<th>Doroneuria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Hesperperlas</td>
<td>167</td>
<td>28</td>
</tr>
<tr>
<td>Real Doroneuria</td>
<td>51</td>
<td>136</td>
</tr>
</tbody>
</table>
Boosting Comparison

AdaboostM1 with ADTree classifier

- Correctly Classified Instances: 303, 79.3194%
- Incorrectly Classified Instances: 79, 20.6806%
- Mean absolute error: 0.2277
- Relative absolute error: 45.6144%

<table>
<thead>
<tr>
<th>Classified as -&gt;</th>
<th>Hesperperla</th>
<th>Doroneuria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Hesperperlas</td>
<td>167</td>
<td>28</td>
</tr>
<tr>
<td>Real Doroneuria</td>
<td>51</td>
<td>136</td>
</tr>
</tbody>
</table>
Boosting Comparison

- **RepTree classifier only (reduced error pruning)**
- Correctly Classified Instances: 294, 75.3846 %
- Incorrectly Classified Instances: 96, 24.6154 %
- Mean absolute error: 0.3012
- Relative absolute error: 60.606 %

<table>
<thead>
<tr>
<th>Classified as -&gt;</th>
<th>Hesperperla</th>
<th>Doroneuria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Hesperperlas</td>
<td>169</td>
<td>41</td>
</tr>
<tr>
<td>Real Doroneuria</td>
<td>55</td>
<td>125</td>
</tr>
</tbody>
</table>
Boosting Comparison

AdaboostM1 with RepTree classifier

- Correctly Classified Instances: 324 (83.0769 %)
- Incorrectly Classified Instances: 66 (16.9231 %)
- Mean absolute error: 0.1978
- Relative absolute error: 39.7848 %

<table>
<thead>
<tr>
<th>Classified as -&gt;</th>
<th>Hesperperla</th>
<th>Doroneuria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real Hesperperlas</td>
<td>180</td>
<td>30</td>
</tr>
<tr>
<td>Real Doroneuria</td>
<td>36</td>
<td>144</td>
</tr>
</tbody>
</table>
References


Stacking

- Apply multiple base learners
  (e.g.: decision trees, naive Bayes, neural nets)
- Meta-learner: Inputs = Base learner predictions
- Training by leave-one-out cross-validation:
  Meta-L. inputs = Predictions on left-out examples
Random Forests

- **Tree bagging** creates decision trees using the bagging technique. The whole set of such trees (each trained on a random sample) is called a decision forest. The final prediction takes the average (or majority vote).

-Random forests(5,11),(994,990) differ in that they use a modified tree learning algorithm that selects, at each candidate split, a random subset of the features.
Back to Stone Flies

Random forest of 10 trees, each constructed while considering 7 random features. Out of bag error: 0.2487. Time taken to build model: 0.14 seconds

Correctly Classified Instances 292 76.4398 % (81.4 with AdaBoost)
Incorrectly Classified Instances 90 23.5602 %
Kappa statistic 0.5272
Mean absolute error 0.344
Root mean squared error 0.4069
Relative absolute error 68.9062 %
Root relative squared error 81.2679 %
Total Number of Instances 382

<table>
<thead>
<tr>
<th>TP Rate</th>
<th>FP Rate</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>ROC Area</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.69</td>
<td>0.164</td>
<td>0.801</td>
<td>0.69</td>
<td>0.741</td>
<td>0.848</td>
<td>cal</td>
</tr>
<tr>
<td>0.836</td>
<td>0.31</td>
<td>0.738</td>
<td>0.836</td>
<td>0.784</td>
<td>0.848</td>
<td>dor</td>
</tr>
</tbody>
</table>

WAvg. 0.764 0.239 0.769 0.764 0.763 0.848

a   b  <-- classified as
129 58 | a = cal
32 163 | b = dor
More on Learning

• Neural Nets
• Support Vectors Machines
• Unsupervised Learning (Clustering)
  – K-Means
  – Expectation-Maximization
Neural Net Learning

• Motivated by studies of the brain.

• A network of “artificial neurons” that learns a function.

• Doesn’t have clear decision rules like decision trees, but highly successful in many different applications. (e.g. face detection)

• We use them frequently in our research.

$10^{11}$ neurons of $>20$ types, $10^{14}$ synapses, 1ms–10ms cycle time

Signals are noisy “spike trains” of electrical potential
McCulloch–Pitts “unit”

Output is a “squashed” linear function of the inputs:

\[ a_i \leftarrow g(in_i) = g(\sum_j W_{j,i} a_j) \]

A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do
(a) is a step function or threshold function

(b) is a sigmoid function $1/(1 + e^{-x})$

Changing the bias weight $W_{0,i}$ moves the threshold location
Simple Feed-Forward Perceptrons

\[ \text{in} = (\sum W_j x_j) + \theta \]
\[ \text{out} = g(\text{in}) \]

\( g \) is the activation function. It can be a step function:
\[ g(x) = 1 \text{ if } x \geq 0 \text{ and } 0 \text{ (or -1) else.} \]

It can be a sigmoid function:
\[ g(x) = \frac{1}{1+\exp(-x)} \]

The sigmoid function is differentiable and can be used in a gradient descent algorithm to update the weights.
Gradient Descent

takes steps proportional to the negative of the gradient of a function to find its local minimum

• Let $X$ be the inputs, $y$ the class, $W$ the weights
  
• $\text{in} = \sum W_j x_j$
  
• $\text{Err} = y - g(\text{in})$
  
• $E = \frac{1}{2} \text{Err}^2$ is the squared error to minimize
  
• $\frac{\partial E}{\partial W_j} = \text{Err} \cdot \frac{\partial \text{Err}}{\partial W_j} = \text{Err} \cdot \frac{\partial}{\partial W_j}(g(\text{in}))(\text{-1})$
  
• = $-\text{Err} \cdot g'(\text{in}) \cdot x_j$
  
• The update is $W_j \leftarrow W_j + \alpha \cdot \text{Err} \cdot g'(\text{in}) \cdot x_j$
  
• $\alpha$ is called the learning rate.
Simple Feed-Forward Perceptrons

repeat
  for each e in examples do
    in = (∑ W_j x_j) + θ
    Err = y[e] – g[in]
    W_j = W_j + α Err g'(in) x_j[e]
  until done

Examples: A=[(.5,1.5),+1], B=[(-.5,.5),-1], C=[(.5,.5),+1]
Initialization: W_1 = 1, W_2 = 2, θ = -2

Note1: when g is a step function, the g'(in) is removed.
Note2: later in back propagation, Err * g'(in) will be called Δ
Note3: We’ll let g(x) = 1 if x >=0 else -1
Graphically

Examples: $A = [(0.5, 1.5), +1]$, $B = [(-0.5, 0.5), -1]$, $C = [(0.5, 0.5), +1]$

Initialization: $W_1 = 1$, $W_2 = 2$, $\theta = -2$

Boundary is $W_1x_1 + W_2x_2 + \theta = 0$
Examples:
A=[(.5,1.5),+1],
B=[(-.5,.5),-1],
C=[(.5,.5),+1]
Initialization: \( W_1 = 1, \ W_2 = 2, \ \theta = -2 \)

\[
\begin{align*}
\text{A} = [(0.5,1.5),+1] \\
\text{in} &= 0.5(1) + (1.5)(2) - 2 = 1.5 \\
g(\text{in}) &= 1; \ Err = 0; \text{NO CHANGE}
\end{align*}
\]

\[
\begin{align*}
\text{B} = [(-0.5,0.5),-1] \\
\text{In} &= (-0.5)(1) + (0.5)(2) - 2 = -1.5 \\
g(\text{in}) &= -1; \ Err = 0; \text{NO CHANGE}
\end{align*}
\]

\[
\begin{align*}
\text{C} = [(0.5,0.5),+1] \\
\text{in} &= 0.5(1) + (0.5)(2) - 2 = -0.5 \\
g(\text{in}) &= -1; \ Err = 1-(-1)=2
\end{align*}
\]

Let \( \alpha = .5 \)

\[
\begin{align*}
\text{W1} &\gets \text{W1} + 0.5(2)(.5) \text{ leaving out } g' \\
&\gets 1 + 1(0.5) = 1.5 \\
\text{W2} &\gets \text{W2} + 0.5(2)(.5) \\
&\gets 2 + 1(0.5) = 2.5 \\
\theta &\gets \theta + 0.5(+1 - (-1)) \\
&\gets -2 + 0.5(2) = -1
\end{align*}
\]

repeat
for each e in examples do
in = (\( \sum \ W_j x_j \) + \( \theta \))
Err = \( y[e] - g[\text{in}] \)
\( W_j = W_j + \alpha \ Err \ g'(\text{in}) \ x_j[e] \)
until done
Graphically

Examples: A = [(0.5, 1.5), +1], B = [(-0.5, 0.5), -1], C = [(0.5, 0.5), +1]

Initialization: \( W_1 = 1 \), \( W_2 = 2 \), \( \theta = -2 \)

Boundary is \( W_1 x_1 + W_2 x_2 + \theta = 0 \)

approximately correct boundary
Back Propagation

• Simple single layer networks with feed forward learning were not powerful enough.
• Could only produce simple linear classifiers.
• More powerful networks have multiple hidden layers.
• The learning algorithm is called back propagation, because it computes the error at the end and propagates it back through the weights of the network to the beginning.
The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

**function** BACK-PROP-LEARNING(examples, network)
**returns** a neural network

**inputs:**
- `examples`, a set of examples, each with input vector `x` and output vector `y`.
- `network`, a multilayer network with `L` layers, weights `W_{j,i}`, activation function `g`

**local variables:** `Δ`, a vector of errors, indexed by network node

**for each** weight `w_{i,j}` in `network` **do**

```
  w_{i,j} ← a small random number
```

**repeat**

**for each** example `(x,y)` in `examples` **do**

```
/* Propagate the inputs forward to compute the outputs. */
  for each node `i` in the input layer **do**  // Simply copy the input values.
    a_i ← x_i
  for `l` = 2 to `L` **do**  // Feed the values forward.
    for each node `j` in layer `l` **do**
      \[ in_j = \sum_i w_{i,j} a_i \]
      \[ a_j ← g(in_j) \]
  for each node `j` in the output layer **do**  // Compute the error at the output.
    \[ Δ[j] ← g'(in_j) \times (y_j - a_j) \]

/* Propagate the deltas backward from output layer to input layer */
  for `l` = `L` - 1 to 1 **do**
    for each node `i` in layer `l` **do**
      \[ Δ[i] ← g'(in_i) \sum_j w_{i,j} Δ[j] \]  // “Blame” a node as much as its weight

/* Update every weight in network using deltas */
  for each weight `w_{i,j}` in `network` **do**
    \[ w_{i,j} ← w_{i,j} + \alpha \times a_i \times Δ[j] \]  // Adjust the weights.
```

**until** some stopping criterion is satisfied

**return** `network`
The backpropagation algorithm

The following is the backpropagation algorithm for learning in multilayer networks.

function BACK-PROP-LEARNING(examples, network)
returns a neural network

inputs:
examples, a set of examples, each with input vector \( \mathbf{x} \) and output vector \( \mathbf{y} \).

network, a multilayer network with \( L \) layers, weights \( W_{j,i} \), activation function \( g \)

local variables: \( \Delta \), a vector of errors, indexed by network node

for each weight \( w_{i,j} \) in network do
\[ w_{i,j} \leftarrow \text{a small random number} \]
repeat
  for each example \((x, y)\) in examples do
    /* Propagate the inputs forward to compute the outputs. */
    for each node \(i\) in the input layer do
      \(a_i \leftarrow x_i\)  // Simply copy the input values.
    for \(l = 2\) to \(L\) do
      for each node \(j\) in layer \(l\) do
        \(in_j \leftarrow \sum_i w_{i,j} a_i\)
        \(a_j \leftarrow g(in_j)\)  // Feed the values forward.
Backward Propagation 1

\[
\text{for each node } j \text{ in the output layer do} \quad \Delta[j] \leftarrow g'(in_j) \times (y_j - a_j) \\
\]

- Node \( n_f \) is the only node in our output layer.
- Compute the error at that node and multiply by the derivative of the weighted input sum to get the change \( \Delta_f \).

layer 1           2           3=L

\[ x_1 \quad w_{11} \quad n_1 \quad w_{1f} \quad n_f \]
\[ x_2 \quad w_{21} \quad n_1 \quad w_{1f} \quad n_f \]
\[ x_3 \quad w_{31} \quad n_2 \quad w_{2f} \quad n_f \]

\( \Delta_f \)
Backward Propagation 2

/* Propagate the deltas backward from output layer to input layer */
for \( l = L - 1 \) to 1 do
  for each node \( i \) in layer \( l \) do
    \[ \Delta[i] \leftarrow g'(in_i) \sum_j w_{i,j} \Delta[j] \]  // “Blame” a node as much as its weight

- At each of the other layers, the deltas use
  - the derivative of its input sum
  - the sum of its output weights
  - the delta computed for the output error
Backward Propagation 3

/* Update every weight in network using deltas. */
for each weight $w_{i,j}$ in network do
    $w_{i,j} \leftarrow w_{i,j} + \alpha \times a_i \times \Delta[j]$       // Adjust the weights.

Now that all the deltas are defined, the weight updates just use them.
Back Propagation Summary

• Compute delta values for the output units using observed errors.

• Starting at the output-1 layer
  – repeat
    • propagate delta values back to previous layer
    • update weights between the two layers
  – till done with all layers

• This is done for all examples and multiple epochs, till convergence or enough iterations.
Time taken to build model: **16.2 seconds**

Correctly Classified Instances 307 80.3665 % (did not boost)
Incorrectly Classified Instances 75 19.6335 %
Kappa statistic 0.6056
Mean absolute error 0.1982
Root mean squared error 0.41
Relative absolute error 39.7113 %
Root relative squared error 81.9006 %
Total Number of Instances 382

<table>
<thead>
<tr>
<th>TP Rate</th>
<th>FP Rate</th>
<th>Precision</th>
<th>Recall</th>
<th>F-Measure</th>
<th>ROC Area</th>
<th>Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.706</td>
<td>0.103</td>
<td>0.868</td>
<td>0.706</td>
<td>0.779</td>
<td>0.872</td>
<td>cal</td>
</tr>
<tr>
<td>0.897</td>
<td>0.294</td>
<td>0.761</td>
<td>0.897</td>
<td>0.824</td>
<td>0.872</td>
<td>dor</td>
</tr>
<tr>
<td>W Avg.</td>
<td>0.804</td>
<td>0.2</td>
<td>0.814</td>
<td>0.804</td>
<td>0.802</td>
<td>0.872</td>
</tr>
</tbody>
</table>

=== Confusion Matrix ===

```
a b <-- classified as
132 55 | a = cal
20 175 | b = dor
```
Handwritten digit recognition

\[
\begin{array}{cccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

3-nearest-neighbor = 2.4% error
400–300–10 unit MLP = 1.6% error
LeNet: 768–192–30–10 unit MLP = 0.9% error

Current best (kernel machines, vision algorithms) \(\approx\) 0.6% error