Machine Learning – Classifiers and Boosting

Reading

Ch 18.6-18.12, 20.1-20.3.2
Outline

• Different types of learning problems

• Different types of learning algorithms

• Supervised learning
  – Decision trees
  – Naïve Bayes
  – Perceptrons, Multi-layer Neural Networks
  – Boosting

• Applications: learning to detect faces in images
You will be expected to know

- Classifiers:
  - Decision trees
  - K-nearest neighbors
  - Naïve Bayes
  - Perceptrons, Support vector Machines (SVMs), Neural Networks

- Decision Boundaries for various classifiers
  - What can they represent conveniently? What not?
Inductive learning

• Let $\mathbf{x}$ represent the input vector of attributes
  – $x_j$ is the $j$th component of the vector $\mathbf{x}$
  – $x_j$ is the value of the $j$th attribute, $j = 1, \ldots, d$

• Let $f(\mathbf{x})$ represent the value of the target variable for $\mathbf{x}$
  – The implicit mapping from $\mathbf{x}$ to $f(\mathbf{x})$ is unknown to us
  – We just have training data pairs, $D = \{\mathbf{x}, f(\mathbf{x})\}$ available

• We want to learn a mapping from $\mathbf{x}$ to $f$, i.e.,
  $h(\mathbf{x}; \theta)$ is “close” to $f(\mathbf{x})$ for all training data points $\mathbf{x}$

  $\theta$ are the parameters of our predictor $h(\mathbf{..})$

• Examples:
  – $h(\mathbf{x}; \theta) = \text{sign}(w_1x_1 + w_2x_2 + w_3)$
  – $h_k(\mathbf{x}) = (x_1 \text{ OR } x_2) \text{ AND } (x_3 \text{ OR } \neg(x_4))$
### Training Data for Supervised Learning

<table>
<thead>
<tr>
<th>Example</th>
<th>Alt</th>
<th>Bar</th>
<th>Fri</th>
<th>Hun</th>
<th>Pat</th>
<th>Price</th>
<th>Rain</th>
<th>Res</th>
<th>Type</th>
<th>Est</th>
<th>Wait</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X_1$</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>French</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>$X_2$</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>30–60</td>
<td>F</td>
</tr>
<tr>
<td>$X_3$</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Some</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Burger</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>$X_4$</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>10–30</td>
<td>T</td>
</tr>
<tr>
<td>$X_5$</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Full</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>French</td>
<td>&gt;60</td>
<td>F</td>
</tr>
<tr>
<td>$X_6$</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$</td>
<td>T</td>
<td>T</td>
<td>Italian</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>$X_7$</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>None</td>
<td>$</td>
<td>T</td>
<td>F</td>
<td>Burger</td>
<td>0–10</td>
<td>F</td>
</tr>
<tr>
<td>$X_8$</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$</td>
<td>T</td>
<td>T</td>
<td>Thai</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>$X_9$</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>F</td>
<td>Full</td>
<td>$</td>
<td>T</td>
<td>F</td>
<td>Burger</td>
<td>&gt;60</td>
<td>F</td>
</tr>
<tr>
<td>$X_{10}$</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>Italian</td>
<td>10–30</td>
<td>F</td>
</tr>
<tr>
<td>$X_{11}$</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>None</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>0–10</td>
<td>F</td>
</tr>
<tr>
<td>$X_{12}$</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Burger</td>
<td>30–60</td>
<td>T</td>
</tr>
</tbody>
</table>
True Tree (left) versus Learned Tree (right)
Classification Problem with Overlap
Decision Boundaries

Decision Region 1

Decision Boundary

Decision Region 2
Classification in Euclidean Space

• A classifier is a partition of the space $\mathbf{x}$ into disjoint decision regions
  – Each region has a label attached
  – Regions with the same label need not be contiguous
  – For a new test point, find what decision region it is in, and predict the corresponding label

• Decision boundaries = boundaries between decision regions
  – The “dual representation” of decision regions

• We can characterize a classifier by the equations for its decision boundaries

• Learning a classifier $\Leftrightarrow$ searching for the decision boundaries that optimize our objective function
Example: Decision Trees

- When applied to real-valued attributes, decision trees produce “axis-parallel” linear decision boundaries.

- Each internal node is a binary threshold of the form $x_j > t$?

  converts each real-valued feature into a binary one

  requires evaluation of $N-1$ possible threshold locations for $N$ data points, for each real-valued attribute, for each internal node.
Decision Tree Example

Decision tree with a split at Income > t1.
Decision Tree Example

Income > t1

Debt > t2

??

Income

Debt

t2

t1

??

Income > t1
Decision Tree Example

- Income > t1
- Debt > t2
- Income > t3
Decision Tree Example

Note: tree boundaries are linear and axis-parallel
A Simple Classifier: Minimum Distance Classifier

- **Training**
  - Separate training vectors by class
  - Compute the mean for each class, \( \mu_k \), \( k = 1, \ldots, m \)

- **Prediction**
  - Compute the closest mean to a test vector \( x' \) (using Euclidean distance)
  - Predict the corresponding class

- In the 2-class case, the decision boundary is defined by the locus of the hyperplane that is halfway between the 2 means and is orthogonal to the line connecting them

- This is a very simple-minded classifier – easy to think of cases where it will not work very well
Minimum Distance Classifier

![Minimum Distance Classifier Diagram](image-url)
Another Example: Nearest Neighbor Classifier

• The nearest-neighbor classifier
  – Given a test point \( x' \), compute the distance between \( x' \) and each input data point
  – Find the closest neighbor in the training data
  – Assign \( x' \) the class label of this neighbor
  – (sort of generalizes minimum distance classifier to exemplars)

• If Euclidean distance is used as the distance measure (the most common choice), the nearest neighbor classifier results in piecewise linear decision boundaries

• Many extensions
  – e.g., kNN, vote based on k-nearest neighbors
  – k can be chosen by cross-validation
Local Decision Boundaries

Boundary? Points that are equidistant between points of class 1 and 2
Note: locally the boundary is linear
Finding the Decision Boundaries

Feature 1 vs. Feature 2
Finding the Decision Boundaries

Feature 1

Feature 2
Finding the Decision Boundaries
Overall Boundary = Piecewise Linear

Decision Region for Class 1

Decision Region for Class 2

Feature 1

Feature 2
Nearest-Neighbor Boundaries on this data set?
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing $k$ "simplifies" decision boundary
  - Majority voting means less emphasis on individual points

$K = 1$  
$K = 3$
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing $k$ “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

$K = 5$  
$K = 7$
kNN Decision Boundary

- piecewise linear decision boundary
- Increasing $k$ “simplifies” decision boundary
  - Majority voting means less emphasis on individual points

\[ K = 25 \]

- True (“best”) decision boundary
  - In this case is linear
  - Compared to kNN: not bad!
The kNN Classifier

- The kNN classifier often works very well.

- Easy to implement.

- Easy choice if characteristics of your problem are unknown.

- Can be sensitive to the choice of distance metric.
  - Often normalize feature axis values, e.g., z-score or [0, 1]
  - Categorical feature axes are difficult, e.g., Color as Red/Blue/Green

- Can encounter problems with sparse training data.

- Can encounter problems in very high dimensional spaces.
  - Most points are corners.
  - Most points are at the edge of the space.
  - Most points are neighbors of most other points.
**Linear Classifiers**

- Linear classifier ⇔ single linear decision boundary (for 2-class case)

- We can always represent a linear decision boundary by a linear equation:
  \[ w_1 x_1 + w_2 x_2 + \ldots + w_d x_d = \sum w_j x_j = w^t x = 0 \]

- In d dimensions, this defines a (d-1) dimensional hyperplane
  - d=3, we get a plane; d=2, we get a line

- For prediction we simply see if \( \sum w_j x_j > 0 \)

- The \( w_i \) are the weights (parameters)
  - Learning consists of searching in the d-dimensional weight space for the set of weights (the linear boundary) that minimizes an error measure
  - A threshold can be introduced by a “dummy” feature that is always one; its weight corresponds to (the negative of) the threshold

- Note that a minimum distance classifier is a special (restricted) case of a linear classifier
A Possible Decision Boundary
Another Possible Decision Boundary
Minimum Error
Decision Boundary

FEATURE 1

FEATURE 2
The Perceptron Classifier (pages 729-731 in text)

• The perceptron classifier is just another name for a linear classifier for 2-class data, i.e.,

\[
\text{output}(x) = \text{sign}( \sum w_j x_j )
\]

• Loosely motivated by a simple model of how neurons fire

• For mathematical convenience, class labels are +1 for one class and -1 for the other

• Two major types of algorithms for training perceptrons
  – Objective function = classification accuracy ("error correcting")
  – Objective function = squared error (use gradient descent)

  – Gradient descent is generally faster and more efficient – but there is a problem! No gradient!
The Perceptron Classifier (pages 729-731 in text)

- **Input Attributes (Features)**
- **Weights** For Input Attributes
- **Bias or Threshold**
- **Transfer Function**
- **Output**
Two different types of perceptron output

x-axis below is $f(x) = f = \text{weighted sum of inputs}$
y-axis is the perceptron output

Thresholded output (step function), takes values +1 or -1

Sigmoid output, takes real values between -1 and +1

The sigmoid is in effect an approximation to the threshold function above, but has a gradient that we can use for learning

- Sigmoid function is defined as
  \[
  \sigma[f] = \left[ \frac{2}{1 + \exp[-f]} \right] - 1
  \]

- Derivative of sigmoid
  \[
  \frac{\partial \sigma}{\partial f}[f] = 0.5 \ast (\sigma[f]+1) \ast (1-\sigma[f])
  \]
Squared Error for Perceptron with Sigmoidal Output

- Squared error: $E[w] = \sum_i \left[ \sigma(f[x(i)]) - y(i) \right]^2$

  where $x(i)$ is the $i$th input vector in the training data, $i=1,..N$
  $y(i)$ is the $i$th target value (-1 or 1)

  $f[x(i)] = \sum w_j x_j$ is the weighted sum of inputs
  $\sigma(f[x(i)])$ is the sigmoid of the weighted sum

- Note that everything is fixed (once we have the training data) except for the weights $w$

- So we want to minimize $E[w]$ as a function of $w$
Gradient Descent Learning of Weights

Gradient Descent Rule:

\[ \mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \eta \Delta (E[\mathbf{w}]) \]

where

\[ \Delta (E[\mathbf{w}]) \] is the gradient of the error function \( E \) wrt weights, and
\( \eta \) is the learning rate (small, positive)

Notes:

1. This moves us downhill in direction \( \Delta (E[\mathbf{w}]) \) (steepest downhill)
2. How far we go is determined by the value of \( \eta \)
Gradient Descent Update Equation

- From basic calculus, for perceptron with sigmoid, and squared error objective function, gradient for a single input $x(i)$ is
  $$\Delta (E[w]) = -(y(i) - \sigma[f(i)]) \frac{\partial \sigma[f(i)]}{\partial x_j(i)}$$

- Gradient descent weight update rule:
  $$w_j = w_j + \eta (y(i) - \sigma[f(i)]) \frac{\partial \sigma[f(i)]}{\partial x_j(i)}$$

  - can rewrite as:
  $$w_j = w_j + \eta \times \text{error} \times c \times x_j(i)$$
Pseudo-code for Perceptron Training

Initialize each $w_j$ (e.g., randomly)

While (termination condition not satisfied)
  for $i = 1: N$  % loop over data points (an iteration)
    for $j = 1: d$  % loop over weights
      $\text{deltawj} = \eta \left( y(i) - \sigma[f(i)] \right) \frac{\partial \sigma[f(i)]}{\partial x_j(i)} x_j(i)$
      $w_j = w_j + \text{deltawj}$
    end
  calculate termination condition
end

- Inputs:  N features, N targets (class labels), learning rate $\eta$
- Outputs: a set of learned weights
Comments on Perceptron Learning

• Iteration = one pass through all of the data

• Algorithm presented = incremental gradient descent
  – Weights are updated after visiting each input example
  – Alternatives
    • Batch: update weights after each iteration (typically slower)
    • Stochastic: randomly select examples and then do weight updates

• A similar iterative algorithm learns weights for thresholded output (step function) perceptrons

• Rate of convergence
  – \(E[w]\) is convex as a function of \(w\), so no local minima
  – So convergence is guaranteed as long as learning rate is small enough
    • But if we make it too small, learning will be *very* slow
  – But if learning rate is too large, we move further, but can overshoot the solution and oscillate, and not converge at all
Support Vector Machines (SVM): “Modern perceptrons” (section 18.9, R&N)

• A modern linear separator classifier
  – Essentially, a perceptron with a few extra wrinkles

• Constructs a “maximum margin separator”
  – A linear decision boundary with the largest possible distance from the decision boundary to the example points it separates
  – “Margin” = Distance from decision boundary to closest example
  – The “maximum margin” helps SVMs to generalize well

• Can embed the data in a non-linear higher dimension space
  – Constructs a linear separating hyperplane in that space
    • This can be a non-linear boundary in the original space
    – Algorithmic advantages and simplicity of linear classifiers
    – Representational advantages of non-linear decision boundaries

• Currently most popular “off-the shelf” supervised classifier.
Multi-Layer Perceptrons (Artificial Neural Networks)  
(sections 18.7.3-18.7.4 in textbook)

• What if we took K perceptrons and trained them in parallel and then took a weighted sum of their sigmoidal outputs?
  – This is a multi-layer neural network with a single “hidden” layer (the outputs of the first set of perceptrons)
  – If we train them jointly in parallel, then intuitively different perceptrons could learn different parts of the solution
    • They define different local decision boundaries in the input space
• What if we hooked them up into a general Directed Acyclic Graph?
  – Can create simple “neural circuits” (but no feedback; not fully general)
  – Often called neural networks with hidden units

• How would we train such a model?
  – Backpropagation algorithm = clever way to do gradient descent
  – Bad news: many local minima and many parameters
    • training is hard and slow
  – Good news: can learn general non-linear decision boundaries
  – Generated much excitement in AI in the late 1980’s and 1990’s
  – Techniques like boosting, support vector machines, are often preferred
Multi-Layer Perceptrons (Artificial Neural Networks)  
(sections 18.7.3-18.7.4 in textbook)

\[ z = \text{XOR}(x, y) \]
Naïve Bayes Model  

Basic Idea: We want to estimate $P(C | X_1, \ldots, X_n)$, but it’s hard to think about computing the probability of a class from input attributes of an example.

Solution: Use Bayes’ Rule to turn $P(C | X_1, \ldots, X_n)$ into an equivalent expression that involves only $P(C)$ and $P(X_i | C)$.

We can estimate $P(C)$ easily from the frequency with which each class appears within our training data, and $P(X_i | C)$ from the frequency with which each $X_i$ appears in each class $C$ within our training data.
Naïve Bayes Model

Bayes Rule: \( P(C \mid X_1, \ldots, X_n) \) is proportional to \( P(C) \prod_i P(X_i \mid C) \)

[Note: denominator \( P(X_1, \ldots, X_n) \) is constant for all classes, may be ignored.]

Features \( X_i \) are conditionally independent given the class variable \( C \)
- choose the class value \( c_i \) with the highest \( P(c_i \mid x_1, \ldots, x_n) \)
- simple to implement, often works very well
- e.g., spam email classification: \( X \)'s = counts of words in emails

Conditional probabilities \( P(X_i \mid C) \) can easily be estimated from labeled data
- Problem: Need to avoid zeroes, e.g., from limited training data
- Solutions: Pseudo-counts, beta\([a,b]\) distribution, etc.
Naïve Bayes Model (2)

\[ P(C \mid X_1, \ldots, X_n) = \alpha \prod P(X_i \mid C) P(C) \]

Probabilities \( P(C) \) and \( P(X_i \mid C) \) can easily be estimated from labeled data:

\[ P(C = c_j) \approx \frac{\#(\text{Examples with class label } c_j)}{\#(\text{Examples})} \]

\[ P(X_i = x_{ik} \mid C = c_j) \approx \frac{\#(\text{Examples with } X_i \text{ value } x_{ik} \text{ and class label } c_j)}{\#(\text{Examples with class label } c_j)} \]

Usually easiest to work with logs:

\[
\log [ P(C \mid X_1, \ldots, X_n) ] \\
= \log \alpha + \sum [ \log P(X_i \mid C) + \log P(C) ]
\]

DANGER: Suppose ZERO examples with \( X_i \) value \( x_{ik} \) and class label \( c_j \)? An unseen example with \( X_i \) value \( x_{ik} \) will NEVER predict class label \( c_j \)!

Practical solutions: Pseudocounts, e.g., add 1 to every \( \#() \), etc.

Theoretical solutions: Bayesian inference, beta distribution, etc.
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Classifier Bias — Decision Tree or Linear Perceptron?
Summary

• Learning
  – Given a training data set, a class of models, and an error function, this is essentially a search or optimization problem

• Different approaches to learning
  – Divide-and-conquer: decision trees
  – Global decision boundary learning: perceptrons
  – Constructing classifiers incrementally: boosting

• Learning to recognize faces
  – Viola-Jones algorithm: state-of-the-art face detector, entirely learned from data, using boosting+decision-stumps