Supervised Learning

- Big idea: Learning as acquiring a function on feature vectors
- Background
  - Nearest Neighbors
  - Decision Trees
  - Neural Nets
  - Support Vector Machines

Machine Learning

Learning: Acquiring a function, based on past inputs and values, that can predict values for future (similar) inputs.

1. Learning how to pronounce words (letters → sounds)
2. Learning to throw a ball (target → torques)
3. Learning to recognize handwritten characters (images → characters)
4. Learning to diagnose disease from lab tests (lab results → disease identity)
5. Etc.

But, how about learning to program or to design a circuit or to tie shoelaces?

Machine Learning

Some basic distinctions:
1. Is the desired function output for an input known?
2. Is the function output value a continuous or discrete value?
3. What is the representation of the function to be learned?

Some general issues:
1. Representational adequacy: can the correct function be represented?
2. Complexity control: how complex a model is best?
3. Features: which subset of features is best?
Feature Space

Space where feature values define the coordinate axes. The feature vector for each instance defines a point in feature space.

A set of instances described by two feature values, \( x_1 \) and \( x_2 \).

Commonly, each feature vector may be labeled with a class label, for example, + or −.

Predicting Bankruptcy

A hypothetical data set

<table>
<thead>
<tr>
<th>Sample</th>
<th>L</th>
<th>R</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>0.2</td>
<td>No</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0.3</td>
<td>No</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>0.5</td>
<td>No</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
<td>0.7</td>
<td>No</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1.0</td>
<td>No</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1.2</td>
<td>No</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1.7</td>
<td>No</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>0.2</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>7</td>
<td>0.3</td>
<td>Yes</td>
</tr>
<tr>
<td>10</td>
<td>6</td>
<td>0.7</td>
<td>Yes</td>
</tr>
<tr>
<td>11</td>
<td>3</td>
<td>1.1</td>
<td>Yes</td>
</tr>
<tr>
<td>12</td>
<td>2</td>
<td>1.5</td>
<td>Yes</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>1.7</td>
<td>Yes</td>
</tr>
<tr>
<td>14</td>
<td>2</td>
<td>1.9</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Supervised Learning (and Un)

**Supervised Learning**: Given a large set of input vectors \( V_i \) and corresponding desired output vectors \( W_i \), learn a function \( F \) such that \( V_i, F(V_i) \Rightarrow \tilde{W}_i \), \( \tilde{W}_i \) is “close” to \( W_i \).

**Unsupervised Learning**: Given a large set of input vectors \( V_e \), find a simple description of them, for example, cluster them into classes or fit a mathematical model to them.

In this lecture, we will focus on supervised learning.

Types of Supervised Learning

**Classification**: the desired output \( y_i \) is a label for a small set of classes, for example, which letter of the alphabet, identity of disease, etc.

**Regression**: the desired output \( y_i \) is a vector of continuous variables, for example, steering signals for a car, predicted lifetime, etc.

In practice, both input and desired output data is noisy – there is error in both features values and desired value.
Representation Adequacy

Linear Discriminant

Linear discriminant $\sum w_x \geq 0$
learn weights on features to discriminate between 2 classes.

Good Case
Decision Boundary $w_1x_1+w_2x_2+w_3=0$
where predicted class changes.

Bad case, not linearly separable

Features

Feature definition
Choice of features determine whether a function may be learnt by a representation (or any representation). Clearly if the input is not correlated with the desired output, nothing can be learnt. But, even if correlation exists it may not be capturable by a particular representation for a particular feature choice.

Features

Scaling
Scaling can matter. Consider predicting outcome of horse races based on weight of horse and weight of jockey.
Choosing best feature subset matters

Perfect separation with one feature

include $x_1$ with slight correlation

Potential errors

Actual best

Looks like best separator

Features
Feature selection

Choosing best feature subset matters

Controlling Complexity

regression problem

Represent function as $n$th degree polynomial (any $n$)

$y = a_1 x + a_2$

(deg 1)

$y = a_1 x^2 + a_2 x + a_3$

(deg 2)

$y = a_1 x^b + \ldots + a_k + 1$

(deg $k$)

Fitting error decreases with increasing $n$

But, which $n$ gives best predictor?

Controlling Complexity

Generalization: Our goal is to obtain a function that is good at predicting unseen data that is similar, but not identical, to that in the input data set.

Given a general class of functions, with enough parameters, it is easy to produce a function that works perfectly on training data but is terrible on new data. This is known as overfitting the training data.

We must control the complexity of our function, for example, by limiting the number or magnitude of parameters, to avoid overfitting.

Avoiding Overfitting

Divide input data set into two sets:

1. Training set – used to choose parameters by improving the quality of fit.
2. Validation set – used to estimate predictive performance and stop training.

Requires more data...but can resample.
Resampling (Sample re-use)

We often don’t have enough data to estimate parameters and performance using independent data sets.

Cross-Validation – Divide the training data into K subsets. Cycle through the K subsets using each subset as a validation set and all the others as training set. Validation set is 1/K of the data and training set is (K-1)/K of the data.

Use average and standard deviation of performance on validation sets to characterize prediction performance (and confidence bounds).

Nearest Neighbor(s)

Supervised learning/Classification

• Definition of “near” needs to be settled (how to combine features of different types)
• Does no feature selection
• Susceptible to class labeling errors
• Prediction is costly when one has many features

Advantages:
• Fast training
• Little bias on type of function
• Very generally applicable

Disadvantages:
• Definition of “near” needs to be settled (how to combine features of different types)
• Does no feature selection
• Susceptible to class labeling errors
• Prediction is costly when one has many features

Training:
• Store all feature vectors in training set, each labeled with its class.

Prediction:
• Given a query feature vector, find “nearest” stored feature vector and return the stored class.

Normalization

• One approach to choosing weights for a distance function is to normalize all the features based on their statistical distribution.

• For example, we can normalize the data so that all features have a mean (average) value of 0 and a standard deviation of 1. To accomplish this we use the following transformation:

\[ x' = \frac{x - \mu}{\sigma_x} \]

\[ \mu = \text{mean value} \]
\[ \sigma_x = \text{standard deviation} \]
\[ T = \text{mean value} \]

• This normalization is not always the best choice but it is often a good place to start.
How to combine L and R into a distance? Typically weighted Euclidean distance. Assume weighting shown graphically (roughly 5 to 1)

\[ \sqrt{\sum (R_i - R_j)^2 + (L_i - L_j)^2} \]

**K- Nearest Neighbors**

- **K** nearest-neighbors – To reduce sensitivity to noise, use **K** (e.g. 3) nearest neighbors and have them vote. In the example below, the two No votes outweigh the single Yes vote.

- To choose best value of **K**, do cross-validation on training set. Note that this slows down training.

**Feature Selection**

- Not all the features in the input are equally useful. Some features may be so noisy or uncorrelated with the desired output that it is best to drop them.

- A simple approach to choosing features is based on evaluating the effectiveness of a set of features by cross-validation on the training set. Then, one could evaluate the effect of dropping one feature at a time and drop the least useful feature. Repeat while performance is non-decreasing.
Nearest Neighbors

Extensions

- **K-D Trees** – Clever algorithms exist for finding the nearest neighbor in high-dimensional spaces, for an example, see the description of K-D Trees in the text.

- **Locally weighted regression** – can do regression by combining values stored with the K-nearest neighbors, for example, return average weighted by (inverse square) distance from query point.

Decision Trees

**Supervised learning/Classification**

**Disadvantages:**
- Does not combine feature values, difficulty with dependent features.
- Finding optimal trees is intractable.

**Advantages:**
- Does feature selection
- Handles features of different types
- Very fast prediction
- Interpretable decision rule

Training:
- Divide feature space into boxes that have uniform labels. Split recursively along each axis to define a tree.

Prediction:
- Test features of new feature vector as dictated by the tree derived during training, return class at leaf of tree.

Occam’s razor: Simplest tree has best chance of generalizing, i.e. correctly classifying unseen cases. We take depth of the tree to measure complexity.

But, finding simplest tree is intractable. So, do a greedy search for simple tree.

Greedy Tree Builder

**Greedy_Tree_Builder(Data)**

```
    test = Pick_best_test(Data)
    split_list = Split_data(Data, test)
    for each subset in split_list
        Greedy_Tree_Builder(subset)
```

**Pick_best_test(Data)**

For each potential test

```
    Pick test producing least average disorder
    in split of Data based on test result
```

Hill climbing without backup, using average disorder as a heuristic measure of distance from a final state in which there is no disorder in the data, that is, all the data have the same class.
### Measuring Disorder

**Entropy**

Split \( m \) objects into two bins of size \( m_1 \) and size \( m_2 \), what is disorder?

- If all objects in one bin, disorder = 0
- If objects evenly split between two bins, disorder is maximal (= 1)
- Disorder measure should be symmetrical, interchanging contents of the bins should not matter

\[
\sum - \frac{m_i}{m} \log_2 \frac{m_i}{m} = - \frac{m_1}{m} \log_2 \frac{m_1}{m} - \frac{m_2}{m} \log_2 \frac{m_2}{m}
\]

### Entropy

**Entropy (disorder) of a split**

\[
p_i = \frac{m_i}{m} \quad \text{the probability of being in bin } i
\]

\[
0 \leq p_i \leq 1 \quad \sum p_i = 1
\]

\[
\sum - p_i \log_2 p_i
\]

Assume \( 0 \log_2 0 = 0 \)

<table>
<thead>
<tr>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>Entropy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>-1 log₂ 1 - 0 log₂ 0 = 0 - 0 = 0</td>
</tr>
<tr>
<td>( \frac{1}{2} )</td>
<td>( \frac{1}{2} )</td>
<td>( -\frac{1}{2} \log_2 \frac{1}{2} - \frac{1}{2} \log_2 \frac{1}{2} = \frac{1}{2} + \frac{1}{2} = 1 )</td>
</tr>
<tr>
<td>( \frac{1}{3} )</td>
<td>( \frac{2}{3} )</td>
<td>( -\frac{1}{3} \log_2 \frac{1}{3} - \frac{2}{3} \log_2 \frac{2}{3} = 0.5 + 0.311 = 0.811 )</td>
</tr>
</tbody>
</table>

### Decision Trees

**Extensions**

- **Discrete attributes with \( k \) values:** Use \( k \)-way split in tree, but need to watch out for attributes with large \( k \), their average disorder will be low but the split will be uninformative. In practice one needs to normalize for this effect.

- **Continuous attributes:** Consider a binary split between each attribute value in training set. \( O(n) \) splits to consider for \( n \) samples (for each attribute).

- **Samples with missing values:** Treat each sample with missing values as going down (fractionally) all branches of tests on the missing values. The fraction of the one sample down a branch is set to be the same as the fraction of the samples in the data set that would take that branch.
Decision Tree Predicting Bankruptcy

Test | No on left branch | Yes on left branch | No on right branch | Yes on right branch | Average Disorder
--- | --- | --- | --- | --- | ---
L<5 | Y | N | Y | N | 0.92
0.8 | 3 | 0 | 0 | 3 | 0.90
0.5 | 2 | 0 | 1 | 3 | 0.59
L<5 | N | Y | N | Y | 0.60
0.8 | 3 | 0 | 0 | 3 | 0.80
0.5 | 2 | 0 | 1 | 3 | 0.60

AD 0.90 0.76 0.69 0.60 0.79 0.88 0.85
R<0.85
L<1.5
Y
Y
N
N
No
Yes
No
Yes

Note that R<0.85 has much higher average disorder (0.98) than L<1.5 (0.63) but leads to a good tree.
**Decision Trees**

*Overfitting*

Given enough features, we can typically keep subdividing the data until every leaf has a single class. But this may overfit the data.

Build an decision tree to predict the roll of a fair die by using a training set with the following features:

1. Month
2. Day
3. Minute
4. Dow Jones Average
5. Phase of the moon

We can build a tree in which every roll ends up in a unique leaf, but...

We want to either stop growing tree to avoid overfitting or prune back an overly complex tree to improve its ability to generalize. Generally, pruning is more effective.

**Rules and Trees**

Given a tree, we can always construct equivalent rules.

\[
\begin{align*}
R_1 & \text{ if } A_1 = r_1 \\
R_2 & \text{ if } A_1 = r_1 \\
R_3 & \text{ if } A_1 = s_1 \\
& \text{ then } + \quad \text{ then } + \quad \text{ then } - \\
& \text{then } + \quad \text{then } -
\end{align*}
\]

Each leaf in the tree can be represented by a rule.

However, rules are more general than trees, there are rules that cannot be modeled as trees.

**Rules and Trees**

Data set with four features F, G, J, K

Assume every + has value 1 for F and G or J and K

\[
R_1 \text{ if } F=1 \\
R_2 \text{ if } J=1 \\
R_3 \text{ if } F=0 \\
& \text{ then } + \quad \text{then } + \quad \text{then } - \\
& \text{then } + \quad \text{then } -
\]

We can’t capture this in a tree. Best you can do is order the tests. Note that there are “extra” tests on the rules derived from this tree

\[
\begin{align*}
& \text{If } F=1 \\
& \text{G=0} \\
& \text{J=1} \\
& \text{K=1} \\
& \text{Check to see that dropping tests does not introduce errors}
\end{align*}
\]

If J=1

\[
\begin{align*}
& \text{K=1} \\
& \text{Then } +
\end{align*}
\]

Check to see that dropping tests does not introduce errors

**Neural Nets**

*Supervised learning*

Training:
- Choose connection weights that minimize error

Prediction:
- Propagate input feature values through the network of “artificial neurons”

Advantages:
- Fast prediction
- Does feature weighting
- Very generally applicable

Disadvantages:
- Very slow training
- Overfitting is easy
Perceptron Unit

Single unit

\[
y = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 & \text{otherwise} \end{cases}
\]

Note: \( w, x, w, x, w, p \) 0

alternatively \( [w, w] \) \( x_i \) = \( w_0 \)

I.e. locus of \((x_1, x_2)\) where dot product with vector \([w, w]\) is equal to constant \(w_0\). This is true of points on a line perpendicular to vector \([w, w]\).

*adj* equation:

\[
x_1 \cdot w = w_0
\]

\[
\text{Let } w = ||[w, w]|| = (w_1^2 + w_2^2)^{1/2}
\]

Perceptron Decision Boundaries

This type of decision boundary can handle linearly separable problems

But not

Perceptron Learning Rule

\[
\Delta w = \eta (y - y') \bar{x}
\]

Change weights on non-zero inputs in direction of error.

Guaranteed to converge for linearly separable problems; chaotic otherwise

Line (Plane) Equations

Equation of a line: \( ax + by - c = 0 \)

When \( \sqrt{a^2 + b^2} = 1 \) we have \( \hat{n} = [a, b] \)

\( \hat{n} \cdot \bar{x} = c \) offset normal

Every point \( \bar{x} \) on the line has: \( \hat{n} \cdot \bar{x} = c \)

Points in the half-space away from where normal points have: \( \hat{n} \cdot \bar{x} \leq c \)

Points in the half-space towards where normal points have: \( \hat{n} \cdot \bar{x} \geq c \)

When \( \sqrt{a^2 + b^2} \neq 1 \) divide offset by magnitude of normal vector \( c/\sqrt{a^2 + b^2} \)
Multi-Layer Perceptron

More powerful than single layer. Lower layers transform the input problem into more tractable (linearly separable) problems for subsequent layers. But, there is no general learning rule!

XOR Problem

Not linearly separable

Linearly separable

Sigmoid Unit

Training

Error over the training set for a given weight vector:

Our goal is to find weight vector that minimizes error

Desired output: $y^*$
Minimizing by Gradient Descent

**Basic Problem:** Given initial parameters $w_0$ with error $E = G(w_0)$, find $w$ that is local minimum of $G$, that is, all first derivatives of $G$ are zero.

**Approach:**
1. Find "slope" of $G$ at $w_0$
2. Move to $w_1$ which is "down slope"

$$
\nabla_{w} G = \left[ \frac{\partial G}{\partial w_0}, \frac{\partial G}{\partial w_1}, \ldots, \frac{\partial G}{\partial w_n} \right]
$$

Gradient is n-dimensional "slope"

$$
w_{0+1} = w_1 - r \nabla_{w} G
$$

Gradient descent, $r$ is "rate" – a small step size

Stop when gradient is very small (or change in $G$ is very small)

---

An Example

$$
E = G(w) = \frac{1}{2} (w_1^2 + w_2^2)
$$

$$
\nabla_{w} G = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix}
$$

Note: for non-linear function, gradient changes as a function of "position".

Gradient vector is always perpendicular to contour lines.

---

Gradient of Error

$$
E = \frac{1}{2} \sum_i (y_i - F(x_i, \theta))^2
$$

$$
y_i = F(x_i, \theta) = s(w_1 x_{i1} + w_2 x_{i2} - w_0)
$$

$$
\frac{\partial E}{\partial w_j} = -(y_j - y_i) \frac{\partial y_j}{\partial w_j}
$$

$$
\frac{\partial y_j}{\partial w_j} = s(z_j) \left( \frac{z_j}{\partial z_j} s(z_j) \right)
$$

Derivative of the sigmoid

$$
s(z) = \frac{1}{1 + e^{-z}}
$$

$$
\frac{d s(z)}{dz} = \left[ \frac{1 + e^{-z}}{1 + e^{-z}} \right] e^{-z} = s(z) (1 - s(z))
$$
Backpropagation
An efficient method of implementing gradient descent for neural networks

\[ w_{i,j} = w_{i,j} - r \delta_j y_i \]

Descent rule

\[ \delta_j = \frac{dE}{dy_j} \sum_k \delta_k w_{j,k} \]

Backprop rule

1. Initialize weights to small random values
2. Choose a random sample input feature vector
3. Compute total input \((z_j)\) and output \((y_i)\) for each unit (forward prop)
4. Compute \(\delta_j\) for output layer
5. Compute \(\delta_i\) for preceding layer by backprop rule (repeat for all layers)
6. Compute weight change by descent rule (repeat for all weights)

\[ \delta_j = y_j (1 - y_j) (y_j - y'_j) \]

\[ \delta_i = y_i (1 - y_i) \delta_j w_{i,j} \]

\[ \delta_i = y_i (1 - y_i) \delta_k w_{i,k} \]

Why the sum over \(k\)? Consider the simplified net:

\[ E = \frac{1}{2} [(y_j - y'_j)^2 + (y_k - y'_k)^2] \]

A change in \(w_1\) affects the error via a change in \(y_1\), which affects \(y_2\) and \(y_3\)

Backprop Example
Forward prop: Compute \(z_j\) and \(y_i\) given \(x_k\), \(w_l\)

\[ \delta_3 = y'_3 (1 - y_3) (y_3 - y'_3) \]

\[ \delta_2 = y'_2 (1 - y_2) \delta_3 w_{23} \]

\[ \delta_1 = y'_1 (1 - y_1) \delta_j w_{13} \]

\[ w_{33} = w_{33} - r \delta_3 (-1) \]

\[ w_{22} = w_{22} - r \delta_2 (-1) \]

\[ w_{11} = w_{11} - r \delta_1 (-1) \]

Compare to the direct derivation earlier

Note that all computations are local!

Training Neural Nets

Without overfitting, hopefully...

Given: Data set, desired outputs and a neural net with \(m\) weights. Find a setting for the weights that will give good predictive performance on new data. Estimate expected performance on new data.

1. Split data set (randomly) into three subsets:
   - Training set – used for picking weights
   - Validation set – used to stop training
   - Test set – used to evaluate performance
2. Pick random, small weights as initial values
3. Perform iterative minimization of error over training set
4. Stop when error on validation set reaches a minimum (to avoid overfitting).
5. Use best weights to compute error on test set, which is estimate of performance on new data. Do not repeat training to improve this.

Can use cross-validation if data set is too small to divide into three subsets.
On-line vs off-line

There are two approaches to performing the error minimization:
- **On-line training** – present X<sub>i</sub> and Y<sub>i</sub> (chosen randomly from the training set). Change the weights to reduce the error on this instance. Repeat.
- **Off-line training** – change weights to reduce the total error on training set (sum over all instances).

On-line training is a stochastic approximation to gradient descent since the gradient based on one instance is "noisy" relative to the full gradient (based on all instances). This can be beneficial in pushing the system out of shallow local minima.

Classification vs Regression

A neural net with a single sigmoid output unit is aimed at binary classification. Class is 0 if y<0.5 and 1 otherwise.

For multiway classification, use one output unit per class.

A sigmoid output unit is not suitable for regression, since sigmoids are designed to change quickly from 0 to 1. For regression, we want a linear output unit, that is, remove the output non-linearity. The rest of the net still retains the sigmoid units.

Support Vector Machines: Foundations

- Key choice of separation strategy:
  - Maximize "street" between groups
- Attack maximization problem:
  - Lagrange multipliers
- New problem is a quadratic minimization:
  - Susceptible to numerical methods
- Result depends on dot products only
  - Enables use of kernel methods.