Instance-Based Learning

Instructor: Alan Ritter

Many Slides from Pedro Domingos
Instance-Based Learning

**Key idea:** Just store all training examples $\langle x_i, f(x_i) \rangle$

**Nearest neighbor:**
- Given query instance $x_q$, first locate nearest training example $x_n$, then estimate $\hat{f}(x_q) \leftarrow f(x_n)$

**$k$-Nearest neighbor:**
- Given $x_q$, take vote among its $k$ nearest neighbors (if discrete-valued target function)
- Take mean of $f$ values of $k$ nearest neighbors (if real-valued)

$$\hat{f}(x_q) \leftarrow \frac{1}{k} \sum_{i=1}^{k} f(x_i)$$
Advantages and Disadvantages

Advantages:

- Training is very fast
- Learn complex target functions easily
- Don’t lose information

Disadvantages:

- Slow at query time
- Lots of storage
- Easily fooled by irrelevant attributes
Distance Measures

• Numeric features:
  – Euclidean, Manhattan, $L^n$-norm:
    $$L^n(x_1, x_2) = \sqrt[n]{\sum_{i=1}^{\#\text{dim}} |x_{1,i} - x_{2,i}|^n}$$
  – Normalized by: range, std. deviation

• Symbolic features:
  – Hamming/overlap
  – Value difference measure (VDM):
    $$\delta(val_i, val_j) = \sum_{h=1}^{\#\text{classes}} |P(c_h | val_i) - P(c_h | val_j)|^n$$

• In general: arbitrary, encode knowledge
Voronoi Diagram

$S$: Training set

**Voronoi cell of** $x \in S$:
All points closer to $x$ than to any other instance in $S$

**Region of class** $C$:
Union of Voronoi cells of instances of $C$ in $S$
Distance-Weighted $k$-NN

Might want to weight nearer neighbors more heavily ... 

$$\hat{f}(x_q) \leftarrow \frac{\sum_{i=1}^{k} w_i f(x_i)}{\sum_{i=1}^{k} w_i}$$

where 

$$w_i \equiv \frac{1}{d(x_q, x_i)^2}$$

and $d(x_q, x_i)$ is distance between $x_q$ and $x_i$

Notice that now it makes sense to use all training examples instead of just $k$: 
Curse of Dimensionality

• Imagine instances described by 20 attributes, but only 2 are relevant to target function

• **Curse of dimensionality:**
  – Nearest neighbor is easily misled when hi-dim $X$
  – Easy problems in low-dim are hard in hi-dim
  – Low-dim intuitions don’t apply in hi-dim

• **Examples:**
  – Normal distribution
  – Uniform distribution on hypercube
  – Points on hypergrid
  – Approximation of sphere by cube
  – Volume of hypersphere
Things Get Weird in High Dimensions

» High-Dimensional Spheres look like porcupines instead of balls

» Distances between points in high dimensions are all about the same
Things Get Weird in High Dimensions

» High-Dimensional Spheres

Pythagorean theorem says:

\[ 1^2 + 1^2 = (1 + r)^2 \]

\[ r = \sqrt{2} - 1 \approx 0.41 \]

Inside the unit square
Things Get Weird in High Dimensions

» High-Dimensional Spheres

3d Pythagorean theorem says:

\[ 1^2 + 1^2 + 1^2 = (1 + r)^2 \]

Thus

\[ r = \sqrt{3} - 1 \approx 0.73 \]

Bigger, but still inside the unit cube

Figure 3.17: 3d spheres in spheres
Things Get Weird in High Dimensions

» High-Dimensional Spheres

\[ D = N \]

\[ r = \sqrt{N} - 1 \]

\( N = 1000 \Rightarrow r = 30.6 \)

Radius of the middle hypersphere extends way beyond the unit hypercube
Things Get Weird in High Dimensions

» Distances between points
  » Maximum Distance between any two points in a unit hypercube grows as $\sqrt{D}$
  » But can show that variance is constant (independet of D): $\frac{1}{\sqrt{18}}$

» Effective variance behaves as: $\frac{1}{\sqrt{18D}}$
Things Get Weird in High Dimensions

```python
import numpy as np
import matplotlib.pyplot as plt
import sys
import math

N = 100

def DistancesBetweenRandomPoints(D):
    data = np.random.random((N,D))

    dsum = 0.0
    dlist = []
    for i in range(N):
        for j in range(N):
            if i != j:
                dlist += [np.linalg.norm(data[i,:] - data[j,:])] # Euclidean distance between rows i and j
            count += 1
    return dlist

plt.hist([(x / math.sqrt(2)) for x in DistancesBetweenRandomPoints(2)])
plt.hist([(x / math.sqrt(10)) for x in DistancesBetweenRandomPoints(10)])
plt.hist([(x / math.sqrt(100)) for x in DistancesBetweenRandomPoints(100)])
plt.hist([(x / math.sqrt(1000)) for x in DistancesBetweenRandomPoints(1000)])
plt.hist([(x / math.sqrt(10000)) for x in DistancesBetweenRandomPoints(10000)])
plt.show()
```
Things Get Weird in High Dimensions
Feature Selection

- **Filter approach:**
  Pre-select features individually
  - E.g., by info gain

- **Wrapper approach:**
  Run learner with different combinations of features
  - Forward selection
  - Backward elimination
  - Etc.
**FORWARD SELECTION**\((FS)\)

\(FS\): Set of features used to describe examples

Let \(SS = \emptyset\)

Let \(BestEval = 0\)

Repeat

- Let \(BestF = \text{None}\)
  - For each feature \(F\) in \(FS\) and not in \(SS\)
    - Let \(SS' = SS \cup \{F\}\)
    - If \(\text{Eval}(SS') > BestEval\)
      - Then Let \(BestF = F\)
      - Let \(BestEval = \text{Eval}(SS')\)
  
- If \(BestF \neq \text{None}\)
  - Then Let \(SS = SS \cup \{BestF\}\)

Until \(BestF = \text{None}\) or \(SS = FS\)

Return \(SS\)
BACKWARD_ELIMINATION($FS$)

$FS$: Set of features used to describe examples

Let $SS = FS$
Let $BestEval = Eval(SS)$
Repeat
  Let $WorstF = None$
  For each feature $F$ in $SS$
    Let $SS' = SS - \{F\}$
    If $Eval(SS') \geq BestEval$
      Then Let $WorstF = F$
      Let $BestEval = Eval(SS')$
  If $WorstF \neq None$
    Then Let $SS = SS - \{WorstF\}$
Until $WorstF = None$ or $SS = \emptyset$
Return $SS$
Reducing Computational Cost

- Efficient retrieval: $k$-D trees
  (only work in low dimensions)

- Efficient similarity comparison:
  - Use cheap approx. to weed out most instances
  - Use expensive measure on remainder

- Form prototypes

- Edited $k$-NN:
  Remove instances that don’t affect frontier
Overfitting Avoidance

- Set $k$ by cross-validation
- Form prototypes
- Remove noisy instances
  - E.g., remove $x$ if all of $x$’s $k$ nearest neighbors are of another class
Collaborative Filtering
(AKA Recommender Systems)

- **Problem:**
  Predict whether someone will like a Web page, newsgroup posting, movie, book, CD, etc.

- **Previous approach:**
  Look at content

- **Collaborative filtering:**
  - Look at what similar users liked
  - Similar users = Similar likes & dislikes
Collaborative Filtering

- Represent each user by vector of ratings
- Two types:
  - Yes/No
  - Explicit ratings (e.g., 0 - *****)
- Predict rating:
  \[ \hat{R}_{ik} = \bar{R}_i + \alpha \sum_{X_j \in N_i} W_{ij} (R_{jk} - \bar{R}_j) \]
- Similarity (Pearson coefficient):
  \[ W_{ij} = \frac{\sum_k (R_{ik} - \bar{R}_i)(R_{jk} - \bar{R}_j)}{\sqrt{\sum_k (R_{ik} - \bar{R}_i)^2} \sqrt{\sum_k (R_{jk} - \bar{R}_j)^2}} \]
Fine Points

- Primitive version:

\[ \hat{R}_{ik} = \alpha \sum_{X_j \in \mathcal{N}_i} W_{ij} R_{jk} \]

- \( \alpha = (\sum |W_{ij}|)^{-1} \)

- \( \mathcal{N}_i \) can be whole database, or only \( k \) nearest neighbors

- \( R_{jk} = \) Rating of user \( j \) on item \( k \)

- \( \bar{R}_j = \) Average of all of user \( j \)'s ratings

- Summation in Pearson coefficient is over all items rated by both users

- In principle, any prediction method can be used for collaborative filtering
**Example**

<table>
<thead>
<tr>
<th></th>
<th>$R_1$</th>
<th>$R_2$</th>
<th>$R_3$</th>
<th>$R_4$</th>
<th>$R_5$</th>
<th>$R_6$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>2</td>
<td>-</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>5</td>
</tr>
<tr>
<td>Bob</td>
<td>1</td>
<td>5</td>
<td>4</td>
<td>-</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>Chris</td>
<td>5</td>
<td>2</td>
<td>-</td>
<td>2</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>Diana</td>
<td>3</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>4</td>
</tr>
</tbody>
</table>