Naïve Bayes
Machine Learning

- Up until now: how to use a model to make optimal decisions

- Machine learning: how to acquire a model from data / experience
  - Learning parameters (e.g. probabilities)
  - Learning structure (e.g. BN graphs)
  - Learning hidden concepts (e.g. clustering, neural nets)

- Today: model-based classification with Naive Bayes
Classification
Example: Spam Filter

- **Input:** an email
- **Output:** spam/ham

- **Setup:**
  - Get a large collection of example emails, each labeled “spam” or “ham”
  - Note: someone has to hand label all this data!
  - Want to learn to predict labels of new, future emails

- **Features:** The attributes used to make the ham / spam decision
  - Words: FREE!
  - Text Patterns: $dd, CAPS
  - Non-text: SenderInContacts, WidelyBroadcast
  - ...

---

Dear Sir.

First, I must solicit your confidence in this transaction, this is by virtue of its nature as being utterly confidential and top secret. ... 

TO BE REMOVED FROM FUTURE MAILINGS, SIMPLY REPLY TO THIS MESSAGE AND PUT "REMOVE" IN THE SUBJECT.

99 MILLION EMAIL ADDRESSES FOR ONLY $99

Ok, I know this is blatantly OT but I'm beginning to go insane. Had an old Dell Dimension XPS sitting in the corner and decided to put it to use, I know it was working pre being stuck in the corner, but when I plugged it in, hit the power nothing happened.
Example: Digit Recognition

- Input: images / pixel grids
- Output: a digit 0-9

Setup:
- Get a large collection of example images, each labeled with a digit
- Note: someone has to hand label all this data!
- Want to learn to predict labels of new, future digit images

Features: The attributes used to make the digit decision
- Pixels: (6,8)=ON
- Shape Patterns: NumComponents, AspectRatio, NumLoops
- ...
- Features are increasingly induced rather than crafted
Other Classification Tasks

- Classification: given inputs $x$, predict labels (classes) $y$

- Examples:
  - Medical diagnosis (input: symptoms, classes: diseases)
  - Fraud detection (input: account activity, classes: fraud / no fraud)
  - Automatic essay grading (input: document, classes: grades)
  - Customer service email routing
  - Review sentiment
  - Language ID
  - ... many more

- Classification is an important commercial technology!
Model-Based Classification

- **Model-based approach**
  - Build a model (e.g. Bayes’ net) where both the output label and input features are random variables
  - Instantiate any observed features
  - Query for the distribution of the label conditioned on the features

- **Challenges**
  - What structure should the BN have?
  - How should we learn its parameters?
Naïve Bayes for Digits

- Naïve Bayes: Assume all features are independent effects of the label

- Simple digit recognition version:
  - One feature (variable) $F_{i,j}$ for each grid position $<i,j>$
  - Feature values are on / off, based on whether intensity is more or less than 0.5 in underlying image
  - Each input maps to a feature vector, e.g.:
    \[ \{F_{0,0} = 0, F_{0,1} = 0, F_{0,2} = 1, F_{0,3} = 1, F_{0,4} = 0, \ldots F_{15,15} = 0\} \]
  - Here: lots of features, each is binary valued

- Naïve Bayes model:
  \[ P(Y|F_{0,0} \ldots F_{15,15}) \propto P(Y) \prod_{i,j} P(F_{i,j}|Y) \]
- What do we need to learn?
A general Naive Bayes model:

\[
P(Y, F_1 \ldots F_n) = P(Y) \prod_i P(F_i|Y)
\]

- We only have to specify how each feature depends on the class
- Total number of parameters is \textit{linear} in \(n\)
- Model is very simplistic, but often works anyway
Inference for Naïve Bayes

- **Goal:** compute posterior distribution over label variable Y
  - **Step 1:** get joint probability of label and evidence for each label
    
    \[
    P(Y, f_1 \ldots f_n) = \begin{bmatrix}
    P(y_1, f_1 \ldots f_n) \\
    P(y_2, f_1 \ldots f_n) \\
    \vdots \\
    P(y_k, f_1 \ldots f_n)
    \end{bmatrix}
    \]
    
    \[
    \frac{P(f_1) \prod_i P(f_i | y_1)}{P(f_2) \prod_i P(f_i | y_2)} \\
    \vdots \\
    \frac{P(f_k) \prod_i P(f_i | y_k)}{P(f_1 \ldots f_n)}
    \]

- **Step 2:** sum to get probability of evidence

- **Step 3:** normalize by dividing Step 1 by Step 2

\[
P(Y | f_1 \ldots f_n) = \frac{P(Y, f_1 \ldots f_n)}{P(f_1 \ldots f_n)}
\]
General Naïve Bayes

What do we need in order to use Naïve Bayes?

- Inference method (we just saw this part)
  - Start with a bunch of probabilities: \( P(Y) \) and the \( P(F_i|Y) \) tables
  - Use standard inference to compute \( P(Y|F_1...F_n) \)
  - Nothing new here

- Estimates of local conditional probability tables
  - \( P(Y) \), the prior over labels
  - \( P(F_i|Y) \) for each feature (evidence variable)
  - These probabilities are collectively called the \textit{parameters} of the model and denoted by \( \theta \)
  - Up until now, we assumed these appeared by magic, but...
  - ...they typically come from training data counts: we’ll look at this soon
Example: Conditional Probabilities

\[ P(Y) \]

\[
\begin{array}{c|c}
1 & 0.1 \\
2 & 0.1 \\
3 & 0.1 \\
4 & 0.1 \\
5 & 0.1 \\
6 & 0.1 \\
7 & 0.1 \\
8 & 0.1 \\
9 & 0.1 \\
0 & 0.1 \\
\end{array}
\]

\[ P(F_{3,1} = \text{on}|Y) \]  \[ P(F_{5,5} = \text{on}|Y) \]

\[
\begin{array}{c|c}
1 & 0.01 \\
2 & 0.05 \\
3 & 0.05 \\
4 & 0.30 \\
5 & 0.80 \\
6 & 0.90 \\
7 & 0.05 \\
8 & 0.60 \\
9 & 0.50 \\
0 & 0.80 \\
\end{array}
\]

\[
\begin{array}{c|c}
1 & 0.05 \\
2 & 0.01 \\
3 & 0.90 \\
4 & 0.80 \\
5 & 0.90 \\
6 & 0.90 \\
7 & 0.25 \\
8 & 0.85 \\
9 & 0.60 \\
0 & 0.80 \\
\end{array}
\]
Naïve Bayes for Text

- **Bag-of-words Naïve Bayes:**
  - Features: $W_i$ is the word at position $i$
  - As before: predict label conditioned on feature variables (spam vs. ham)
  - As before: assume features are conditionally independent given label
  - New: each $W_i$ is identically distributed

- **Generative model:**
  
  $P(Y, W_1 \ldots W_n) = P(Y) \prod_i P(W_i|Y)$

- **“Tied” distributions and bag-of-words**
  - Usually, each variable gets its own conditional probability distribution $P(F|Y)$
  - In a bag-of-words model
    - Each position is identically distributed
    - All positions share the same conditional probs $P(W|Y)$
    - Why make this assumption? *regardless of position 23 or 49? Same word distribution given class*
  - Called “bag-of-words” because model is insensitive to word order or reordering

*word distribution

Word at position

$i$, not $i^{th}$ word in

the dictionary!

*regardless of position 23 or 49?

Same word distribution given class

*word or phrase?? bag of phrase

*doc length long or short
Example: Spam Filtering

- **Model:** \[ P(Y, W_1 \ldots W_n) = P(Y) \prod_i P(W_i|Y) \]

- **What are the parameters?**

\[
\begin{array}{l}
\text{P(Y)} \\
\hline
\text{ham: 0.66} \\
\text{spam: 0.33} \\
\hline
\end{array}
\]

\[
\begin{array}{l}
\text{P(W|spam)} \\
\hline
\text{the: 0.0156} \\
\text{to: 0.0153} \\
\text{and: 0.0115} \\
\text{of: 0.0095} \\
\text{you: 0.0093} \\
\text{a: 0.0086} \\
\text{with: 0.0080} \\
\text{from: 0.0075} \\
\text{...} \\
\hline
\end{array}
\]

\[
\begin{array}{l}
\text{P(W|ham)} \\
\hline
\text{the: 0.0210} \\
\text{to: 0.0133} \\
\text{of: 0.0119} \\
\text{2002: 0.0110} \\
\text{with: 0.0108} \\
\text{from: 0.0107} \\
\text{and: 0.0105} \\
\text{a: 0.0100} \\
\text{...} \\
\hline
\end{array}
\]

- **Where do these tables come from?**
Empirical Risk Minimization

- **Empirical risk minimization**
  - Basic principle of machine learning
  - We want the model (classifier, etc) that does best on the true test distribution
  - Don’t know the true distribution so pick the best model on our actual training set
  - Finding “the best” model on the training set is phrased as an optimization problem

- **Main worry: overfitting to the training set**
  - Better with more training data (less sampling variance, training more like test)
  - Better if we limit the complexity of our hypotheses (regularization and/or small hypothesis spaces)
Important Concepts

- Data: labeled instances (e.g. emails marked spam/ham)
  - Training set
  - Held out set
  - Test set

- Features: attribute-value pairs which characterize each x

- Experimentation cycle
  - Learn parameters (e.g. model probabilities) on training set
  - (Tune hyperparameters on held-out set)
  - Compute accuracy of test set
  - Very important: never “peek” at the test set!

- Evaluation (many metrics possible, e.g. accuracy)
  - Accuracy: fraction of instances predicted correctly

- Overfitting and generalization
  - Want a classifier which does well on test data
  - Overfitting: fitting the training data very closely, but not generalizing well
  - We’ll investigate overfitting and generalization formally in a few lectures
Overfitting

Degree 15 polynomial
Example: Overfitting

\[ P(\text{features, } C = 2) \]

\[ P(C = 2) = 0.1 \]
\[ P(\text{on}|C = 2) = 0.8 \]
\[ P(\text{off}|C = 2) = 0.1 \]
\[ P(\text{on}|C = 2) = 0.01 \]

\[ P(\text{features, } C = 3) \]

\[ P(C = 3) = 0.1 \]
\[ P(\text{on}|C = 3) = 0.8 \]
\[ P(\text{off}|C = 3) = 0.7 \]
\[ P(\text{on}|C = 3) = 0.0 \]

*Over 3 due to 0 prob.

2 wins!!
Example: Overfitting

- Posterior determined by *relative* probabilities (odds ratios):

\[
\frac{P(W|\text{ham})}{P(W|\text{spam})} \quad \frac{P(W|\text{spam})}{P(W|\text{ham})}
\]

| south-west : inf | screens : inf |
| nation : inf | minute : inf |
| morally : inf | guaranteed : inf |
| nicely : inf | $205.00$ : inf |
| extent : inf | delivery : inf |
| seriously : inf | signature : inf |

...  

*infinity due to zero probability*
Generalization and Overfitting

- Relative frequency parameters will overfit the training data!
  - Just because we never saw a 3 with pixel (15,15) on during training doesn’t mean we won’t see it at test time
  - Unlikely that every occurrence of “minute” is 100% spam
  - Unlikely that every occurrence of “seriously” is 100% ham
  - What about all the words that don’t occur in the training set at all?
  - In general, we can’t go around giving unseen events zero probability

- As an extreme case, imagine using the entire email as the only feature (e.g. document ID)
  - Would get the training data perfect (if deterministic labeling)
  - Wouldn’t generalize at all
  - Just making the bag-of-words assumption gives us some generalization, but isn’t enough

- To generalize better: we need to smooth or regularize the estimates
Parameter Estimation

- Estimating the distribution of a random variable
- **Elicitation**: ask a human (why is this hard?)
- **Empirically**: use training data (learning!)
  - E.g.: for each outcome $x$, look at the *empirical rate* of that value:
    \[
P_{ML}(x) = \frac{\text{count}(x)}{\text{total samples}}
\]

- This is the estimate that maximizes the *likelihood of the data*

\[
L(x, \theta) = \prod_{i} P_{\theta}(x_i)
\]

*P(data|parameter theta such as class)
Maximum Likelihood?

- Relative frequencies are the maximum likelihood estimates

\[ \theta_{ML} = \arg \max_\theta P(X|\theta) \]
\[ = \arg \max_\theta \prod_i P_\theta(X_i) \]
\[ \quad \Rightarrow \quad P_{ML}(x) = \frac{\text{count}(x)}{\text{total samples}} \]

- Another option is to consider the most likely parameter value given the data

\[ \theta_{MAP} = \arg \max_\theta P(\theta|X) \]
\[ = \arg \max_\theta P(X|\theta)P(\theta)/P(X) \]
\[ = \arg \max_\theta P(X|\theta)P(\theta) \]
Unseen Events
Laplace Smoothing

- Laplace’s estimate:
  - Pretend you saw every outcome once more than you actually did

\[
P_{\text{LAP}}(x) = \frac{c(x) + 1}{\sum_x [c(x) + 1]} = \frac{c(x) + 1}{N + |X|}
\]

Data size = 3, vocabulary = 2

\[
P_{\text{ML}}(X) = \frac{N_{\text{r}}}{N}
\]

\[
P_{\text{LAP}}(X) = \frac{N_{\text{r}} + 1}{N + 1}
\]

\[
\begin{align*}
\text{r} & \quad \text{r} & \quad \text{b} \\
& & \quad 2/3, 1/3 \\
& & 3/5, 2/5
\end{align*}
\]
Laplace Smoothing

- Laplace’s estimate (extended):
  - Pretend you saw every outcome $k$ extra times

  $$P_{LAP,k}(x) = \frac{c(x) + k}{N + k|X|}$$

  - What’s Laplace with $k = 0$?
  - $k$ is the **strength** of the prior

- Laplace for conditionals:
  - Smooth each condition independently:

  $$P_{LAP,k}(x|y) = \frac{c(x, y) + k}{c(y) + k|X|}$$

  - $P_{LAP,0}(X) =$
  - $P_{LAP,1}(X) =$
  - $P_{LAP,100}(X) =$

  *more generalization?

*102/203, 101/203
Estimation: Linear Interpolation*

- In practice, Laplace often performs poorly for $P(X|Y)$:
  - When $|X|$ is very large
  - When $|Y|$ is very large

- Another option: linear interpolation
  - Also get the empirical $P(X)$ from the data
  - Make sure the estimate of $P(X|Y)$ isn’t too different from the empirical $P(X)$

$$P_{LIN}(x|y) = \alpha \hat{P}(x|y) + (1.0 - \alpha) \hat{P}(x)$$

- What if $\alpha$ is 0? 1?
Real NB: Smoothing

- For real classification problems, smoothing is critical
- New odds ratios:

\[
\frac{P(W|\text{ham})}{P(W|\text{spam})} \quad \frac{P(W|\text{spam})}{P(W|\text{ham})}
\]

<table>
<thead>
<tr>
<th>ham</th>
<th>spam</th>
</tr>
</thead>
<tbody>
<tr>
<td>helvetica</td>
<td>11.4</td>
</tr>
<tr>
<td>seems</td>
<td>10.8</td>
</tr>
<tr>
<td>group</td>
<td>10.2</td>
</tr>
<tr>
<td>ago</td>
<td>8.4</td>
</tr>
<tr>
<td>areas</td>
<td>8.3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Now we’ve got two kinds of unknowns
- Parameters: the probabilities $P(X|Y)$, $P(Y)$
- Hyperparameters: e.g. the amount / type of smoothing to do, $k$, $\alpha$

What should we learn where?
- Learn parameters from training data
- Tune hyperparameters on different data
  - Why?
- For each value of the hyperparameters, train and test on the held-out data
- Choose the best value and do a final test on the test data
Errors, and What to Do

Examples of errors

Dear GlobalSCAPE Customer,

GlobalSCAPE has partnered with ScanSoft to offer you the latest version of OmniPage Pro, for just $99.99* - the regular list price is $499! The most common question we've received about this offer is - Is this genuine? We would like to assure you that this offer is authorized by ScanSoft, is genuine and valid. You can get the...

To receive your $30 Amazon.com promotional certificate, click through to http://www.amazon.com/apparel and see the prominent link for the $30 offer. All details are there. We hope you enjoyed receiving this message. However, if you'd rather not receive future e-mails announcing new store launches, please click...
What to Do About Errors?

- Need more features—words aren’t enough!
  - Have you emailed the sender before?
  - Have 1K other people just gotten the same email?
  - Is the sending information consistent?
  - Is the email in ALL CAPS?
  - Do inline URLs point where they say they point?
  - Does the email address you by (your) name?

- Can add these information sources as new variables in the NB model

- Next class we’ll talk about classifiers which let you easily add arbitrary features more easily, and, later, how to induce new features
Baselines

- First step: get a baseline
  - Baselines are very simple “straw man” procedures
  - Help determine how hard the task is
  - Help know what a “good” accuracy is

- Weak baseline: most frequent label classifier
  - Gives all test instances whatever label was most common in the training set
  - E.g. for spam filtering, might label everything as ham
  - Accuracy might be very high if the problem is skewed
  - E.g. calling everything “ham” gets 66%, so a classifier that gets 70% isn’t very good...

- For real research, usually use previous work as a (strong) baseline
Perceptrons and Logistic Regression
Linear Classifiers
Feature Vectors

Hello,
Do you want free printr cartriges? Why pay more when you can get them ABSOLUTELY FREE! Just

\[
\begin{align*}
x & \quad f(x) & \quad y \\
Hello, & \quad \{ # \text{free} : 2, \ \text{YOUR\_NAME} : 0, \ \text{MISSPELLED} : 2, \ \text{FROM\_FRIEND} : 0 \ldots \} & \quad \text{SPAM or} \\
& & \quad + \\
\& \quad \{ \text{PIXEL-7,12} : 1, \ \text{PIXEL-7,13} : 0 \ldots \ \text{NUM\_LOOPS} : 1 \ldots \} & \quad \text{“2”}
\end{align*}
\]
Some (Simplified) Biology

- Very loose inspiration: human neurons
Linear Classifiers

- Inputs are feature values
- Each feature has a weight
- Sum is the activation

\[ \text{activation}_w(x) = \sum_i w_i \cdot f_i(x) = w \cdot f(x) \]

- If the activation is:
  - Positive, output +1
  - Negative, output -1
Weights

- Binary case: compare features to a weight vector
- Learning: figure out the weight vector from examples

The dot product $w \cdot f$ is positive means the positive class.

$w$ for spam: "free" increase spam, your name decreases spam, etc.
Binary Decision Rule

- In the space of feature vectors
  - Examples are points
  - Any weight vector is a hyperplane
  - One side corresponds to $Y=+1$
  - Other corresponds to $Y=-1$

$$w$$

<table>
<thead>
<tr>
<th>Feature</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIAS</td>
<td>-3</td>
</tr>
<tr>
<td>free</td>
<td>4</td>
</tr>
<tr>
<td>money</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

$$f \cdot w = 0$$
Learning: Binary Perceptron

- Start with weights = 0
- For each training instance:
  - Classify with current weights
  - If correct (i.e., $y=y^*$), no change!
  - If wrong: adjust the weight vector

*Iterative algorithm parameter=weight (cf. naive bayes; parameter=probability)
Learning: Binary Perceptron

- Start with weights = 0
- For each training instance:
  - Classify with current weights
    \[
    y = \begin{cases} 
    +1 & \text{if } w \cdot f(x) \geq 0 \\
    -1 & \text{if } w \cdot f(x) < 0
    \end{cases}
    \]
  - If correct (i.e., y=y*), no change!
  - If wrong: adjust the weight vector by adding or subtracting the feature vector. Subtract if y* is -1.
    \[
    w = w + y^* \cdot f
    \]
Examples: Perceptron

- Separable Case

*iteration for every data points until no update
Multiclass Decision Rule

- If we have multiple classes:
  - A weight vector for each class:
    \[ w_y \]
  - Score (activation) of a class \( y \):
    \[ w_y \cdot f(x) \]
  - Prediction highest score wins
    \[ y = \arg \max_y \ w_y \cdot f(x) \]

Binary = multiclass where the negative class has weight zero
Learning: Multiclass Perceptron

- Start with all weights = 0
- Pick up training examples one by one
- Predict with current weights

\[ y = \text{arg max}_y \; w_y \cdot f(x) \]

- If correct, no change!
- If wrong: lower score of wrong answer, raise score of right answer

\[ w_y = w_y - f(x) \]
\[ w_{y^*} = w_{y^*} + f(x) \]
Example: Multiclass Perceptron

“win the vote”
“win the election”
“win the game”

*feature vector

[1] \( f_1 = (1,1,0,1,1,0,\ldots) \rightarrow \text{sport but wrong should be politics, change weight} \)

[2] \( f_2 = (1,1,0,0,1,0,\ldots) \rightarrow \text{correct, no change} \)

[3] \( f_3 = (1,1,1,0,1,0,\ldots) \rightarrow \text{wrong should be sports} \)

\[ w_{SPORTS} \]

BIAS : 1
win : 0
game : 0
vote : 0
the : 0
...  

[1] after one iteration \( w = (0,-1,0,-1,-1) \)

[2] decision for second sentence \( y=2 \)

[3] \( y=-2 \)

\[ w_{POLITICS} \]

BIAS : 0
win : 0
game : 0
vote : 0
the : 0
...  

[1] \( w = (1,1,0,1,1) \)

[2] \( y=3 \)

[3] \( y=3 \)

\[ w_{TECH} \]

BIAS : 0
win : 0
game : 0
vote : 0
the : 0
...  

[1] \( w = (0,0,0,0,0,0,0) \)

[2] \( y=0 \)

[3] \( y=0 \)
Properties of Perceptrons

- **Separability**: true if some parameters get the training set perfectly correct
- **Convergence**: if the training is separable, perceptron will eventually converge (binary case)
- **Mistake Bound**: the maximum number of mistakes (binary case) related to the margin or degree of separability

\[
\text{mistakes} < \frac{k}{\delta^2}
\]

*\(k\): # of features  
*\(\delta\): margin
Problems with the Perceptron

- **Noise**: if the data isn’t separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)

- **Mediocre generalization**: finds a “barely” separating solution

- **Overtraining**: test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting
Improving the Perceptron
Non-Separable Case: Deterministic Decision

Even the best linear boundary makes at least one mistake.
Non-Separable Case: Probabilistic Decision

*70% red, 30% blue
How to get probabilistic decisions?

- Perceptron scoring: \( z = w \cdot f(x) \)
  - If \( z = w \cdot f(x) \) very positive \( \rightarrow \) want probability going to 1
  - If \( z = w \cdot f(x) \) very negative \( \rightarrow \) want probability going to 0

- Sigmoid function

\[
\phi(z) = \frac{1}{1 + e^{-z}}
\]
Best \( w \)?

- **Maximum likelihood estimation:**

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)
\]

with:

\[
P(y^{(i)} = +1 | x^{(i)}; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

\[
P(y^{(i)} = -1 | x^{(i)}; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

= Logistic Regression
Separable Case: Deterministic Decision – Many Options

*perceptron no preference, converge either case
Separable Case: Probabilistic Decision – Clear Preference

*logistic regression prefer this due to optimizing objective function
Multiclass Logistic Regression

- **Recall Perceptron:**
  - A weight vector for each class: \( w_y \)
  - Score (activation) of a class \( y \): \( w_y \cdot f(x) \)
  - Prediction highest score wins: \( y = \text{arg max}_y \ w_y \cdot f(x) \)

- **How to make the scores into probabilities?**

\[
Z_1, Z_2, Z_3 \rightarrow \frac{e^{Z_1}}{e^{Z_1} + e^{Z_2} + e^{Z_3}}, \quad \frac{e^{Z_2}}{e^{Z_1} + e^{Z_2} + e^{Z_3}}, \quad \frac{e^{Z_3}}{e^{Z_1} + e^{Z_2} + e^{Z_3}}
\]

original activations

softmax activations
Best w?

- Maximum likelihood estimation:

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]

with:

\[
P(y^{(i)}|x^{(i)}; w) = \frac{e^{w_{y^{(i)}} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}
\]

= Multi-Class Logistic Regression
Optimization and Neural Nets
This Lecture

- Optimization
  - i.e., how do we solve:

\[
\max_w ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]
Hill Climbing

- Recall from CSPs lecture: simple, general idea
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- What’s particularly tricky when hill-climbing for multiclass logistic regression?
  - Optimization over a continuous space
    - Infinitely many neighbors!
  - How to do this efficiently?
1-D Optimization

- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
  - Then step in best direction

- Or, evaluate derivative:
  \[
  \frac{\partial g(w_0)}{\partial w} = \lim_{h \to 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}
  \]
  - Tells which direction to step into
2-D Optimization
Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: \( g(w_1, w_2) \)

- Updates:
  \[
  w_1 \leftarrow w_1 + \alpha \cdot \frac{\partial g}{\partial w_1}(w_1, w_2)
  \]
  \[
  w_2 \leftarrow w_2 + \alpha \cdot \frac{\partial g}{\partial w_2}(w_1, w_2)
  \]

- Updates in vector notation:
  \[
  w \leftarrow w + \alpha \cdot \nabla_w g(w)
  \]
  with: \( \nabla_w g(w) = \left[ \frac{\partial g}{\partial w_1}(w), \frac{\partial g}{\partial w_2}(w) \right] \)
Gradient Ascent

- Idea:
  - Start somewhere
  - Repeat: Take a step in the gradient direction
What is the Steepest Direction?

- First-Order Taylor Expansion:
  \[ g(w + \Delta) \approx g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2 \]

- Steepest Descent Direction:
  \[ \max_{\Delta : \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2 \]

- Recall:
  \[ \max_{\Delta : \|\Delta\| \leq \varepsilon} \Delta^T a \rightarrow \Delta = \varepsilon \frac{a}{\|a\|} \]

- Hence, solution:
  \[ \Delta = \varepsilon \frac{\nabla g}{\|\nabla g\|} \]

Gradient direction = steepest direction!

\[ \nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \end{bmatrix} \]
Gradient in n dimensions

\[ \nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\ \frac{\partial g}{\partial w_2} \\ \vdots \\ \frac{\partial g}{\partial w_n} \end{bmatrix} \]
Optimization Procedure: Gradient Ascent

- init $w$
- for iter = 1, 2, ...

$$w \leftarrow w + \alpha \times \nabla g(w)$$

- $\alpha$: learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
  - Crude rule of thumb: update changes $w$ about 0.1 – 1 %

*do not get affected by rescaling $g(w)$*
Batch Gradient Ascent on the Log Likelihood Objective

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)
\]

\[g(w)\]

- init \( w \)
- for iter = 1, 2, ...

\[w \leftarrow w + \alpha \sum_i \nabla \log P(y^{(i)} | x^{(i)}; w)\]
Stochastic Gradient Ascent on the Log Likelihood Objective

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]

**Observation:** once gradient on one training example has been computed, might as well incorporate before computing next one

- **init** \(w\)
- **for** iter = 1, 2, ...
  - **pick** random \(j\)

\[
w \leftarrow w + \alpha \times \nabla \log P(y^{(j)}|x^{(j)}; w)
\]
Mini-Batch Gradient Ascent on the Log Likelihood Objective

\[ \max_w \ ln(l(w)) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w) \]

**Observation:** gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- **init** \( w \)
- **for** iter = 1, 2, ...
  - pick random subset of training examples \( J \)
  \[ w \leftarrow w + \alpha \sum_{j \in J} \nabla \log P(y^{(j)} | x^{(j)}; w) \]
How about computing all the derivatives?

- We’ll talk about that once we covered neural networks, which are a generalization of logistic regression.
Multi-class Logistic Regression

- = special case of neural network

\[
P(y_1 | x; w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}
\]
\[
P(y_2 | x; w) = \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}
\]
\[
P(y_3 | x; w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}}
\]

\[
f_1(x) \rightarrow z_1 = f(x)^\top w_1 \text{ vectors}
\]
\[
f_2(x) \rightarrow z_2
\]
\[
f_3(x) \rightarrow z_3
\]
\[
\ldots
\]
\[
f_4(x)
\]
Deep Neural Network = Also learn the features!

\[ z_i^{(k)} = g\left( \sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)} \right) \]

\[ P(y_1|x; w) = \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\[ P(y_3|x; w) = \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

\( g = \) nonlinear activation function

\( x_1 \) \( x_2 \) \( x_3 \) \( x_L \)

\( z_1^{(1)} \) \( z_2^{(1)} \) \( z_3^{(1)} \) \( \ldots \) \( z_K^{(1)} \)

\( z_1^{(2)} \) \( z_2^{(2)} \) \( z_3^{(2)} \) \( \ldots \) \( z_K^{(2)} \)

\( z_1^{(n)} \) \( z_2^{(n)} \) \( z_3^{(n)} \) \( \ldots \) \( z_K^{(n)} \)

\( z_1^{(OUT)} \) \( z_2^{(OUT)} \) \( z_3^{(OUT)} \)
Common Activation Functions

Sigmoid Function

\[ g(z) = \frac{1}{1 + e^{-z}} \]
\[ g'(z) = g(z)(1 - g(z)) \]

Hyperbolic Tangent

\[ g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}} \]
\[ g'(z) = 1 - g(z)^2 \]

Rectified Linear Unit (ReLU)

\[ g(z) = \max(0, z) \]
\[ g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases} \]
Training the deep neural network is just like logistic regression:

$$\max_w \, ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

just \(w\) tends to be a much, much larger vector 😊

→ just run gradient ascent

+ stop when log likelihood of held-out data starts to decrease
Error Backpropagation in MLP

\[
\frac{\partial L}{\partial z_j} = \frac{dy_j}{dz_j} \frac{\partial L}{\partial y_j} \\
\frac{\partial L}{\partial y_i} = \sum_j \frac{dz_j}{dy_i} \frac{\partial L}{\partial z_j} = \sum_j w_{ij} \frac{\partial L}{\partial z_j} \frac{dy_j}{dz_j} \frac{\partial L}{\partial y_j} \\
\frac{\partial L}{\partial w_{ki}} = \sum_n \frac{\partial z_i^n}{\partial w_{ki}} \frac{dy_i^n}{dz_i^n} \frac{\partial L}{\partial y_i^n} = \sum_n \frac{\partial z_i^n}{\partial w_{ki}} \frac{dy_i^n}{dz_i^n} \sum_j w_{ij} \frac{dy_j^n}{dz_j^n} \frac{\partial L}{\partial y_j^n} 
\]

*error signal from previous layer

\( j \): number of nodes in previous layer; \( n \): number of training data

update \( w_{ki} \)
Neural Networks Properties

- **Theorem (Universal Function Approximators).** A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

- **Practical considerations**
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting
    - (hence early stopping!)
Universal Function Approximation Theorem*

- **In words:** Given any continuous function $f(x)$, if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate $f(x)$.

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”
Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”
Leshno and Schocken (1991) “Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”
Fun Neural Net Demo Site

- Demo-site:
  - http://playground.tensorflow.org/
How about computing all the derivatives?

- Derivatives tables:

\[
\begin{align*}
\frac{d}{dx}(a) &= 0 \\
\frac{d}{dx}(x) &= 1 \\
\frac{d}{dx}(au) &= a \frac{du}{dx} \\
\frac{d}{dx}(u + v - w) &= \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx} \\
\frac{d}{dx}(uv) &= u \frac{dv}{dx} + v \frac{du}{dx} \\
\frac{d}{dx}(\frac{u}{v}) &= \frac{1}{v} \frac{du}{dx} - \frac{u}{v^2} \frac{dv}{dx} \\
\frac{d}{dx}(u^n) &= nu^{n-1} \frac{du}{dx} \\
\frac{d}{dx}(\sqrt{u}) &= \frac{1}{2\sqrt{u}} \frac{du}{dx} \\
\frac{d}{dx}(\frac{1}{u}) &= -\frac{1}{u^2} \frac{du}{dx} \\
\frac{d}{dx}(\frac{1}{u^2}) &= -\frac{n}{u^{n+1}} \frac{du}{dx} \\
\frac{d}{dx}[f(u)] &= \frac{df}{du} \frac{du}{dx} \\
\frac{d}{dx}[\ln u] &= \frac{1}{u} \frac{du}{dx} \\
\frac{d}{dx}[\log_e u] &= \frac{1}{u} \frac{du}{dx} \\
\frac{d}{dx}[e^u] &= e^u \frac{du}{dx} \\
\frac{d}{dx}[a^u] &= a^u \ln a \frac{du}{dx} \\
\frac{d}{dx}[u^v] &= vu^{v-1} \frac{du}{dx} + ln u \ u^v \frac{dv}{dx} \\
\frac{d}{dx}[\sin u] &= \cos u \frac{du}{dx} \\
\frac{d}{dx}[\cos u] &= -\sin u \frac{du}{dx} \\
\frac{d}{dx}[\tan u] &= \sec^2 u \frac{du}{dx} \\
\frac{d}{dx}[\cot u] &= -\csc^2 u \frac{du}{dx} \\
\frac{d}{dx}[\sec u] &= \sec u \tan u \frac{du}{dx} \\
\frac{d}{dx}[\csc u] &= -\csc u \cot u \frac{du}{dx}
\end{align*}
\]
How about computing all the derivatives?

- But neural net \( f \) is never one of those?
  - No problem: CHAIN RULE:

\[
\text{If} \quad f(x) = g(h(x))
\]

\[
\text{Then} \quad f'(x) = g'(h(x))h'(x)
\]

→ Derivatives can be computed by following well-defined procedures
Automatic Differentiation

- Automatic differentiation software
  - e.g. Theano, TensorFlow, PyTorch, Chainer
  - Only need to program the function $g(x,y,w)$
  - Can automatically compute all derivatives w.r.t. all entries in $w$
  - This is typically done by caching info during forward computation pass of $f$, and then doing a backward pass = “backpropagation”
  - Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass
- Need to know this exists
Summary of Key Ideas

- Optimize probability of label given input
  \[
  \max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)
  \]

- Continuous optimization
  - Gradient ascent:
    - Compute steepest uphill direction = gradient (= just vector of partial derivatives)
    - Take step in the gradient direction
    - Repeat (until held-out data accuracy starts to drop = “early stopping”)

- Deep neural nets
  - Last layer = still logistic regression
  - Now also many more layers before this last layer
    - = computing the features
    - \(\rightarrow\) the features are learned rather than hand-designed
  - Universal function approximation theorem
    - If neural net is large enough
    - Then neural net can represent any continuous mapping from input to output with arbitrary accuracy
    - But remember: need to avoid overfitting / memorizing the training data \(\rightarrow\) early stopping!
  - Automatic differentiation gives the derivatives efficiently
Decision Trees
Today

- Formalizing Learning
  - Consistency
  - Simplicity

- Decision Trees
  - Expressiveness
  - Information Gain
  - Overfitting
Inductive Learning (Science)

- Simplest form: learn a function from examples
  - A target function: $g$
  - Examples: input-output pairs $(x, g(x))$
    - E.g. $x$ is an email and $g(x)$ is spam / ham
    - E.g. $x$ is a house and $g(x)$ is its selling price

- Problem:
  - Given a hypothesis space $H$
  - Given a training set of examples $x_i$
  - Find a hypothesis $h(x)$ such that $h \sim g$

- Includes:
  - Classification (outputs = class labels)
  - Regression (outputs = real numbers)

- How do perceptron and naïve Bayes fit in? $(H, h, g, \text{etc.})$

*BN cpt(parameter) space
Inductive Learning

- Curve fitting (regression, function approximation):
  - Consistency vs. simplicity
  - Ockham’s razor

*trade off
Consistency vs. Simplicity

- Fundamental tradeoff: bias vs. variance
  - Overfit

- Usually algorithms prefer consistency by default (why?)
  - Minimize training error

- Several ways to operationalize “simplicity”
  - Reduce the hypothesis space
    - Assume more: e.g. independence assumptions, as in naïve Bayes
    - Have fewer, better features / attributes: feature selection
    - Other structural limitations (decision lists vs trees)
  - Regularization
    - Smoothing: cautious use of small counts
    - Many other generalization parameters (e.g. pruning cutoffs)
    - Hypothesis space stays big, but harder to get to the outskirts
Decision Trees
## Reminder: Features

- **Features, aka attributes**
  - Sometimes: TYPE=French
  - Sometimes: $f_{\text{TYPE}=\text{French}}(x) = 1$

<table>
<thead>
<tr>
<th>Example</th>
<th>Attrs</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Alt</td>
<td>Bar</td>
</tr>
<tr>
<td>$X_1$</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>$X_2$</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>$X_3$</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>$X_4$</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>$X_5$</td>
<td>T</td>
<td>F</td>
</tr>
<tr>
<td>$X_6$</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>$X_7$</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>$X_8$</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>$X_9$</td>
<td>F</td>
<td>T</td>
</tr>
<tr>
<td>$X_{10}$</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>$X_{11}$</td>
<td>F</td>
<td>F</td>
</tr>
<tr>
<td>$X_{12}$</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>
Decision Trees

- Compact representation of a function:
  - Truth table
  - Conditional probability table
  - Regression values

- True function
  - Realizable: in $H$
Expressiveness of DTs

- Can express any function of the features

- However, we hope for compact trees

\[ P(C|A, B) \]

- However, we hope for compact trees
Comparison: Perceptrons

- What is the expressiveness of a perceptron over these features?

- For a perceptron, a feature’s contribution is either positive or negative
  - If you want one feature’s effect to depend on another, you have to add a new conjunction feature
  - E.g. adding “PATRONS=full ∧ WAIT = 60” allows a perceptron to model the interaction between the two atomic features

- DTs automatically conjoin features / attributes
  - Features can have different effects in different branches of the tree!

- Difference between modeling relative evidence weighting (NB) and complex evidence interaction (DTs)
  - Though if the interactions are too complex, may not find the DT greedily

<table>
<thead>
<tr>
<th>Example</th>
<th>Attributes</th>
<th>Target</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alt Bar Fri Hun Pat Price Rain Res Type Est</td>
<td>Target</td>
<td>Will Wait</td>
</tr>
<tr>
<td>X1</td>
<td>T F F T Some $$$ F T French 0–10</td>
<td>0–10</td>
</tr>
<tr>
<td>X2</td>
<td>T F F T Full $ F F Thai 30–60</td>
<td>30–60</td>
</tr>
</tbody>
</table>
Hypothesis Spaces

- **How many distinct decision trees with \( n \) Boolean attributes?**
  - = number of Boolean functions over \( n \) attributes
  - = number of distinct truth tables with \( 2^n \) rows
  - = \( 2^{2^n} \)
    - E.g., with 6 Boolean attributes, there are
      18,446,744,073,709,551,616 trees

- **How many trees of depth 1 (decision stumps)?**
  - = number of Boolean functions over 1 attribute
  - = number of truth tables with 2 rows, times \( n \)
  - = \( 4n \)
    - E.g. with 6 Boolean attributes, there are 24 decision stumps

- **More expressive hypothesis space:**
  - Increases chance that target function can be expressed (good)
  - Increases number of hypotheses consistent with training set (bad, why?)
  - Means we can get better predictions (lower bias)
  - But we may get worse predictions (higher variance)
Decision Tree Learning

- Aim: find a small tree consistent with the training examples
- Idea: (recursively) choose “most significant” attribute as root of (sub)tree

function DTL(examples, attributes, default) returns a decision tree

    if examples is empty then return default
    else if all examples have the same classification then return the classification
    else if attributes is empty then return MODE(examples)
    else
        best ← CHOOSE-ATTRIBUTE(attributes, examples)
        tree ← a new decision tree with root test best
        for each value $v_i$ of best do
            examples_i ← {elements of examples with best = $v_i$}
            subtree ← DTL(examples_i, attributes - best, MODE(examples))
            add a branch to tree with label $v_i$ and subtree subtree
        return tree
Choosing an Attribute

- Idea: a good attribute splits the examples into subsets that are (ideally) “all positive” or “all negative”

- So: we need a measure of how “good” a split is, even if the results aren’t perfectly separated out
Information answers questions
- The more uncertain about the answer initially, the more information in the answer
- Scale: bits
  - Answer to Boolean question with prior \(<1/2, 1/2>\)?
  - Answer to 4-way question with prior \(<1/4, 1/4, 1/4, 1/4>\)?
  - Answer to 4-way question with prior \(<0, 0, 0, 1>\)?
  - Answer to 3-way question with prior \(<1/2, 1/4, 1/4>\)?

A probability \(p\) is typical of:
- A uniform distribution of size \(1/p\)
- A code of length \(\log 1/p\)

*\(\sum_i p_i \log 1/p_i\)
Entropy

- General answer: if prior is \(<p_1, ..., p_n>\):
  - Information is the expected code length

\[
H(\langle p_1, \ldots, p_n \rangle) = E_p \log_2 \frac{1}{p_i}
\]

\[
= \sum_{i=1}^{n} -p_i \log_2 p_i
\]

- Also called the entropy of the distribution
  - More uniform = higher entropy
  - More values = higher entropy
  - More peaked = lower entropy
  - Rare values almost “don’t count”
Information Gain

- Back to decision trees!
- For each split, compare entropy before and after
  - Difference is the information gain
  - Problem: there’s more than one distribution after split!

\[
gain(A) = H(p/p+n) - \sum_k (p_k/n_k) H(p_k/p_n + n_k)
\]

\(p\): will wait, \(n\): no wait, \(k\): number of distribution after split

- Solution: use expected entropy, weighted by the number of examples
Next Step: Recurse

- Now we need to keep growing the tree!
- Two branches are done (why?)
- What to do under “full”?
  - See what examples are there...

```
<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>WillWait</th>
</tr>
</thead>
<tbody>
<tr>
<td>X₁</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>French</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>X₂</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>30–60</td>
<td>F</td>
</tr>
<tr>
<td>X₃</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>Some</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>0–10</td>
<td>T</td>
</tr>
<tr>
<td>X₄</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>10–30</td>
<td>T</td>
</tr>
<tr>
<td>X₅</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>Full</td>
<td>$$$</td>
<td>F</td>
<td>T</td>
<td>French</td>
<td>&gt;60</td>
<td>F</td>
</tr>
<tr>
<td>X₆</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$$</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Italian</td>
<td>0–10</td>
</tr>
<tr>
<td>X₇</td>
<td>F</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>None</td>
<td>$</td>
<td>T</td>
<td>F</td>
<td>F</td>
<td>0–10</td>
<td>F</td>
</tr>
<tr>
<td>X₈</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>T</td>
<td>Some</td>
<td>$$</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Thai</td>
<td>0–10</td>
</tr>
<tr>
<td>X₉</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>F</td>
<td>Burger</td>
<td>&gt;60</td>
</tr>
<tr>
<td>X₁₀</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>X₁₁</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>None</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Thai</td>
<td>0–10</td>
</tr>
<tr>
<td>X₁₂</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>Full</td>
<td>$</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>Burger</td>
<td>30–60</td>
</tr>
</tbody>
</table>
```
Example: Learned Tree

- Decision tree learned from these 12 examples:

- Substantially simpler than “true” tree
  - A more complex hypothesis isn't justified by data
  - Also: it’s reasonable, but wrong

*more generalizable, but sometimes wrong
### Example: Miles Per Gallon

<table>
<thead>
<tr>
<th>mpg</th>
<th>cylinders</th>
<th>displacement</th>
<th>horsepower</th>
<th>weight</th>
<th>acceleration</th>
<th>modelyear</th>
<th>maker</th>
</tr>
</thead>
<tbody>
<tr>
<td>good</td>
<td>4</td>
<td>low</td>
<td>low</td>
<td>low</td>
<td>high</td>
<td>75to78</td>
<td>asia</td>
</tr>
<tr>
<td>bad</td>
<td>6</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
<td>70to74</td>
<td>america</td>
</tr>
<tr>
<td>bad</td>
<td>4</td>
<td>medium</td>
<td>medium</td>
<td>low</td>
<td>75to78</td>
<td>asia</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>8</td>
<td>high</td>
<td>high</td>
<td>low</td>
<td>70to74</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>6</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
<td>70to74</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>4</td>
<td>low</td>
<td>medium</td>
<td>low</td>
<td>70to74</td>
<td>asia</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>4</td>
<td>low</td>
<td>medium</td>
<td>low</td>
<td>70to74</td>
<td>asia</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>8</td>
<td>high</td>
<td>high</td>
<td>low</td>
<td>75to78</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>8</td>
<td>high</td>
<td>medium</td>
<td>low</td>
<td>75to78</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>4</td>
<td>low</td>
<td>low</td>
<td>medium</td>
<td>79to83</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>4</td>
<td>medium</td>
<td>low</td>
<td>medium</td>
<td>79to83</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>4</td>
<td>low</td>
<td>low</td>
<td>medium</td>
<td>79to83</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>8</td>
<td>high</td>
<td>medium</td>
<td>low</td>
<td>70to74</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>good</td>
<td>4</td>
<td>low</td>
<td>medium</td>
<td>low</td>
<td>70to74</td>
<td>america</td>
<td></td>
</tr>
<tr>
<td>bad</td>
<td>5</td>
<td>medium</td>
<td>medium</td>
<td>medium</td>
<td>75to78</td>
<td>europe</td>
<td></td>
</tr>
</tbody>
</table>
Find the First Split

- Look at information gain for each attribute
- Note that each attribute is correlated with the target!
- What do we split on?
Result: Decision Stump

mpg values: bad  good

root
22 18
pchance = 0.001

cylinders = 3
0 0
Predict bad

cylinders = 4
4 17
Predict good

cylinders = 5
1 0
Predict bad

cylinders = 6
8 0
Predict bad

cylinders = 8
9 1
Predict bad
*p-chance: correlation btw attribute and target: small is better; If p-chance is bigger, then prune \( \rightarrow \) reduce overfitting
Reminder: Overfitting

- **Overfitting:**
  - When you stop modeling the patterns in the training data (which generalize)
  - And start modeling the noise (which doesn’t)

- **We had this before:**
  - Naïve Bayes: needed to smooth
  - Perceptron: early stopping
The test set error is much worse than the training set error…

…why?

*overfitting -> need pruning
Consider this split
Significance of a Split

- Starting with:
  - Three cars with 4 cylinders, from Asia, with medium HP
  - 2 bad MPG
  - 1 good MPG

- What do we expect from a three-way split?
  - Maybe each example in its own subset?
  - Maybe just what we saw in the last slide?

- Probably shouldn’t split if the counts are so small they could be due to chance

- A chi-squared test can tell us how likely it is that deviations from a perfect split are due to chance*

- Each split will have a significance value, $p_{\text{CHANCE}}$
**Pruning:**

- **Build the full decision tree**
- **Begin at the bottom of the tree**
- **Delete splits in which** $p_{\text{CHANCE}} > \text{MaxP}_{\text{CHANCE}}$
- **Continue working upward until** there are no more prunable nodes
- **Note:** some chance nodes may not get pruned because they were “redeemed” later

*top-down pruning while building tree is bad here because root will be pruned in xor*
Pruning example

- With MaxP_{CHANCE} = 0.1:

  *after applying pruning*

Note the improved test set accuracy compared with the unpruned tree

<table>
<thead>
<tr>
<th></th>
<th>Num Errors</th>
<th>Set Size</th>
<th>Percent Wrong</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Set</td>
<td>5</td>
<td>40</td>
<td>12.50</td>
</tr>
<tr>
<td>Test Set</td>
<td>56</td>
<td>352</td>
<td>15.91</td>
</tr>
</tbody>
</table>

*but training error may be increased*
Regularization

- $\text{MaxP}_{\text{CHANCE}}$ is a regularization parameter
- Generally, set it using held-out data (as usual)
Two Ways of Controlling Overfitting

- **Limit the hypothesis space**
  - E.g. limit the max depth of trees
  - Easier to analyze

- **Regularize the hypothesis selection**
  - E.g. chance cutoff
  - Disprefer most of the hypotheses unless data is clear
  - Usually done in practice
Kernels and Clustering
Non-Separable Data

*need arbitrary decision boundary
Case-Based Reasoning

- Classification from similarity
  - Case-based reasoning
  - Predict an instance’s label using similar instances

- Nearest-neighbor classification
  - 1-NN: copy the label of the most similar data point
  - K-NN: vote the k nearest neighbors (need a weighting scheme)
  - Key issue: how to define similarity
  - Trade-offs: Small k gives relevant neighbors, Large k gives smoother functions

http://www.cs.cmu.edu/~zhuxj/courseproject/knndemo/KNN.html
Parametric / Non-Parametric

- **Parametric models:**
  - Fixed set of parameters
  - More data means better settings

- **Non-parametric models:**
  - Complexity of the classifier increases with data
  - Better in the limit, often worse in the non-limit

- (K)NN is **non-parametric**
Nearest-Neighbor Classification

- Nearest neighbor for digits:
  - Take new image
  - Compare to all training images
  - Assign based on closest example

- Encoding: image is vector of intensities:
  \[ \mathbf{1} = \langle 0.0 \ 0.0 \ 0.3 \ 0.8 \ 0.7 \ 0.1 \ldots 0.0 \rangle \]

- What’s the similarity function?
  - Dot product of two images vectors?
    \[ \text{sim}(x, x') = \mathbf{x} \cdot \mathbf{x}' = \sum_i x_i x'_i \]
  - Usually normalize vectors so \( ||\mathbf{x}|| = 1 \)
  - min = 0 (when?), max = 1 (when?)

\*\( ||\mathbf{x} - \mathbf{x}'||^2 \) euclidean distance small
\( \rightarrow \) dot product bigger (angle small)
Basic Similarity

- Many similarities based on feature dot products:

\[
\text{sim}(x, x') = f(x) \cdot f(x') = \sum_i f_i(x)f_i(x')
\]

- If features are just the pixels:

\[
\text{sim}(x, x') = x \cdot x' = \sum_i x_ix'_i
\]

- Note: not all similarities are of this form
Better similarity functions use knowledge about vision

Example: invariant metrics:
- Similarities are invariant under certain transformations
- Rotation, scaling, translation, stroke-thickness...
- E.g:
  - $16 \times 16 = 256$ pixels; a point in $256$-dim space
  - These points have small similarity in $\mathbb{R}^{256}$ (why?)
  - How can we incorporate such invariances into our similarities?
Rotation Invariant Metrics

- Each example is now a curve in $\mathbb{R}^{256}$
- Rotation invariant similarity:
  \[ s' = \max s( r(3), r(3) ) \]
- E.g. highest similarity between images’ rotation lines

*2) can do data augmentation too
A Tale of Two Approaches...

- Nearest neighbor-like approaches
  - Can use fancy similarity functions
  - Don’t actually get to do explicit learning

- Perceptron-like approaches
  - Explicit training to reduce empirical error
  - Can’t use fancy similarity, only linear
  - Or can they? Let’s find out!

*use kernel
Kernelization
Perceptron Weights

- What is the final value of a weight $w_y$ of a perceptron?
  - Can it be any real vector?
  - No! It’s built by adding up inputs.

$$w_y = 0 + f(x_1) - f(x_5) + \ldots$$

$$w_y = \sum_i \alpha_{i,y} f(x_i)$$

- Can reconstruct weight vectors (the primal representation) from update counts (the dual representation)

$$\alpha_y = \langle \alpha_{1,y}, \alpha_{2,y}, \ldots, \alpha_{n,y} \rangle$$
Dual Perceptron

- How to classify a new example \( x \)?

\[
\text{score}(y, x) = w_y \cdot f(x) = \left( \sum_i \alpha_{i,y} f(x_i) \right) \cdot f(x) = \sum_i \alpha_{i,y} (f(x_i) \cdot f(x)) = \sum_i \alpha_{i,y} K(x_i, x)
\]

- If someone tells us the value of \( K \) for each pair of examples, then we never need to build the weight vectors (or the feature vectors)!

\*inference

\*x, i training data, x test data
Dual Perceptron

- Start with zero counts (alpha)
- Pick up training instances one by one
- Try to classify $x_n$,

$$y = \arg \max_y \sum_i \alpha_{i,y} K(x_i, x_n)$$

- If correct, no change!
- If wrong: lower count of wrong class (for this instance), raise count of right class (for this instance)

$$\alpha_{y,n} = \alpha_{y,n} - 1$$  
$$\alpha_{y^*,n} = \alpha_{y^*,n} + 1$$

$$w_y = w_y - f(x_n)$$  
$$w_{y^*} = w_{y^*} + f(x_n)$$

*training

* dual perceptron

* primal perceptron
Kernelized Perceptron

- If we had a black box (kernel) \( K \) that told us the dot product of two examples \( x \) and \( x' \):
  - Could work entirely with the dual representation
  - No need to ever take dot products ("kernel trick")

\[
\text{score}(y, x) = w_y \cdot f(x) = \sum_{i} \alpha_{i,y} K(x_i, x)
\]

- Like nearest neighbor – work with black-box similarities
- Downside: slow if many examples get nonzero alpha

*work with features vs. work with data
Kernels: Who Cares?

- So far: a very strange way of doing a very simple calculation

- “Kernel trick”: we can substitute any* similarity function in place of the dot product

- Lets us learn new kinds of hypotheses

*positive semi-definite gram (kernel) matrix – Mercer theorem

* Fine print: if your kernel doesn’t satisfy certain technical requirements, lots of proofs break. E.g. convergence, mistake bounds. In practice, illegal kernels sometimes work (but not always).
Non-Linear Separators

- Data that is linearly separable works out great for linear decision rules:

- But what are we going to do if the dataset is just too hard?

- How about... mapping data to a higher-dimensional space:

This and next few slides adapted from Ray Mooney, UT
General idea: the original feature space can always be mapped to some higher-dimensional feature space where the training set is separable:
Some Kernels

- Kernels implicitly map original vectors to higher dimensional spaces, take the dot product there, and hand the result back

- Linear kernel:
  \[ K(x, x') = x' \cdot x' = \sum_i x_i x_i' \]

- Quadratic kernel:
  \[ K(x, x') = (x \cdot x' + 1)^2 \]
  \[ = \sum_{i,j} x_i x_j x_i' x_j' + 2 \sum_i x_i x_i' + 1 \]

- RBF: infinite dimensional representation
  \[ K(x, x') = \exp(-||x - x'||^2) \]

- Discrete kernels: e.g. string kernels
  \[ 2 \alpha^2 \]

*degree d polynomial
*gaussian kernel
*string similarity
Why Kernels?

- Can’t you just add these features on your own (e.g. add all pairs of features instead of using the quadratic kernel)?
  - Yes, in principle, just compute them
  - No need to modify any algorithms
  - But, number of features can get large (or infinite)
  - Some kernels not as usefully thought of in their expanded representation, e.g. RBF kernels

- Kernels let us compute with these features implicitly
  - Example: implicit dot product in quadratic kernel takes much less space and time per dot product
  - Of course, there’s the cost for using the pure dual algorithms: you need to compute the similarity to every training datum

*SVM uses kernel trick (non-parametric)*
Recap: Classification

Classification systems:
- Supervised learning
- Make a prediction given evidence
- We’ve seen several methods for this
- Useful when you have labeled data