

Lecture 11

August 25, 2016

Clustering, Classifiers, Information Theory



Reminder: start the recording

Announcements

- Project 4: Find all DNA targets
 - Due next Thursday 9/1 at 11:45 p.m.
- Project 5: Naïve Bayesian language classifier
 - Due at 11:45 p.m. on Thursday Sept. 8th

<http://courses.washington.edu/ling473/Project5.pdf>
- Reminder: Writing Assignment

<http://courses.washington.edu/ling473/writing-assignment.html>

 - Due at 11:45 p.m. on Tuesday, Sept. 6th
- Project 6 (*optional*): Edit Distance: Wikipedia Articles
 - If you want my feedback, submit by 9/12/2015

<http://courses.washington.edu/ling473/Project6.pdf>

Project 3

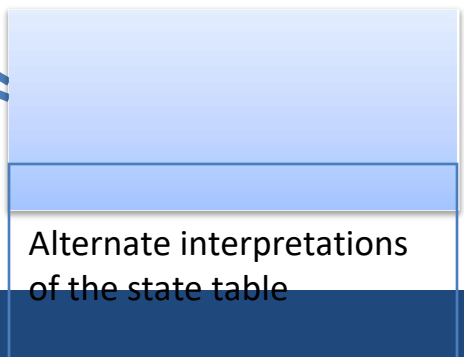
- Correct output:

/opt/dropbox/16-17/473/project3/fsm-output.html

คู้ แะง่ ชั้ น ต่า ง กั คั ม เะง กน
เะา เะงบ ใ้ คุรู่ หนง แลว พด ขน
เะอ หน มา ทราย ขน มา ใหม่
คย
ย น ดึ ทึ่ ใ้ รั รั จั ก คณ
จัน เะเห็น เะา ตาม เะอ แะ ใ้ ทก ที่ เะ็น เะา ตาม ตั ว ตลอด เะย
น ก เล็ก มี หนา อ ก สี แดง
เะา ของ โรง งาน มอ ง หา ของ โรง งาน ใหม่
ทึ่ ตัง
เะา ของ เะา ส่น เะา เะอ ตอ ง ลง มา จั ก ทึ่ เะง
จด สด ยอด ของ ความ รั ทาง เะศ (or เะศ)
ั สก

- Sample solutions:

/opt/dropbox/16-17/473/project3/project3.cs



<http://courses.washington.edu/ling473/project-3.html>


```
Dictionary<int, Func<Char, StateResult>> m = new Dictionary<int, Func<Char, StateResult>>()
{
    { 0, ch => {
        if (V1.Contains(ch))
            return new StateResult(1, ch.ToString());
        if (C1.Contains(ch))
            return new StateResult(2, ch.ToString());
        throw new Exception();
    },
    { 1, ch => {
        if (C1.Contains(ch))
            return new StateResult(2, ch.ToString());
        throw new Exception();
    },
    // ...
};

public String SyllableBreak(String s_in)
{
    StringBuilder sb = new StringBuilder();
    int state = 0;
    for (int i = 0; i < s_in.Length; i++)
    {
        StateResult sr = m[state](s_in[i]);
        sb.Append(sr.output);
        state = sr.newstate;
    }
    return sb.ToString();
}
```

Lambda function FST

```
{ 4, ch => {  
    if (T.Contains(ch))  
        return new StateResult(5, ch.ToString());  
    if (V3.Contains(ch))  
        return new StateResult(6, ch.ToString());  
    if (C3.Contains(ch))  
        // was --> state 9  
        return new StateResult(0, String.Format("{0} ", ch.ToString()));  
    if (V1.Contains(ch))  
        // was --> state 7  
        return new StateResult(1, String.Format(" {0}", ch.ToString()));  
    if (C1.Contains(ch))  
        // was --> state 8  
        return new StateResult(2, String.Format(" {0}", ch.ToString()));  
    throw new Exception();  
}  
},
```

Today

- CFG Parsing
- Clustering
- Classifiers
- Overview of Information Theory

Context-free grammar (CFG)

- or, “Phrase structure grammar”
- or, “Backus-Naur Form” (BNF)
- The most common way of modeling constituency
- The idea of basing a grammar on constituent structure dates back to Wilhem Wundt (1890), but not formalized until Chomsky (1956), and, independently, by Backus (1959).
- This is a particular type of **formal grammar**.
- Before we look at CFGs grammar, let’s put them in “context...”

Defining grammars

- A *grammar* is defined by the tuple

$$G = \langle V, \Sigma, S, R \rangle$$

- V is a finite set of *variables* or *preterminals*
- Σ is a finite set of symbols, called *terminals*
- S is in V and is called the *start symbol*
- R is a finite set of *productions*, which are *rules* of the form

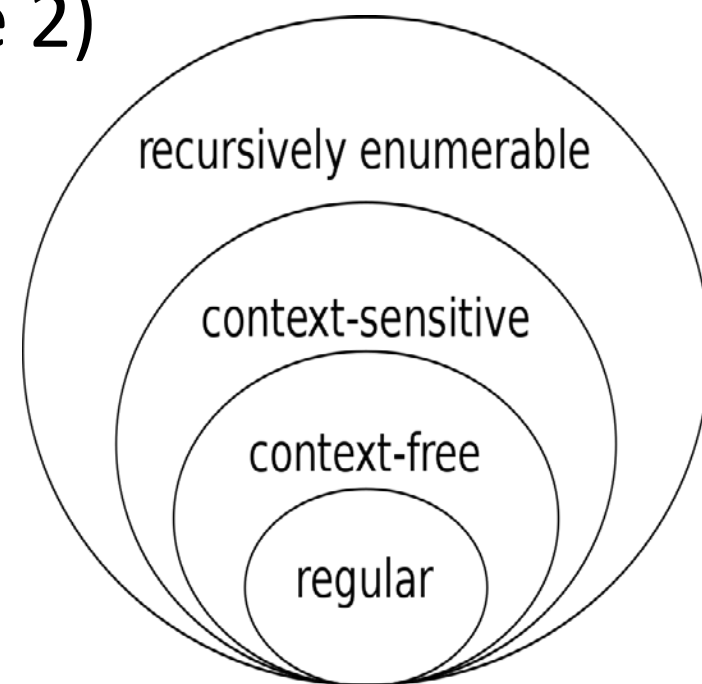
$$\alpha \rightarrow \beta$$

where α and β are strings consisting of terminals and variables.

Types of formal grammars

- Unrestricted, recursively enumerable (type 0)
- Context-sensitive (type 1)
- Context-free grammars (type 2)
- Regular grammars (type 3)

The Chomsky hierarchy



Types of formal grammars

- Unrestricted, recursively enumerable (type 0)

$$\alpha\alpha \rightarrow \beta\beta$$

$\alpha\alpha$ and $\beta\beta$ are any string of terminals and non-terminals

- Context-sensitive (type 1)

$$\alpha\alpha XX\beta\beta \rightarrow \alpha\alpha\alpha\alpha\beta\beta$$

XX is a nonterminal; $\alpha\alpha, \beta\beta, \alpha\alpha$ are any string of terminals and non-terminals; $\alpha\alpha$ may not be empty.

- Context-free grammars (type 2)

$$XX \rightarrow \alpha\alpha$$

XX is a nonterminal; $\alpha\alpha$ are any string of terminals and non-terminals

- Regular grammars (type 3)

$$XX \rightarrow \alpha\alpha YY$$

(i.e. a *right* regular grammar)

XX, YY are nonterminals; $\alpha\alpha$ is any string of terminals; YY may be absent.

increasing restriction

Context-free grammar

(type 2)

- What if we allow our production rule in a linear grammar to have more than one nonterminal?

...we get the most important type of grammar studied in linguistics: the **context-free grammar** (CFG)

Context free grammar

(type 2)

$$GG = (VV, \Sigma, RR, SS)$$

$VV = \{ \text{pre-terminals } vv_0, vv_1, \dots \}$

$\Sigma = \{ \text{terminals } ww_0, ww_1, \dots \}$

$RR = \{ \text{rules } r_i: vv \rightarrow \alpha\alpha \}$

$SS = \text{start symbol, } SS \in VV$

$\alpha\alpha$: sequence of terminals and pre-terminals (or \emptyset)

LL : the language generated by GG

Context-sensitive grammars

(type 1)

- No rule can make a string shorter
- Can be accepted by a 'linear bounded automaton' (LBA), a nondeterministic Turing machine which uses space $O(n)$

Unrestricted grammar

(type 0)

- The most general class
- Recognized by Turing machine
- Generate recursively-enumerable languages

Example CFG

$GG = (VV, \Sigma, RR, \$S$
 $VV = \{ SS, NNNN, NNOONN, VVNN, DDDDDD, NNNNNNnn, VVDDrrVV, AANNAA \}$
 $\Sigma = \{ DD\textit{t}ttDD, DD\textit{t}ttt, tt, DD\textit{t}DD, mmttnn, VVNNNNbb, ffffttff\textit{t}DD, mmDDttff, ttnniiffNN\textit{i}iDD, rrDD\textit{t}ti, iiNNDD\textit{t}t \}$
 $SS = SS$

$R = \{$

$S \rightarrow NP VP$	$Det \rightarrow that \mid this \mid a \mid the$
$S \rightarrow Aux NP VP$	$Noun \rightarrow book \mid flight \mid meal \mid man$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid read$
$NP \rightarrow Det NOM$	$Aux \rightarrow does$
$NOM \rightarrow Noun$	
$NOM \rightarrow Noun NOM$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb NP$	

$\}$

Grammar rewrite rules

S --> NP VP

--> Det NOM VP

--> The NOM VP

--> The Noun VP

--> The man VP

--> The man Verb NP

--> The man read NP

--> The man read Det NOM

--> The man read this NOM

--> The man read this Noun

--> The man read this book

S \rightarrow NP VP

S \rightarrow Aux NP VP

S \rightarrow VP

NP \rightarrow Det NOM

NOM \rightarrow Noun

NOM \rightarrow Noun NOM

VP \rightarrow Verb

VP \rightarrow Verb NP

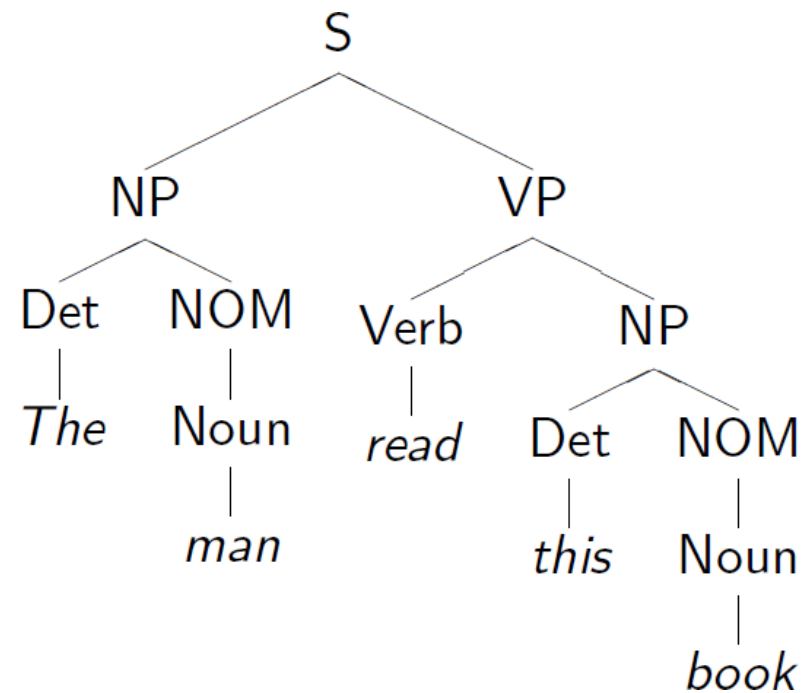
Det \rightarrow *that* | *this* | *a* | *the*

Noun \rightarrow *book* | *flight* | *meal* | *man*

Verb \rightarrow *book* | *include* | *read*

Aux \rightarrow *does*

Parse tree



PCFG

- Probabilistic context-free grammar
- Adds probabilities to each rule
- Each distinct left-hand-side gets a probability mass 1.0
- Rule weights can be estimated from corpora

CFGs can express recursion

- Example of seemingly endless recursion of embedded prepositional phrases:

PP → Prep NP

NP → Noun PP

[S The mailman ate his [NP lunch [PP with his friend [PP from the cleaning staff [PP of the building [PP at the intersection [PP on the north end [PP of town]]]]]]]].

Most programming languages are type-2 grammars (CFGs)

Chomsky Normal Form

- Any CFG can be converted into a form where all rules are of the form:

$$XX \rightarrow YYY$$

$$XX \rightarrow t$$

$$S \rightarrow \lambda$$

(S is the only terminal that can go to the empty string)

Convert $WW \rightarrow XXYYttYY$ to Chomsky Normal Form

CNF conversion

- Steps:
 1. Make S non-recursive
 2. Eliminate $\lambda\lambda$ (except $SS \rightarrow \lambda\lambda$)
 3. Eliminate all chain rules
 4. Remove unused symbols

Convert the following
grammar to Chomsky
Normal Form:

$SS \rightarrow AASSAA$

$SS \rightarrow ttaa$

$AA \rightarrow aa$

$AA \rightarrow SS$

$aa \rightarrow VV$

$aa \rightarrow \lambda\lambda$

CNF conversion

$SS \rightarrow AASSAA$
 $SS \rightarrow ttaa$
 $AA \rightarrow aa$
 $AA \rightarrow SS$
 $aa \rightarrow VV$
 $aa \rightarrow \lambda\lambda$

$SS \rightarrow AASSAA$
 $SS \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow SS$
 $aa \rightarrow VV$
 $aa \rightarrow \lambda\lambda$
 $UU_{aa} \rightarrow tt$

$SS \rightarrow AA\textcolor{red}{XX}$
 $SS \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow SS$
 $aa \rightarrow VV$
 $aa \rightarrow \lambda\lambda$
 $UU_{aa} \rightarrow tt$
 $\textcolor{red}{XX} \rightarrow \textcolor{red}{SS}AA$

$\textcolor{red}{SS} \rightarrow \textcolor{red}{SS}'$
 $\textcolor{red}{SS}' \rightarrow AA\textcolor{red}{XX}$
 $\textcolor{red}{SS}' \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow \textcolor{red}{SS}'$
 $aa \rightarrow VV$
 $aa \rightarrow \lambda\lambda$
 $UU_{aa} \rightarrow tt$
 $\textcolor{red}{XX} \rightarrow \textcolor{red}{SS}'AA$

$SS \rightarrow SS'$
 $SS' \rightarrow AA\textcolor{red}{XX}$
 $SS' \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow SS'$
 $aa \rightarrow VV$

 $UU_{aa} \rightarrow tt$
 $\textcolor{red}{XX} \rightarrow \textcolor{red}{SS}'AA$
 $\textcolor{red}{XX} \rightarrow \textcolor{red}{SS}'$
 $\textcolor{red}{SS}' \rightarrow \textcolor{red}{XX}$

$SS \rightarrow SS'$
 $SS' \rightarrow AAXX$
 $SS' \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow SS'$
 $aa \rightarrow VV$
 $UU_{aa} \rightarrow tt$
 $XX \rightarrow SS'AA$
 $XX \rightarrow SS'$
 $SS' \rightarrow XX$

$SS \rightarrow XX$
 $XX \rightarrow AAXX$
 $XX \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow XX$
 $aa \rightarrow VV$
 $UU_{aa} \rightarrow tt$
 $XX \rightarrow XXAA$
 ~~$XX \rightarrow XX$~~
 ~~$SS' \rightarrow XX$~~

~~$SS \rightarrow XX$~~
 $XX \rightarrow AAXX$
 $XX \rightarrow UU_{aa}aa$
 $AA \rightarrow aa$
 $AA \rightarrow XX$
 $aa \rightarrow VV$
 $UU_{aa} \rightarrow tt$
 $XX \rightarrow XXAA$
 $SS \rightarrow AAXX$
 $SS \rightarrow UU_{aa}aa$
 $SS \rightarrow XXAA$

$XX \rightarrow AAXX$
 $XX \rightarrow UU_{aa}aa$
 $AA \rightarrow VV$
 $AA \rightarrow XX$
 $aa \rightarrow VV$
 $UU_{aa} \rightarrow tt$
 $XX \rightarrow XXAA$
 $SS \rightarrow AAXX$
 $SS \rightarrow UU_{aa}aa$
 $SS \rightarrow XXAA$

$XX \rightarrow AAXX$
 $XX \rightarrow UU_{aa}aa$
 $AA \rightarrow VV$
 ~~$AA \rightarrow XX$~~
 $aa \rightarrow VV$
 $UU_{aa} \rightarrow tt$
 $XX \rightarrow XXAA$
 $SS \rightarrow AAXX$
 $SS \rightarrow UU_{aa}aa$
 $SS \rightarrow XXAA$
 $AA \rightarrow AAXX$
 $AA \rightarrow UU_{aa}aa$
 $AA \rightarrow XXAA$


all done

Parsing context-free grammars (type 2)

- CFGs are widely used to represent surface syntax in natural languages
- Space complexity:
 - you'll need at least one stack
 - space use will depend on the amount of recursion in the input
- Time complexity
 - Generally $O(n^3)$

Parsing

- The opposite of generation: find the structure from a string
- Essentially a search problem
- Find all structure that match an input string
- Two approaches
 - Bottom-up
 - Top-down



This refers to the parse tree, not the search tree. So either method can be done breadth-first or depth-first

Recognizer v. parser

- **Recognizer** (acceptor) is a program that determines whether a sentence is accepted by the grammar or not
- A **parser** determines this as well, and if the sentence is accepted, it also returns the structural configuration(s) of grammar rules for the sentence
- Some parsing systems may also produce compositional semantics

Soundness and completeness

- Correctness: a parser is **sound** if every parse it returns is correct
- A parser **terminates** if it is guaranteed not to enter an infinite loop
- A parser is **complete** for grammar GG and sentence SS if it is sound, produces every possible parse for SS , and terminates
- Often, we settle for sound but incomplete parsers
 - probabilistic parsers may be able to return the *bb*-best parses

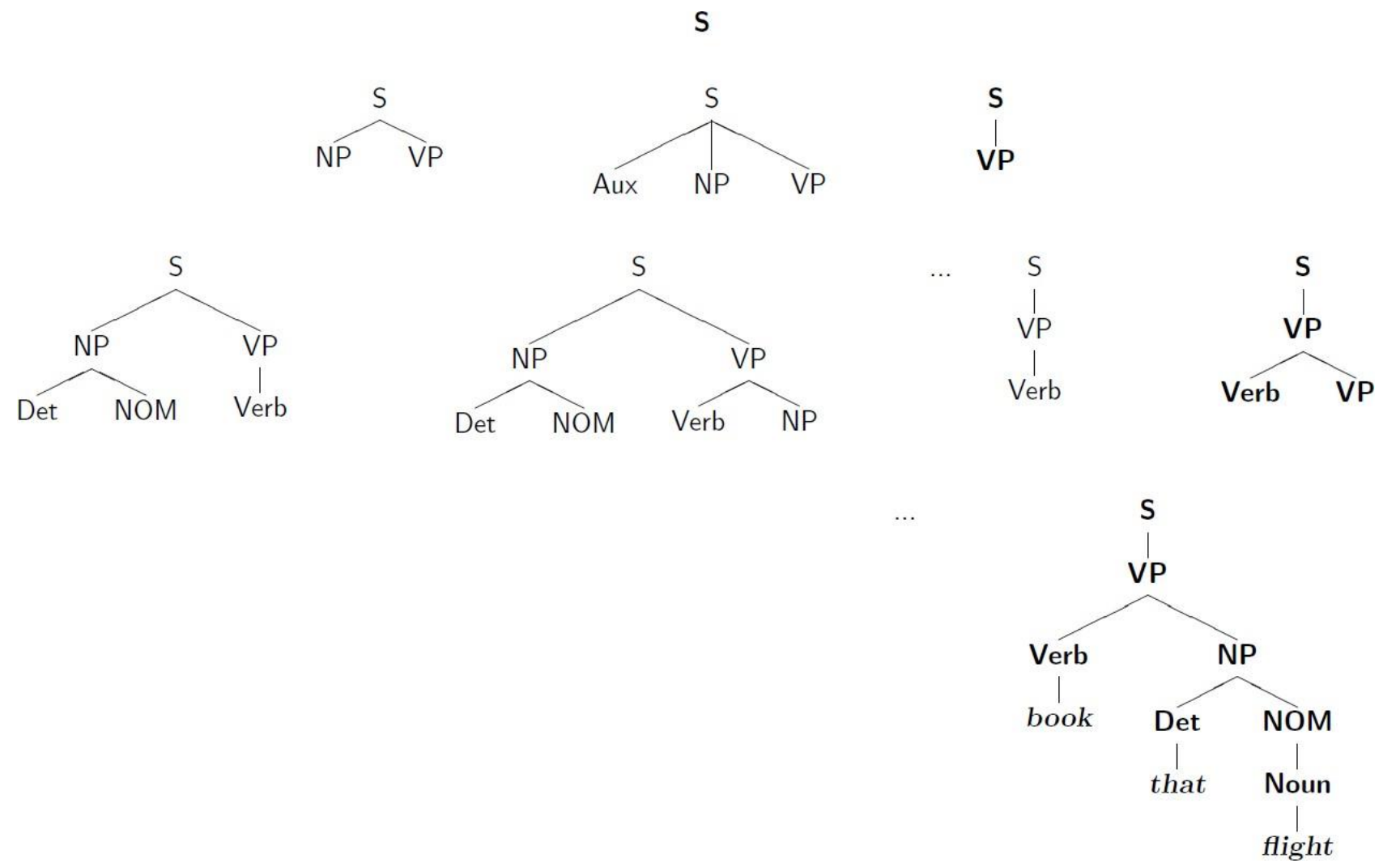
Top-down parsing

- Create a list of goal constituents
- Rewrite goals by matching a goal on the list with the left-hand-side of a rule
- Replace with the right-hand-side
- If there is more than one match to the LHS, try different rules (breadth-first or depth-first)

example

$S \rightarrow NP VP$	$Det \rightarrow that \mid this \mid a \mid the$
$S \rightarrow Aux NP VP$	$Noun \rightarrow book \mid flight \mid meal \mid man$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid read$
$NP \rightarrow Det NOM$	$Aux \rightarrow does$
$NOM \rightarrow Noun$	
$NOM \rightarrow Noun NOM$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb NP$	

Book that flight.



Problems with top-down parsing

- Left-recursive rules lead to infinite recursion
$$NNNN \rightarrow NNNN NNNN$$
- Poor performance when there are many matches for an LHS
 - If there are many rules for S, there's no way to eliminate irrelevant ones. In other words, it does the useless work of expanding things that there is no evidence for
- Doesn't work at the terminals (lexemes)
- Can't make use of common substructure

Bottom-up parsing

- Bottom-up parsing is data-directed
- Start with the string to be parsed
- Match right-hand-sides, condense to LHS
 - Still need to choose when there are multiple possible matches for the RHS
 - Can use breadth-first or depth-first search
- Parsing is complete when all you have left is the start symbol

example

$S \rightarrow NP VP$	$Det \rightarrow that \mid this \mid a \mid the$
$S \rightarrow Aux NP VP$	$Noun \rightarrow book \mid flight \mid meal \mid man$
$S \rightarrow VP$	$Verb \rightarrow book \mid include \mid read$
$NP \rightarrow Det NOM$	$Aux \rightarrow does$
$NOM \rightarrow Noun$	
$NOM \rightarrow Noun NOM$	
$VP \rightarrow Verb$	
$VP \rightarrow Verb NP$	

Book that flight.

Shift-reduce parsing

Stack	Input remaining	Action
()	Book that flight	shift
(Book)	that flight	reduce, Verb \rightarrow book, (Choice #1 of 2)
(Verb)	that flight	shift
(Verb that)	flight	reduce, Det \rightarrow that
(Verb Det)	flight	shift
(Verb Det flight)		reduce, Noun \rightarrow flight
(Verb Det Noun)		reduce, NOM \rightarrow Noun
(Verb Det NOM)		reduce, NP \rightarrow Det NOM
(Verb NP)		reduce, VP \rightarrow Verb NP
(Verb)		reduce, S \rightarrow V
(S)		SUCCESS!

Ambiguity may lead to the need for backtracking.

Shift-reduce parser

- Start with the sentence in an input buffer
 - Shift: push the next input symbol onto the stack
 - Reduce: if a RHS matches the top elements of the stack, pop those elements off and push the LHS
- If either shift or reduce are possible, choose arbitrarily
- If you end up with only the start symbol on the stack, you have a parse
- Otherwise, you can backtrack

Shift-reduce parser

- In the top-down parser, the main decision was which production rule to pick
 - In a bottom-up shift-reduce parser, the decisions are:
 - Should we shift, or reduce
 - If we reduce, then by which rule
- Both of these decisions can be revisited when backtracking

Problems with bottom-up parsing

- No obvious way to generate structures that generate empty surface
- Lexical ambiguity can explode the search space
- Useless constituents can be built locally

Top-down and bottom-up parsers can both be extremely inefficient on real-world NLP parsing problems. Complexity may approach $OO(bb^{nn})$ in the sentence length.

Parsing is hard

- Left-recursive structures must be found, not predicted
- Empty categories must be predicted, not found
- When backtracking, don't redo any work
- Get linguists on board to think about computability from the start

clustering, classifiers, information theory

Labeled Data

- In the POS tagging lecture we talked about **annotating** corpora
 - automatically
 - manually
 - hybrid annotation
- Once annotated, the corpus represents **labeled data** that can be used for training

Machine learning

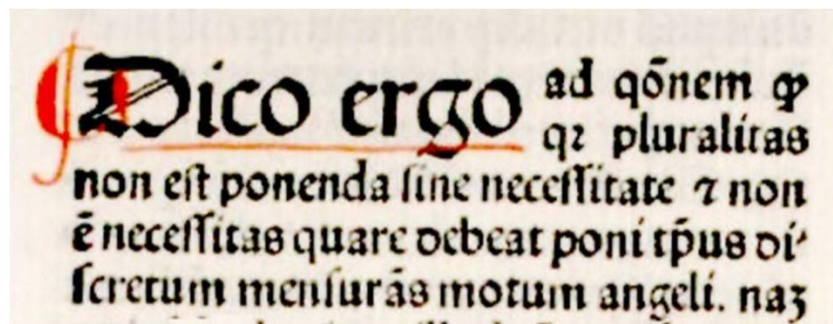
- Requirements
 - a model
 - tunable parameters
 - **objective function**: a repeatable quantitative performance measurement procedure
 - training observations
- Plan:
 - Automatically find patterns in the data

Ockham's ("Occam's") razor

- Earlier reference: John Duns Scotus, c. 1304

Dico ergo ad quaestionem, pluralitas non est ponenda sine necessitate, [et non est necessitas quare debeat poni tempus discretum mensurans motum angeli].

"Adressing this question, *plurality is not to be posited without necessity*, [and it's not necessary likewise to posit discrete time to measure the motion of angels]."



As for Ockham:
"It is futile to do with more things that which can be done with fewer." (c. 1495)

Ockham's razor

- Of all hypotheses $t \in \mathcal{H}$ with “parameters” $\mathcal{Y}_h = \{z_0, z_1, \dots, z_m\}$ which explain the observations, we seek:

$$\operatorname{argmin}_h |\mathcal{Y}_h|$$

number of parameters

- What's the simplest way to explain the observations?

“Make everything as simple as possible, but not simpler”
Albert Einstein

$$\operatorname{argmin}_h |\mathcal{Y}_h|$$

- In practice, this is not so easy
 - No tuple $(\mathbf{t}, \mathcal{Y}_h)$ may model the data with complete accuracy
 - Simplicity/accuracy trade-off
 - Accuracy and complexity are sometimes in direct relation
 - We are often faced with trading **reduced accuracy** for **increased simplicity**.
- Machine learning is a tool for exploring this space

Machine Learning

- Supervised
 - methods that require labeled training data
- Unsupervised
 - methods that are trained on raw (unlabeled) training data
- Semi-supervised
 - bootstrap an unsupervised method with a small amount of labeled data
 - “co-training” between two machine learners

Machine Learning Tasks

- Clustering
 - Unsupervised
 - We don't know what the clusters are/will be
 - We don't know the optimal number of clusters
- Classification
 - Supervised
 - We know the classes we seek



Both tasks partition the data into subsets

Partitioning a corpus

- Training set
 - initial training
- Development set (“dev-test”)
 - used for tuning
- Test set (“held-out”)
 - reserved for reporting results.
 - Model and parameters cannot be adjusted after touching this data

Train:
build model

Test:
evaluate unseens

Generative v. Discriminative

Generative models incorporate parameters which maximize the probability of the training set and then assume that they apply to the observed data

$$\operatorname{argmax}_{\text{?,T}} P(\text{?T}, \text{?T})$$

train/test

Discriminative models model a conditional probability which predicates on variables within the sample

$$\operatorname{argmax}_{\text{?}} P(\text{?T} | \text{?T})$$

train

$$\operatorname{argmax}_{\text{?}} P(\text{?} | \text{?})$$

test

Generative Models

- After training, you have a standalone model
- The model parameters have “captured” the training

$$\operatorname{argmax}_{\text{?T}} P(\text{?T}, \text{?T})$$

- Hidden Markov Model
- Naïve Bayes

Discriminative Models

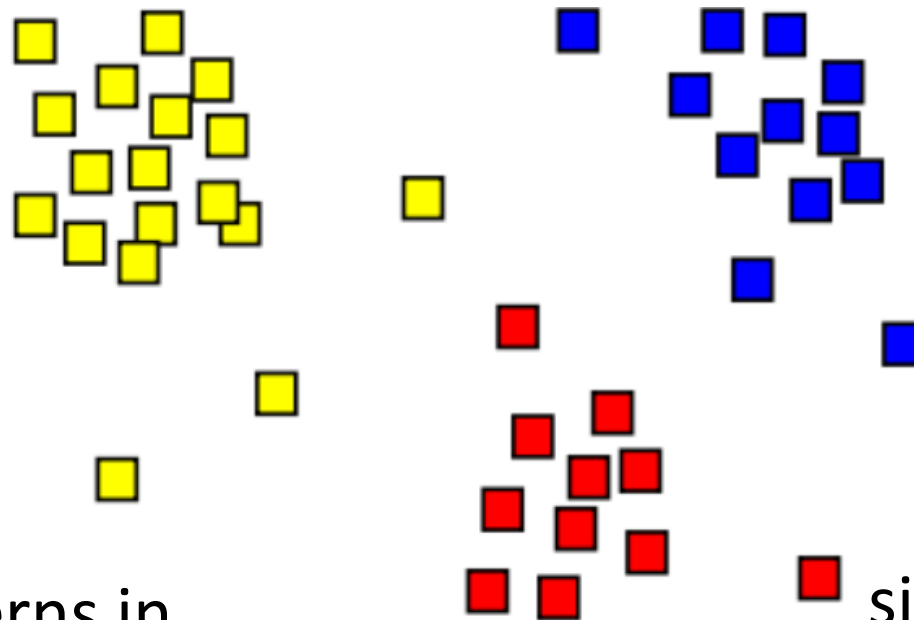
$$\underset{\text{?}}{\operatorname{argmax}} P(\text{?} | \text{?})$$

The diagram illustrates the concept of discriminative models. It shows a transition from a joint probability distribution $P(\text{T} | \text{T})$ to a conditional probability distribution $P(\text{?} | \text{?})$. Red arrows point from the 'T' document icons in the top expression to the '?' document icons in the bottom expression, indicating the shift from a specific class to an unknown class. The 'argmax' operator is shown with a '?' icon below it, representing the task of finding the most likely class for a given input.

- Linear regression
- Maximum entropy classifiers
- Conditional Random Fields (CRF)
- Support Vector Machines (SVM)

Clustering

Automated grouping



Learn patterns in
unlabeled data

Use a distance
metric to find
similarity amongst
instances

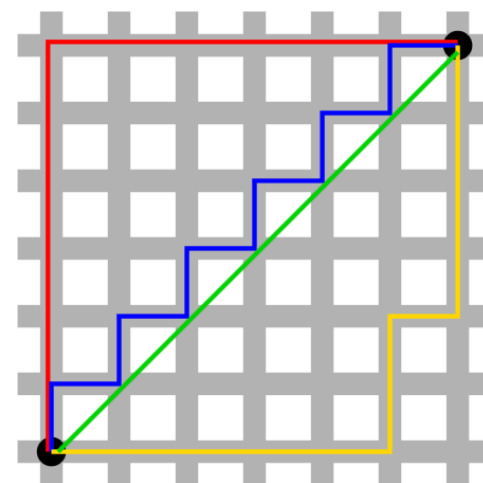
Distance Metrics

- Euclidian
- Cosine
- Hamming distance
- Taxicab distance
- others...

Properties of distance metrics

- Triangle inequality: $zz < AA + yy$?

Distance metrics
are also used in
kNN
classification



Types of clustering

- Hierarchical
 - Agglomerative (bottom-up)
 - Divisive (top-down)
- Non-hierarchical
 - k-Means
 - Expectation Maximization (EM)

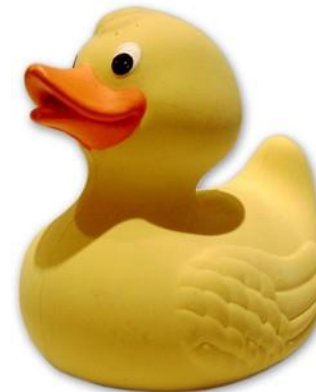
Uses of clustering

- Exploratory data analysis
 - Find groups in an initial analysis
- Generalization
 - Find equivalence classes in the data

Classification

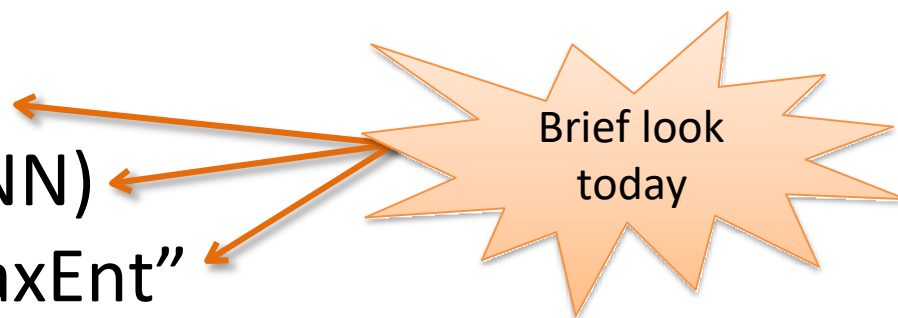
- Classification is automated labeling
- Example: Observe a bird. Its features are:
 - walks_like: a duck
 - swims_like: a duck
 - quacks_like: a duck

Classification result:



Types of classifiers

- Hidden Markov Model (HMM)
- Decision tree
- Support Vector Machine
- Perceptron
- Naïve Bayes
- k-Nearest Neighbor (kNN)
- Maximum Entropy “MaxEnt”
- Neural networks
- Conditional Random Fields (CRF)
- Many more...



Classifiers: genetic algorithms

- Relatively unexplored area of ML, especially within computational linguistics
- Requirements:
 - “Environment” definition
 - “Individual” definition
 - Fitness function
 - Breeding iterations
 - Mutation: random alteration
 - Crossing
 - Efficient model that can simulate millions of generations

Naïve Bayes Classification

- Intuitive
- Simple to implement
- Works well even compared to much more sophisticated techniques
- Does not require large amounts of training data

Why “naïve?”

- It is given this name because of the underlying **independence assumption**:
 - Features are conditionally independent from one another
- This greatly simplifies calculating the model
 - But may weaken the accuracy

Independence assumption

You're given a light, crisp, cold beer in the summertime. What is the probability that it is a lager?



$$NN(ffttffDDrr \mid ftttff \mathbf{t}^{DD}, iirrttttcc, iiNNffii, ttNNmmmmDDrr) = \\ NN(ffttff \mathbf{t}^{DD} \mid ftttffDDrr) NN(iirrttttcc \mid ftttffDDrr) NN(iiNNffii \mid ftttffDDrr) NN(ttNNmmmmDDrr \mid ftttffDDrr)$$



However, do some features influence the others? Is *any type of beer* more often to be served *cold* in the *summer*?

Naïve Bayes Classification

- Recall Bayes Theorem

$$NN(AA|aa) = \frac{NN(aa|AA)NN(AA)}{NN(aa)}$$

- Language classification

$$NN(ffttnnff|DDDDAADD) = \frac{NN(DDDDAADD|ffttnnff)NN(ffttnnff)}{NN(DDDDAADD)}$$

Naïve Bayes language classifier

$$NN(ffttnnff|DDDDAADD) = \frac{NN(DDDDAADD|ffttnnff) \cancel{NN(ffttnnff)}}{\cancel{NN(DDDDAADD)}}$$

$NN(DDDDAADD)$ - Prior probability that the text is in (some) language: 1.0

$NN(ffttnnff)$ – Prior probability of encountering each language: assume all languages are equally likely

Naïve assumption

All **features** are independent of all others

For this task, a “feature” is the occurrence of a word

$$NN(ffttnnff | DDDDAADD)$$

$$= NN(ffttnnff | ww_1, ww_2, \dots, ww_{nn})$$

$$= NN(ww_1, ww_2, \dots, ww_{nn} | fttnnff)$$

$$= \prod_{ii=1}^{nn} NN(ww_{ii} | fttnnff)$$

$$\logprob(ffttnnff | DDDDAADD) = \prod_{ii=1}^{nn} NN(ww_{ii} | fttnnff)$$

Last step

$$\log \text{prob} (f(ttnnff) \mid D) = \prod_{i=1}^n \log^{NN}(w_{ii} \mid f(ttnnff))$$

This gives the (log-)probability of a language given a text. To find the **most** probable language:

$$L = \underset{j}{\text{argmax}} \prod_{i=1}^n \log^{NN}(w_{ii} \mid f(ttnnff)_j)$$

k-Nearest Neighbor Classification

- “Classification by peer pressure”
- Instance-based learning (“lazy learning”)
 - No training
- Need a **distance metric**
- Test instance is given the same label as its *bb* closest neighbors
 - Voting schemes resolve conflict
- To test, need to calculate distance to all training instances
 - This can be slow at runtime

kNN Distance metric

- Should be fast to calculate
- Usually, just a high-dimensional vector space model (VSM)



VSM

Vector Space Model

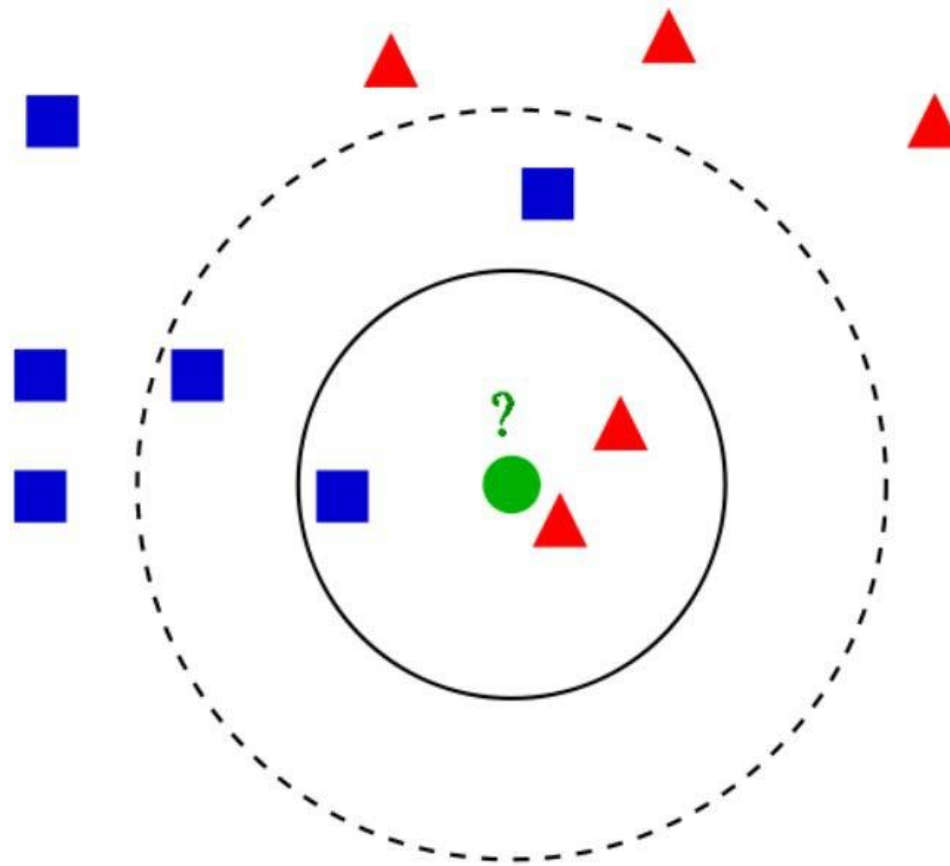
n-Dimensional Euclidian space
where each feature has its
own axis of variability

SVM

Support Vector Machine

A particular type of quadratic
programming classifier

kNN Classification



<http://en.wikipedia.org/wiki/File:KnnClassification.svg>

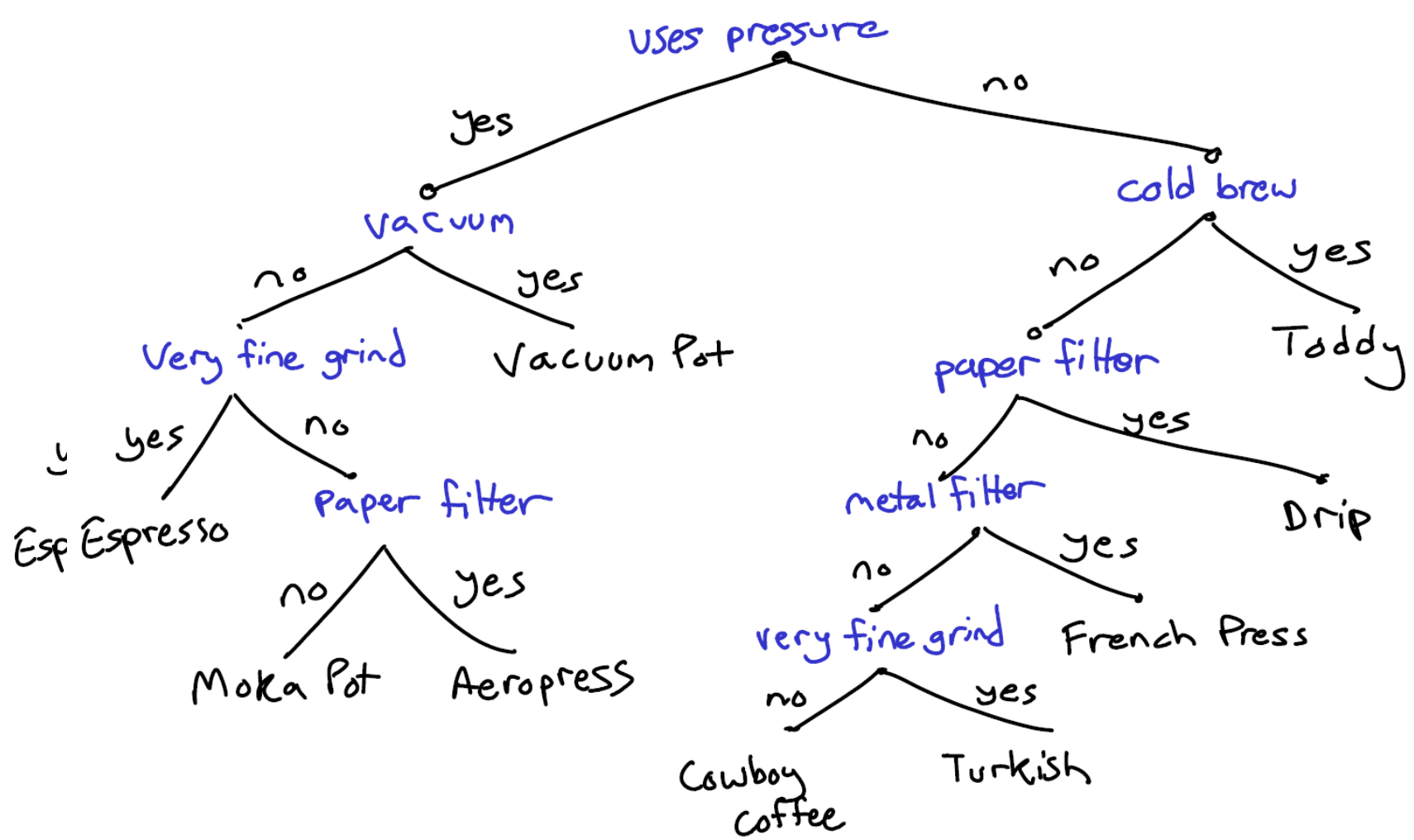
kNN Voting schemes

- Majority voting
 - Choose majority class amongst k closest neighbors
- Weighted voting
 - Weight each of the k neighbors' labels according to the distance to the training instance
 - In principle, this can be applied to an all-neighbors approach

Decision Tree Classifier

- Build a tree where each node represents a test
 - Decision tree: leaf nodes assign labels
 - Regression tree: leaf nodes assign real values
- Decide quality measure for choosing branching features
- Building the tree is expensive, but testing is fast
- Overfitting the data can be a problem

Coffee-makers



Building the tree

- Choose feature that is most discriminative across the training set
 - **Information gain** is commonly used
- Split the training data according to this feature
- Repeat for each subset of data
- Stop at some threshold

Information Theory

- A stochastic look at “information”
- The more **uncertain** a system, the more **bits** are required to describe it

following slides from Rob Malouf

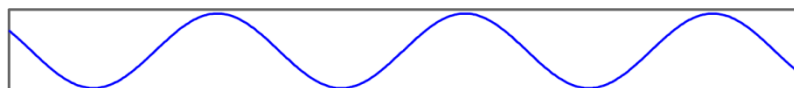
Information Theory

Claude Shannon. 1948. *A mathematical theory of communication.*

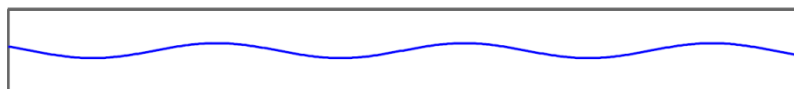
“The fundamental problem of communication is that of **reproducing** at one point... a message **selected** at another point... Semantic aspects of communication are irrelevant to the engineering problem. The significant aspect is that the actual message is one **selected from a set** of possible messages.”

Information theory

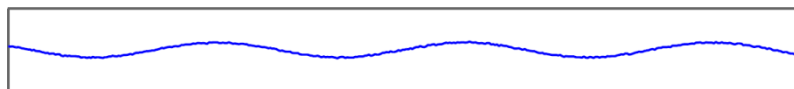
original signal



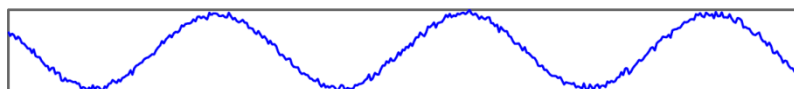
attenuate



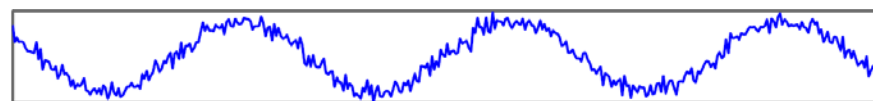
add noise



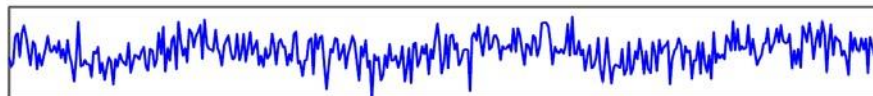
boost



Repeat process 5 times



Repeat process 100 times



Information theory

- Digital communications involves the transfer of symbols
- drawn from a discrete alphabet
 - English letters
 - English words
 - Decimal digits
 - Binary digits
 - DNA sequences
 - Quantized analog signals

Encoding information

- Minimal “piece” of information is one **bit**
- A bit can take on two values: $\{ 0, 1 \}$
- There are 2^{bb} ways to arrange W bits
- Therefore the number of bits required to encode nn different sequences is:

$$\lceil \log_2 nn \rceil$$

Example

- Transmit information about a poker hand
{ straight flush, four of a kind, full house, flush, straight, three of a kind, two pair, pair, high card }
- There are 9 “messages”
- Baseline message length:

$$\lceil \log_2 9 \rceil = 4$$

Binary code for poker hands

straight flush	0000
four of a kind	0001
full house	0010
flush	0100
straight	1000
three of a kind	0011
two pair	0101
pair	1001
high card	0111

Note: Some messages (e.g. 0110, 1010...) are unused; suggesting that there is waste in this encoding

Prefix encoding

- Probabilities can be used to reduce the expected message length

straight flush	0.0000154	000011
four of a kind	0.000240	0000100
full house	0.00144	0000101
flush	0.00196	00000
straight	0.00393	0001
three of a kind	0.0211	010
two pair	0.0475	011
pair	0.422	001
high card	0.501	1

- Now the expected length is reduced from 4 bits to 2.01 bits

Encoding information

- This encoding is even better

straight flush	0.0000154	11111111
four of a kind	0.000240	11111110
full house	0.00144	1111110
flush	0.00196	111110
straight	0.00393	11110
three of a kind	0.0211	1110
two pair	0.0475	110
pair	0.422	10
high card	0.501	0

- Here, the expected number of bits per hand is 1.61
- Can we do better? How would we find out?

Information and probability

- The information encoded is the identity of the poker hand
- The length of the message ought to be related to its information content
- A message that the opponent only has a pair or high card seems less informative than a message that they have four of a kind
 - because it happens more often
- Transmitting rare messages is more informative than transmitting common ones
- How can we make this more precise?

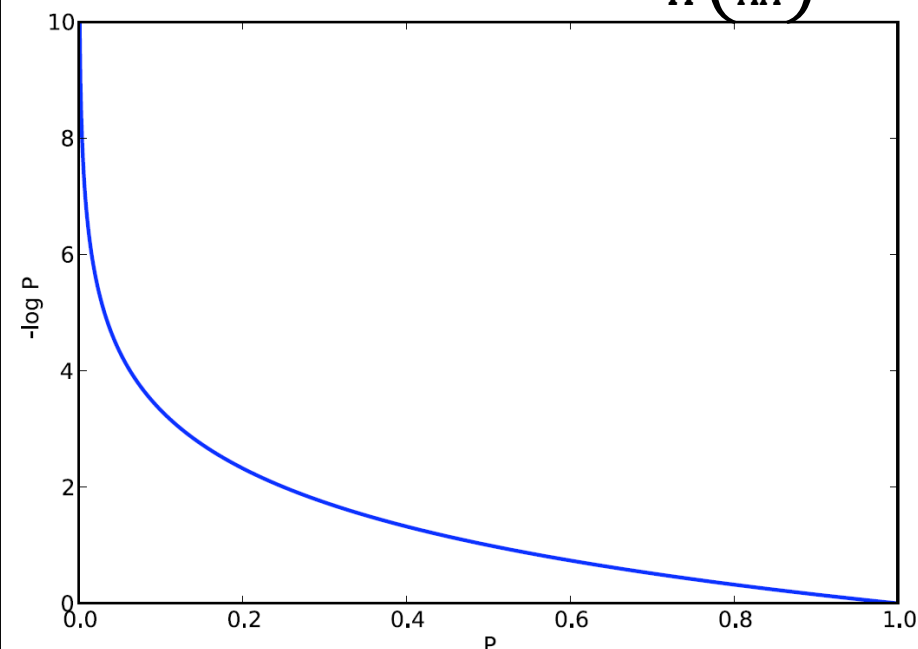
Information

- Let's assume information content of a message $II(cc)$ is a function of its probability
- Some basic properties that it seems like $II(cc)$ should have:
 - Information is non-negative $II(cc) \geq 0$
 - Certain messages contain no information $II(1) = 0$
 - Information should be additive for jointly occurring independent messages
 - $II(cc)$ should monotonically decrease versus cc

Information

- One math function which matches these criteria is

$$H(AA) = -\log_{bb} cc(AA)$$



the base VV doesn't matter too much, because it just changes the measure by a constant factor

For $VV = 2$ we are measuring in bits
For $VV = DD$ we are measuring in nats
For $VV = 10$ we are measuring in hartleys

Entropy

- For the information content a message or a whole system, which is called its **entropy**, we sum over all possible messages or states
 - If we knew in advance which message we're selecting, we wouldn't need to code it

Entropy

The measure of uncertainty in a system

$$H(X) = - \sum_{i=1}^N P(x_i) \log_2 P(x_i)$$

Information Theory

- Joint Entropy

$$H(X, Y) = - \sum_{xx} \sum_{yy} P_{xx, yy} (\log P_{xx, yy})$$

- Conditional Entropy

$$H(Y|X) = H(X, Y) - H(X)$$

- Mutual Information (or Information Gain): the expected reduction in entropy due to knowing something

$$IG(Y|X) = H(Y) - H(Y|X)$$

Source Coding Theorem (Shannon)

The expected code length $EE[CC]$ for a random variable XX under an optimal encoding is

$$HH(X) \leq EE[CC] \leq HH(X) + 1$$

- This gives a lower bound for lossless compression and cryptography
- Guarantees that there is such an encoding
- Establishes the link between a probability distribution and information representation
- For the poker hands, $HH(X) = 1.42$

Maximum Entropy Classification

- Model what is known; assume nothing about what is unknown
- Find a distribution that maximizes entropy (assumes the least)

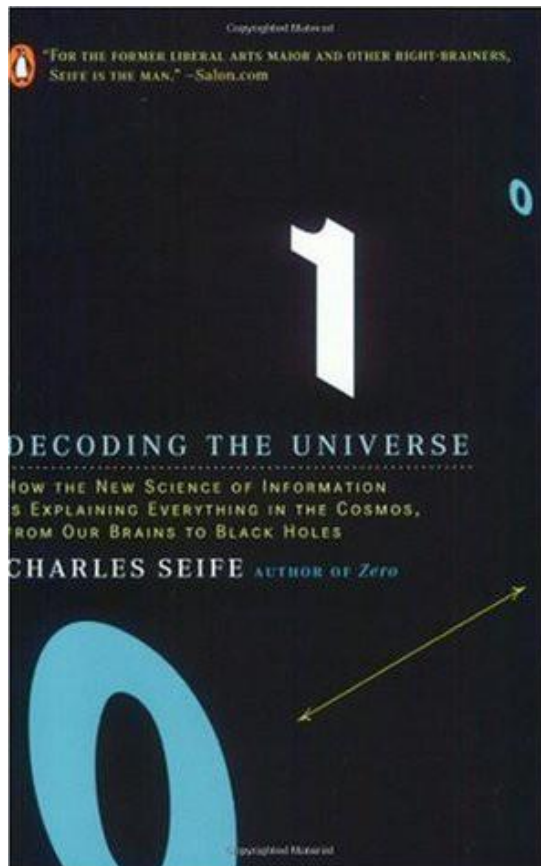
$$P(y|A) = \frac{\sum_j \lambda_j f_j(x, y)}{Z}$$

Training: Try to estimate λ_j such that

$$c^* = \operatorname{argmax}_{c \in \mathcal{C}} H(c)$$

Testing: Evaluate $P(y|A)$

Further information on... information theory



**Decoding the Universe: How the New Science
of Information Is Explaining Everything in the
Cosmos, from Our Brains to Black Holes**
January 30, 2007

by [Charles Seife](#)

note: this is a popular science
recommendation, not a textbook
or academic treatment

C# Tutorial (continued...)

IEnumerable, yield, and deferred execution

- Before describing the trie data structure, let's look at iterators which enumerate a sequence of elements

Examples in C#. If you use another language, it will be instructive to think about how to adapt the solutions to your language

- Enumeration is obvious when the data is at hand and you want to use it all:

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };  
  
foreach (String s in data)  
    Console.WriteLine(s);
```

We can pass (a reference to) the array around too, no problem

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };  
// ...  
ProcessSomeStrings(data);  
// ...
```

```
void ProcessSomeStrings(String[] the_strings)  
{  
    foreach (String s in the_strings)  
        Console.WriteLine(s);  
}
```

What if we only want to “process” the four-letter words?

```
String[] data = { "able", "bodied", "cows", "don't", "eat", "fish" };  
// ...
```

```
List<String> filtered = new List<String>();  
foreach (String s in data)  
    if (s.Length == 4)  
        filtered.Add(s);  
ProcessSomeStrings(filtered);  
// ...
```

This doesn't seem very nice. For one thing, we have to use more memory and waste time copying the elements we care about to a new list.

```
void ProcessSomeStrings(List<String> the_strings)  
{  
    foreach (String s in the_strings)  
        Console.WriteLine(s);  
}
```

Is there a way to pass this function enough information to filter the *original list* itself, where it lies?

- Remember the non-filtered example for a second

```
void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

- The processing function doesn't really care about the fact that the data is in an array
- This violates an important programming maxim:

A flexible interface *demands the least* and *provides the most*:

- Inputs* are as general as possible (allowing clients to supply any level of specificity, i.e. be lazy)
- Outputs* are as specific as possible (allowing clients to capitalize on work products, i.e. be lazy).

```
void ProcessSomeStrings(String[] the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

The extra (unused) demands this function is making by asking for String[]:

- That the strings all be in memory at the same time
 - That the strings be randomly accessible by an index
 - That the number of strings be known and fixed before the function starts
-
- To modify this to comply with the maxim, we first ask:
 - Q: What is the absolute minimum that this function actually needs to accomplish it's work?
 - Answer: a way to iterate strings

Interfaces

- `IEnumerable<T>` is one of many system-defined **interfaces** that a class can elect to implement

An **interface** is a named set of zero or more function signatures with no implementation(s)

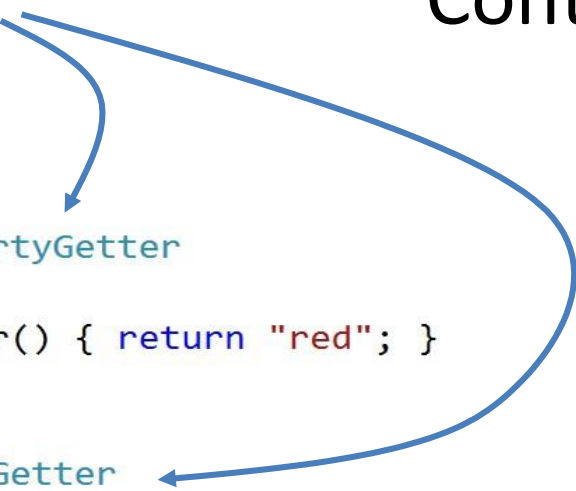
- To implement an interface, a class defines a matching implementation for every function in the interface
- Interfaces are sometimes described as contracts
- You can define and use a reference to an interface just like any other object reference

Contrived Example

```
interface IPropertyGetter
{
    String GetColor();
}

class Strawberry : IPropertyGetter
{
    public String GetColor() { return "red"; }
}

class Ferrari : IPropertyGetter
{
    public String GetColor() { return "yellow"; }
}
```



- This looks like C++ class inheritance
 - yes, but it's more ad-hoc
 - C# classes can have **single inheritance** of other classes, and **multiple inheritance** of interfaces
 - Interfaces can inherit from other interfaces (not shown)

IEnumerable<T>

- This is one of the simplest interfaces defined in the BCL (base class libraries)
- This interface provides just one thing: a way to iterate over elements of type T
- All of the system arrays, collections, dictionaries, hash sets, etc. implement IEnumerable<T>
 - Implementing IEnumerable<T> on your own classes can be very useful, but you don't need to worry about that
 - For now, what's important is that you get to use it, because it's available on all of the system collections

IEnumerator<T>

- **IEnumerable<T>** has only one function, which allows a caller or caller(s) to obtain an *enumerator* object which is able to iterate over elements
 - The actual enumerator object is an object that implements a different interface, called **IEnumerator<T>**
 - This “factory” design allows a caller to initiate and maintain several simultaneous iterations if needed
 - The enumerator object, **IEnumerator<T>** can only:
 - Get the current element
 - Move to the next element
 - Tell you if you’ve reached the end
 - Note: There’s no count
 - ICollection inherits from IEnumerable to provide this

Interfaces as function arguments

- Using interfaces as function arguments allows you to require the absolute minimum functionality the function actually needs
- In this way, the ad-hoc nature of interfaces allows us to comply with the maxim

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

Now, this function is exposing the **weakest (most general) requirement** possible for the processing it has to do. This provides more flexibility to callers since they can choose whatever level of specificity is convenient. The function can be used in the widest possible variety of situations.

Example

```
String[] d1 = { "able", "bodied", "cows", "don't", "eat", "fish" };  
ProcessSomeStrings(d1);
```


```
List<String> d2 = new List<String> { "clifford", "the", "big", "red", "dog" };  
ProcessSomeStrings(d2);
```

```
HashSet<String> d3 = new HashSet<String> { "these", "must", "be", "distinct" };  
ProcessSomeStrings(d3);
```

```
Dictionary<String,int> d4 =  
    new Dictionary<String, int> { { "the", 334596 }, { "in", 153024 } };  
ProcessSomeStrings(d4.Keys);
```

```
void ProcessSomeStrings(IEnumerable<String> the_strings)  
{  
    foreach (String s in the_strings)  
        Console.WriteLine(s);  
}
```

Python users might not be impressed, but the difference is that this is all 100% strongly typed



Iteration is efficient

- That's cool, `IEnumerable<T>` lets a function **not care** about where a sequence of elements is coming from
 - We don't copy the elements around
 - Iterators let us access elements right from their source
- All of those examples iterate over elements that **already exist** somewhere
- Is there a way to iterate over data that's generated on-the-fly, doesn't exist yet, or is never persisted at all?
- Yes!

Iterating over on-the-fly data

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}
```

see next slide

```
// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...
```

This is exactly the same as before, but this time there's no "collection" of elements sitting anywhere

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This function doesn't care. In fact, it can't even tell.

yield keyword

- The **yield** keyword makes it easy to define your own custom iterator functions
- Any function that contains the `yield` keyword becomes special
 - It must be declared as returning an `IEnumerable<T>`
 - Deferred execution means that the function's body is not necessarily invoked when you "call" it
 - It must deliver zero or more elements of type `T` using:
`yield return t;`
 - Sometime later, control may continue immediately after this statement to allow you to yield additional elements
 - It may signal the end of the sequence by using:
`yield break;`

Custom iterator function example

```
IEnumerable<String> GetNewsStories(int desired_count)
{
    for (int i = 0; i < desired_count; i++)
        yield return RealtimeNewswireSource.GetLatestStory();
}
```

code from this custom iterator function is *not* executed at this point.

```
// ...
IEnumerable<String> d5 = GetNewsStories(7);
ProcessSomeStrings(d5);
// ...
```

d5 refers to an iterator that “knows how” to get a certain sequence of strings when asked

```
void ProcessSomeStrings(IEnumerable<String> the_strings)
{
    foreach (String s in the_strings)
        Console.WriteLine(s);
}
```

This finally demands the strings, causing our custom iterator function to execute—interleaved with this loop!

Closures

- Lambda expressions automatically capture local variables that they reference, which are then passed around as part of the lambda variable
 - Caution: languages do this differently with respect to reference (the lambda expression will modify the original value) versus value (the lambda expression has a snapshot of the value)
- This can lead to interesting scoping issues

Lambda expressions

```
// recall Select(ch => ('a' <= ch && ch <= 'z') || ch == '\\' ? ch : ' ');

String s = "Al's 20 fat-ish oxen.";
Func<Char, Char> myfunc = (ch) => ('a' <= ch && ch <= 'z') || ch == '\\' ? ch : ' ';
IEnumerable<Char> iech = s.Select(myfunc);
// iech is now a deferred enumerator for the characters in: " l's    fat ish oxen "

Func<Char, Char> myfunc = (ch) =>
{
    if ('a' <= ch && ch <= 'z')
        return ch;
    if (ch == '\\')
        return ch;
    return ' ';
};

Func<String, int, bool> string_is_longer_than = (s, i) => s.Length > i;

bool b = string_is_longer_than("hello", 3);    // true
```

Closure example

```
int x = 3;
Action a = () =>
{
    Console.WriteLine(x);
};
a();           // prints 3
x = 5;
a();           // prints 5
```

State machine example

```
using System;
using System.Collections.Generic;
using System.Linq;

static class Program
{
    static class MainClass
    {
        enum State { Zero, One, Two };

        static Dictionary<State, Func<Char, State>> machine = new Dictionary<State, Func<Char, State>>
        {
            {
                State.Zero, (ch) => { return State.Two; }
            },
            {
                State.One, (ch) => { return State.One; }
            },
        };

        static void Main(String[] args)
        {
            String s = "the string to parse";
            int i = 0;

            State state = State.Zero;
            while (i < s.Length)
                state = machine[state](s[i++]);

        }
    }
}
```

A dictionary
of lambda
functions

The state machine

LINQ in C#

- Sequences: `IEnumerable<T>`
- Deferred execution
- Type inference
- Strong typing
 - despite the 'var' keyword
 - (C# 4.0 now has the 'dynamic' keyword, which allows true runtime typing where desired)

LINQ operators

- Filter/Quantify:
Where, ElementAt, First, Last, OfType
- Aggregate:
Count, Any, All, Sum, Min, Max
- Partition/Concatenate:
Take, Skip, Concat
- Project/Generate:
Select, SelectMany, Empty, Range, Repeat
- Set:
Union, Intersect, Except, Distinct
- Sort/Ordering:
OrderBy, ThenBy, OrderByDescending, Reverse
- Convert/Render:
Cast, ToArray, ToList, ToDictionary

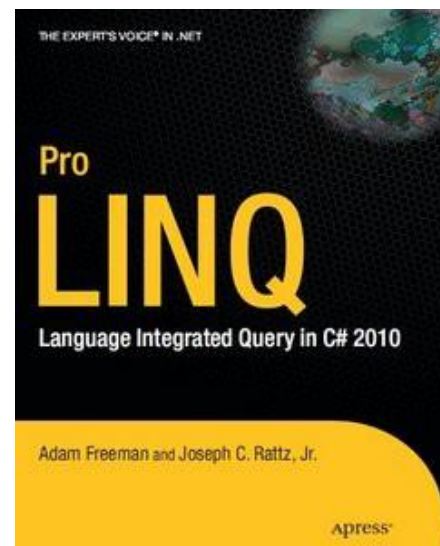
Example

```
Dictionary<String, int> pet_sym_map =  
    File.ReadAllLines("/programming/analytical-grammar/erg-funcs/pet-symbol-key.txt")  
    .Select(s => s.Split(sc7, StringSplitOptions.RemoveEmptyEntries))  
    .Where(rgs => rgs.Length == 3)  
    .Select(rgs => new { id = int.Parse(rgs[0]), sym = rgs[1].Trim().ToLower() })  
    .GroupBy(a => a.sym)  
    .Select(grp => grp.ArgMin(a => a.id))  
    .ToDictionary(a => a.sym, a => a.id);
```

C# LINQ (Language Integrated Query)

- Declarative operations on sequences
- Recommendation:

Joseph C. Rattz, Jr. (2007) *Pro LINQ: Language Integrated Query in C# 2008*. Apress.




Programming Paradigms: Procedural

- Procedural (“imperative”) programming
 - FORTRAN (1954) grew out of hardware assembly languages, which are necessarily procedural
 - We explicitly specify the (synchronous) steps for doing something (i.e. an algorithm)

```
int Factorial(int n)
{
    int f = 1;
    for (int i = n; i > 1; i--)
        f *= i;
    return f;
}
```

Functional Programming

- A type of declarative programming
 -  Constraint-based syntax formalisms such as unification grammars (LFG, HPSG, ...) are also declarative
- Like function definitions in math, the “program” describes asynchronous relationships
- Functions are stateless and should have no side-effects
- Immutable values
 - As a bonus, this really facilitates concurrent programming
- Scheme, Haskell, F#

F# Example

- F# interactive on patas:

```
$ mono /opt/fsharp/bin/fsi.exe --gui-  
      (you must use an ANSI terminal)
```

```
> let rec factorial = function  
    | 0 -> 1  
    | n -> n * factorial(n -  
1);; val factorial : int -> int  
> factorial 5;;  
val it : int =  
120  
> #quit;;
```