In [1]:
```
from pylab import *
```

Markov Chains

Assume we have a state space consisting of N states $0 ... N-1$.

A *Markov chain* is an infinite sequence of random states X_i such that

$$P(X_{n+1} = x | X_1 = x_1 \dots X_n = x_n) = P(X_{n+1} = x | X_n = x_n)$$

That is, the probability of each subsequent state only depends on the previous state.

There are many variations:

- usually, the probability is independent of n (*time homogeneous*)
- the probability can depend on a bounded number of previous states giving a higher order Markov chain
- the set of states can be countably infinite
- the state index n can be continuous (continuous time Markov process)

We represent time homogeneous Markov chains with a finite state space as a matrix of transition probabilities M_{ij}.

Here, $P(j \to i) = M_{ij}$ and $\sum_i M_{ij} = 1$.

In [2]:
```
M = rand(5,5)
```

In [3]:
```
sum(M,axis=0)
```

Out[3]:
```
array([ 3.60015634,  1.61285595,  3.19101919,  3.26341937,  1.79654585])
```

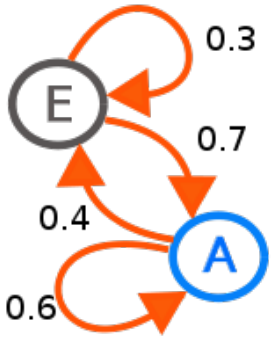
In [4]:
```
M /= sum(M,axis=0)[newaxis,:]
sum(M,axis=0)
```

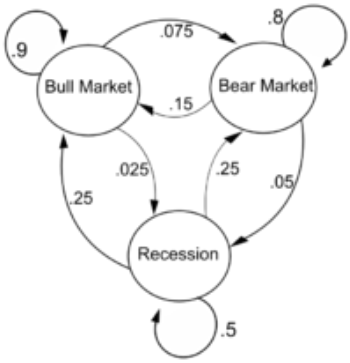
Out[4]:
```
array([ 1.,  1.,  1.,  1.,  1.])
```

We call M a *transition matrix* or a *stochastic matrix*.

(Note that the matrix is often written the other way around in the literature.)

We often draw a transition diagram; this shows the state and the probabilities of transitions between them:





We can compute the probability of being in the next state given that we are in state 0 with a matrix multiplication.

In [5]:
```
state = zeros(5)
state[0] = 1
dot (M,state)
```

Out[5]:
```
array([ 0.25597683,  0.16287684,  0.24856257,  0.13584864,  0.19673512])
```

Sampling from a Markov Chain

To sample from a Markov chain, we need to be able to sample randomly from a discrete distribution $P(x)$.

This is actually quite simple:

- compute the cumulative distribution function $c_x = \sum_{i=1}^{x} P(x)$
- pick a uniformly random sample $s \in [0, 1]$
- return $\arg\max_x \{x | c_x < s\}$
- we speed up the latter search with a binary search

In [6]:
```
v = add.accumulate(M[:,0])
```

In [7]:
```
def binsearch(a,x,lo,hi):
    while lo<hi:
        mid = (lo+hi)//2
        v = a[mid]
        if v<x: lo = mid+1
        elif v>x: hi = mid
        else: return mid
    assert lo==hi
    return lo
```

In [8]:
```
binsearch(v,0.9,0,len(M))
```

Out[8]: 4

In [9]:
```
def rsample(dist):
    assert amin(dist)>=0.0 and amax(dist)<=1.0
    v = add.accumulate(dist)
    assert abs(v[-1]-1.0)<1e-6
    val = rand()
    result = binsearch(v,val,0,len(v))
    return result
```

In [10]:
```
def sample_chain(M,state,N):
    result = [state]
    for i in range(N):
        state = rsample(M[:,state])
        result.append(state)
    return result
```

In [11]:
```
sample_chain(M,0,10)
```

Out[11]: [0, 0, 3, 3, 0, 3, 0, 4, 4, 4, 3]

In [12]:
```
M.shape
```

Out[12]: (5, 5)

n-grams and Markov Chains

An n-gram is a sequence of n letters, syllables, words or other linguistic entity (usually words).

An n-gram model is a model of the probability that n words occur.

Usually, n-gram models are expressed as a conditional probability, given a very long sequence of words $w_1 \ldots w_L$:

$P(w_i|w_{i-1} \ldots w_{i-(n-1)})$

Sometimes, it might alwo be expressed as a joint probability:

$P(w_i \ldots w_i - n - 1$

(Boundary conditions are handled by adding special symbols to "fill things up".)

An n-gram model defines a Markov chain.

Google has released a large *n*-gram model for English and other languages, and you can experiment with it here:

http://books.google.com/ngrams/

In [13]:
```
s = """Humpty Dumpty sat on a wall,
Humpty Dumpty had a great fall;
All the King's horses and all the King's men
Couldn't put Humpty together again""".lower()
```

In [14]:
```
import re
words = re.split(r'\W+',s)
words[:10]
```

Out[14]:
```
['humpty', 'dumpty', 'sat', 'on', 'a', 'wall', 'humpty', 'dumpty', 'had', 'a']
```

In [15]:
```
from collections import Counter
bigrams = Counter()
for i in range(len(words)-1):
    bigrams[tuple(words[i:i+2])] += 1
```

In [16]:
```
bigrams
```

Out[16]:
```
Counter({('the', 'king'): 2, ('king', 's'): 2, ('all', 'the'): 2, ('humpty',
'dumpty'): 2, ('horses', 'and'): 1, ('humpty', 'together'): 1, ('couldn', 't'):
1, ('on', 'a'): 1, ('s', 'men'): 1, ('t', 'put'): 1, ('s', 'horses'): 1, ('had',
'a'): 1, ('wall', 'humpty'): 1, ('together', 'again'): 1, ('a', 'wall'): 1,
('a', 'great'): 1, ('men', 'couldn'): 1, ('great', 'fall'): 1, ('put',
'humpty'): 1, ('fall', 'all'): 1, ('dumpty', 'sat'): 1, ('sat', 'on'): 1,
('and', 'all'): 1, ('dumpty', 'had'): 1})
```

In [17]:
```
sorted(bigrams.items())
```

Out[17]:
```
[(('a', 'great'), 1),
 (('a', 'wall'), 1),
 (('all', 'the'), 2),
 (('and', 'all'), 1),
 (('couldn', 't'), 1),
 (('dumpty', 'had'), 1),
 (('dumpty', 'sat'), 1),
 (('fall', 'all'), 1),
 (('great', 'fall'), 1),
 (('had', 'a'), 1),
 (('horses', 'and'), 1),
 (('humpty', 'dumpty'), 2),
 (('humpty', 'together'), 1),
 (('king', 's'), 2),
 (('men', 'couldn'), 1),
 (('on', 'a'), 1),
 (('put', 'humpty'), 1),
 (('s', 'horses'), 1),
 (('s', 'men'), 1),
 (('sat', 'on'), 1),
 (('t', 'put'), 1),
 (('the', 'king'), 2),
 (('together', 'again'), 1),
 (('wall', 'humpty'), 1)]
```

So, we see that:

$P(\text{greatla}) = 0.5$

$P(\text{wallla}) = 0.5$

$P(< \text{anything else} > \text{la}) = 0.0$

Note that there are many pairs of words (even in this restricted vocabulary) that we don't see, even though their probability isn't zero.

The process of fixing this is called *smoothing*.

A simple mechanism is *pseudocounts*: here, we assume that anything that hasn't occurred has actually occurred some small number of times (1, 0.5, or smaller).

Alternatively, we can compute n-grams for different n and linearly interpolate.

Note that n-grams for $n > 2$ give rise to higher order Markov chains.

We can transform higher order Markov chains into first order Markov chains by associating states with each context of $n - 1$ words. However, then the emitted symbol differs from the state label.

State Distributions and Steady State

Assume we are given a vector of probabilities p of states.

The probability distribution after one state transition is just the product $M \cdot p$.

In [18]:
```
p = zeros(5)
p[0] = 1.0
```

In [19]:
```
for i in range(10):
    print p
    p = dot(M,p)
```

```
[ 1.  0.  0.  0.  0.]
[ 0.25597683  0.16287684  0.24856257  0.13584864  0.19673512]
[ 0.17948957  0.21895805  0.18838814  0.14250297  0.27066127]
[ 0.15713521  0.22432154  0.18493953  0.13602575  0.29757797]
[ 0.15273871  0.22165457  0.18532782  0.1331679   0.30711101]
[ 0.15232334  0.21904625  0.18613681  0.13229857  0.31019503]
[ 0.15261353  0.21757972  0.18662398  0.13211676  0.31106601]
[ 0.15287207  0.21690412  0.18685461  0.13211322  0.31125598]
[ 0.15301436  0.21662935  0.18694996  0.13213655  0.31126978]
[ 0.15307918  0.21652856  0.18698542  0.13215364  0.3112532 ]
```

This should look familiar: after a lot of applications of the matrix, it converges to a *steady state* distribution.

This steady state distribution is an *eigenvector* of the matrix.

There are a number of concepts about Markov chains that are important:

- a state j is *accessible* from a state i if there is some path connecting the two states with non-zero probability
- two states *communicate* if they are accessible from each other
- a Markov chain is *irreducible* if all states are accessible to each other
- a state is *transient* if there is a non-zero probability that we never return to it
- a state is *recurrent* if it is not transient
- a state is *positive recurrent* if the expected recurrence time is finite
- a state is *periodic* with period k if we are guaranteed to return to it exactly every k steps
- a state is *absorbing* if there are no transitions out of it
- a Markov chain is *absorbing* if every state can reach an absorbing state
- a state is *ergodic* if it is positive recurrent and not periodic

Can we construct matrices illustrating these concepts?

Reversible Markov Chains

A Markov chain is *reversible* if there is some probability distribution p such that

$$p \cdot M = M \cdot p$$

Note that $p \cdot M$ is the Markov chain running in reverse.

This means that if p is the *steady state distribution*, running the Markov chain forwards or backwards gives the same result.

Reversible Markov chains are very important for modern pattern recognition, statistics, physics, and simulations. They are used in Gibbs samplers and the Metropolis algorithm for probabilistic decision making.

Hidden Markov Models

In Markov chains,

- the state sequence is what we are directly interested in
- we can "observe" the state sequence directly

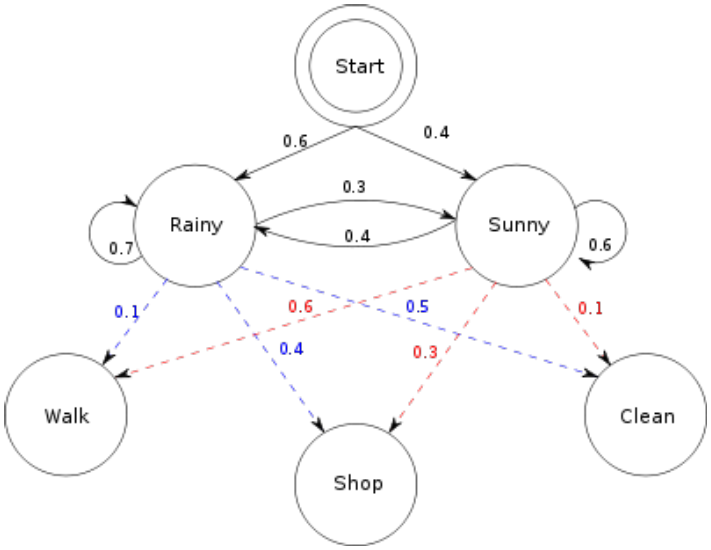
In *Hidden Markov Models*...

- states are associated with actions or observations
- often we can only observe the actions or observations and want to infer the state

That is, each state x *emits* one symbol y from a set of k symbols.

Extremely widely used:

- speech recognition: hidden state is part-of-phoneme, observation is acoustic signal
- handwriting recognition: hidden state is part-of-handwriting, observation is pen position
- bioinformatics: hidden state is part-of-gene, observation is DNA "letter"
- natural language processing: hidden state is part-of-speech, observation is word



To specify an HMM, we need:

- sets of states, a symbol alphabet
- a matrix of transition probabilities between states
- a matrix of emission probabilities, one for each state

HMM Algorithms

Usually, we are only given a training set of observations (outputs) $Y_1 \ldots Y_M$, but may also be given a model (state transition matrix, emission probabilities).

There are several different things we may want to compute:

- given a model M and a sequence of observations Y
  - determine $P_M(Y)$
  - determine the most likely state sequence X (most likely explanation)
  - determine the probability distribution of the next state (filtering)
  - determine the probability distribution for state X_k (smoothing)
- given a set of training observations Y and constraints on the model
  - find a model that assigns the overall highest likelihood to the training data

Why do we need constraints? Is there a trivial example?