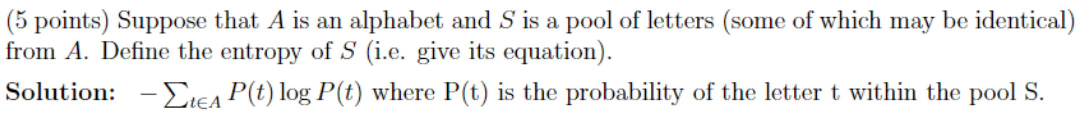
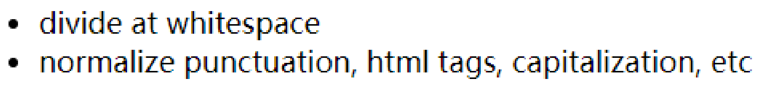
Midterm 2 possible questions


(5 points) Suppose that A is an alphabet and S is a pool of letters (some of which may be identical) from A. Define the entropy of S (i.e. give its equation).

Solution: $-\sum_{t\in A} P(t)\log P(t)$ where P(t) is the probability of the letter t within the pool S.

- Word types vs. word tokens
  word type: a dictionary entry such as "cat"
  word token: a word in a specific position in the text
- **For n words, there are n^2 possible bigrams. But our amount of training data stays the same. So the training data is even less adequate for bigrams than it was for single words.**

- Tokenization
  Our first job is to produce a clean string of words, suitable for use in the classifier.

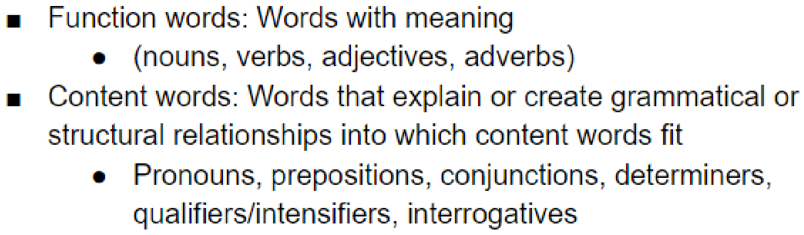


- divide at whitespace
- normalize punctuation, html tags, capitalization, etc

Format features such as punctuation and capitalization may be helpful or harmful, depending on the task.
So "normalize" may either involve throwing away the information or converting it into a format that's easier to process.

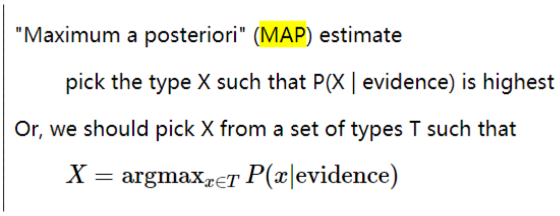
- stemming (Porter stemmer)
  Stemming converts inflected forms of a word to a single standardized form. It removes prefixes and suffixes, leaving only the content part of the word (its "stem")
  i.e. help, helping, helpful -> help

- stop words & rare words
  Very frequent words ("stop words") often convey very little information about the topic. So they are typically deleted before doing a bag of words analysis.
  Rare words may be deleted. More often, all rare words are mapped into a single placeholder value (e.g. **UNK**). This allows us to treat them all as a single item, with a reasonable probability of being observed.
- Backchannel: occurs when one participant is speaking and another participant interjects responses to the speaker (“hmm”, “uh-huh”, “yeah”)
- function vs. content



- Function words: Words with meaning
  - (nouns, verbs, adjectives, adverbs)
- Content words: Words that explain or create grammatical or structural relationships into which content words fit
  - Pronouns, prepositions, conjunctions, determiners, qualifiers/intensifiers, interrogatives

- MAP and MLE versions of the estimation equations

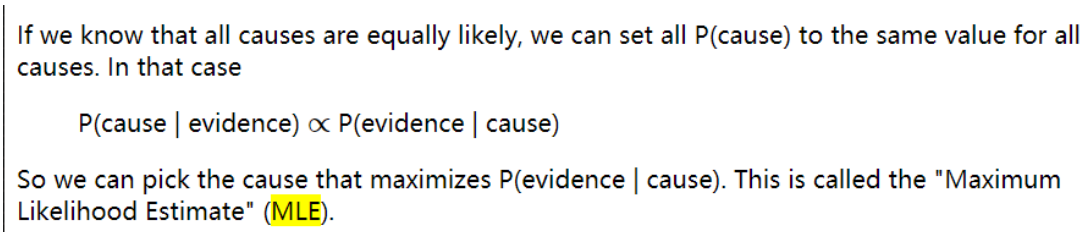


"Maximum a posteriori" (MAP) estimate

pick the type X such that P(X | evidence) is highest

Or, we should pick X from a set of types T such that

$$X = \text{argmax}_{x \in T}\, P(x|\text{evidence})$$



If we know that all causes are equally likely, we can set all P(cause) to the same value for all causes. In that case

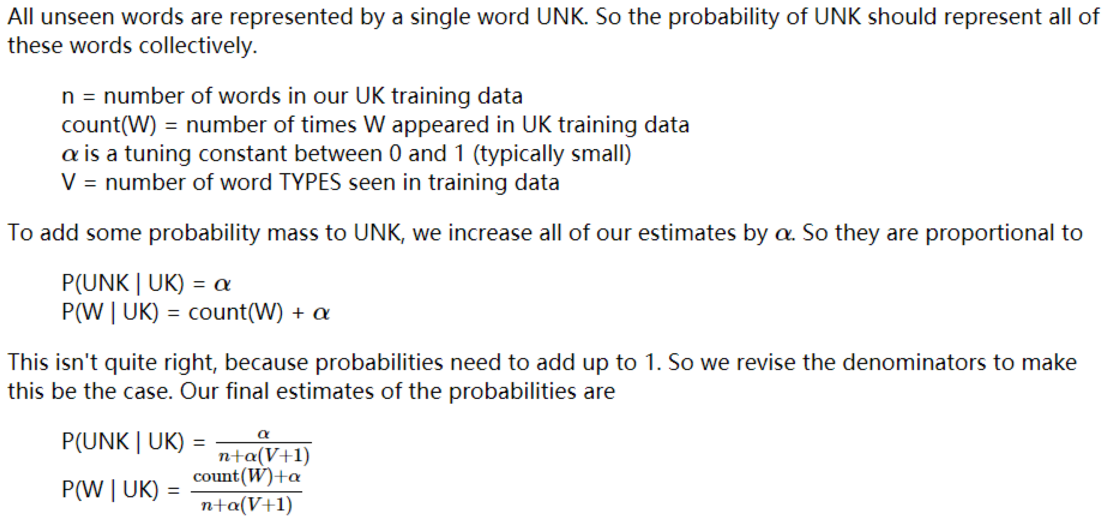
P(cause | evidence) $\propto$ P(evidence | cause)

So we can pick the cause that maximizes P(evidence | cause). This is called the "Maximum Likelihood Estimate" (MLE).

The MLE estimate can be very inaccurate if the probabilities of different causes are actually different and not the same value as was assumed. On the other hand, it can be a sensible choice if we have no information about the prior probabilities or if there is reason the believe P(Cause) should be constant across all possible causes.

- **Avoiding overfitting ---smoothing!!!**

for uncommon words: Words that didn't appear in the training data get estimated zero probability. Words that were uncommon in the training data get inaccurate estimates.



All unseen words are represented by a single word UNK. So the probability of UNK should represent all of these words collectively.

n = number of words in our UK training data
count(W) = number of times W appeared in UK training data
α is a tuning constant between 0 and 1 (typically small)
V = number of word TYPES seen in training data

To add some probability mass to UNK, we increase all of our estimates by α. So they are proportional to

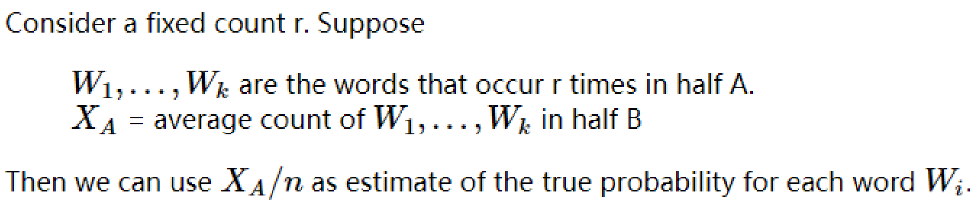
P(UNK | UK) = α
P(W | UK) = count(W) + α

This isn't quite right, because probabilities need to add up to 1. So we revise the denominators to make this be the case. Our final estimates of the probabilities are

P(UNK | UK) = $\frac{\alpha}{n+\alpha(V+1)}$
P(W | UK) = $\frac{\text{count}(W)+\alpha}{n+\alpha(V+1)}$

overestimates probability of unseen words
underestimates probability of common words

- Deleted estimation: **Also called cross-validation or held-out estimation.**

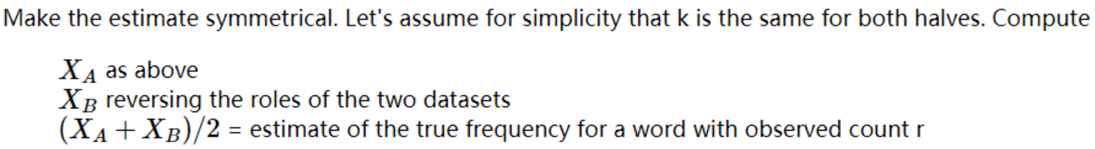


Consider a fixed count r. Suppose

$W_1, \ldots, W_k$ are the words that occur r times in half A.
X_A = average count of $W_1, \ldots, W_k$ in half B

Then we can use X_A/n as estimate of the true probability for each word W_i.

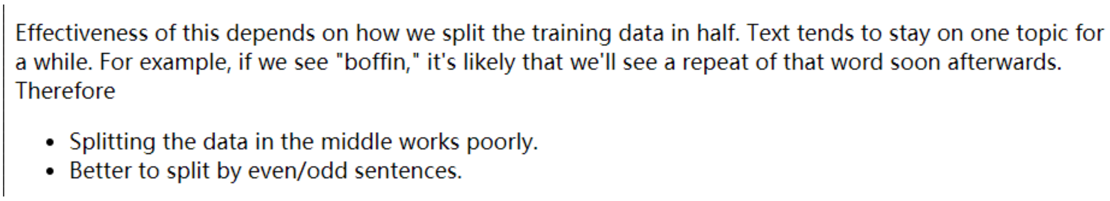
Tweaks on deleted estimation:



Make the estimate symmetrical. Let's assume for simplicity that k is the same for both halves. Compute

X_A as above
X_B reversing the roles of the two datasets
$(X_A + X_B)/2$ = estimate of the true frequency for a word with observed count r

Text tends to stay on one topic for a while.



> Effectiveness of this depends on how we split the training data in half. Text tends to stay on one topic for a while. For example, if we see "boffin," it's likely that we'll see a repeat of that word soon afterwards. Therefore
>
> - Splitting the data in the middle works poorly.
> - Better to split by even/odd sentences.

Performance of Deleted Estimation:

Infrequent words overestimate their true probability. Estimates for common words are usually accurate.

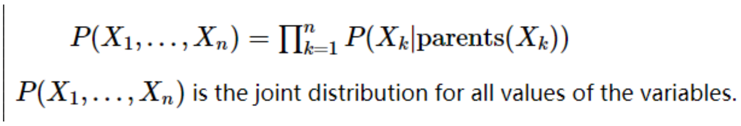
- N-gram smoothing

  Idea 1: If we haven't seen an ngram, **guess its probability from the probabilites of its prefix (e.g. "the angry") and the last word ("armadillo").**

  Idea 2: **Guess that an unseen word is more likely in contexts where we've seen many different words. I.e. some contexts are much more predictable ("hand me the …" can be followed by anything but "I went to mcdonalds and bought a …" is usually followed by "hamburger".**

A high-level point: conditional independence isn't some weakened variant of independence. Neither property implies the other.

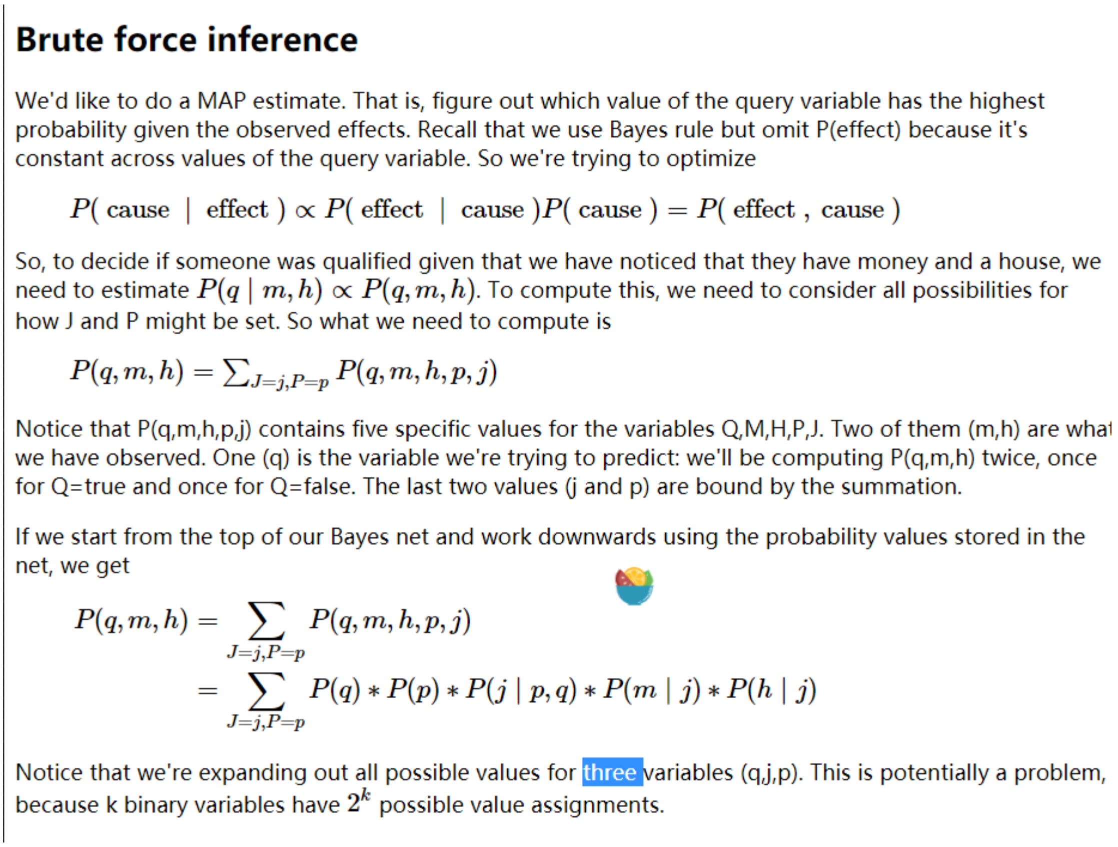
A Bayes net is a directed acyclic (no cycles) graph (called a DAG for short). So nodes are partially ordered by the ancestor/descendent relationships.
Each node is conditionally independent of its ancestors, given its parents. That is, the ancestors can ONLY influence node A by working via parents(A)



$$P(X_1, \ldots, X_n) = \prod_{k=1}^{n} P(X_k|\text{parents}(X_k))$$

$P(X_1, \ldots, X_n)$ is the joint distribution for all values of the variables.

the choice of variable order affects whether we get a better or worse Bayes net representation. We could imagine a learning algorithm that tries to find the best Bayes net, e.g. try different variable orders and see which is most compact. In practice, we usually depend on domain experts to supply the causal structure.



Brute force inference

We'd like to do a MAP estimate. That is, figure out which value of the query variable has the highest probability given the observed effects. Recall that we use Bayes rule but omit P(effect) because it's constant across values of the query variable. So we're trying to optimize

$$P(\text{ cause } \mid \text{ effect }) \propto P(\text{ effect } \mid \text{ cause })P(\text{ cause }) = P(\text{ effect }, \text{ cause })$$

So, to decide if someone was qualified given that we have noticed that they have money and a house, we need to estimate $P(q \mid m, h) \propto P(q, m, h)$. To compute this, we need to consider all possibilities for how J and P might be set. So what we need to compute is

$$P(q,m,h) = \sum_{J=j,P=p} P(q,m,h,p,j)$$

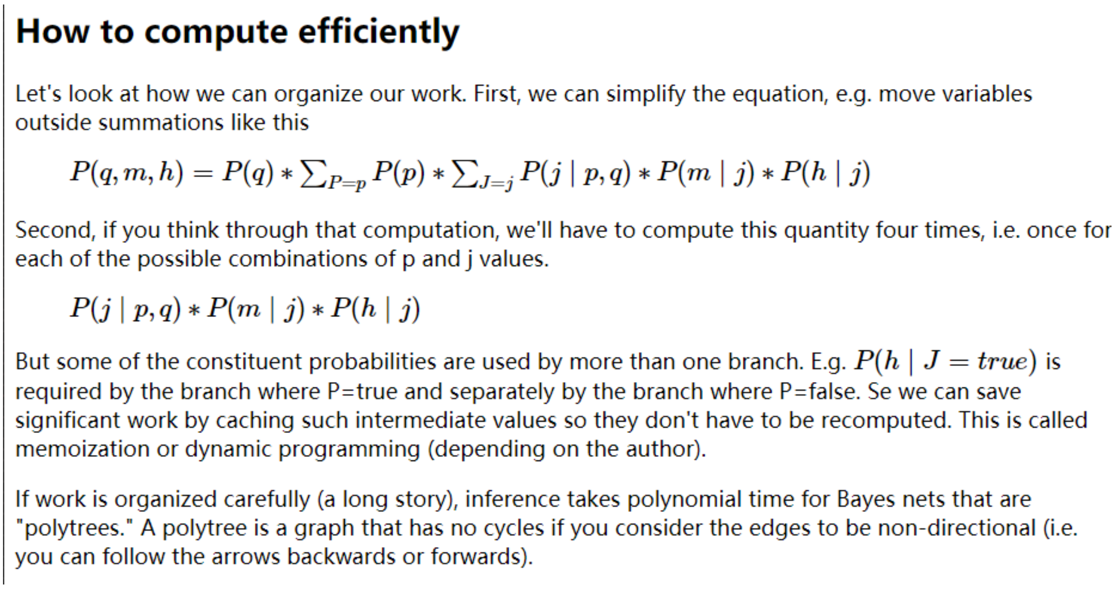
Notice that P(q,m,h,p,j) contains five specific values for the variables Q,M,H,P,J. Two of them (m,h) are what we have observed. One (q) is the variable we're trying to predict: we'll be computing P(q,m,h) twice, once for Q=true and once for Q=false. The last two values (j and p) are bound by the summation.

If we start from the top of our Bayes net and work downwards using the probability values stored in the net, we get

$$\begin{aligned} P(q,m,h) &= \sum_{J=j,P=p} P(q,m,h,p,j) \\ &= \sum_{J=j,P=p} P(q) * P(p) * P(j \mid p,q) * P(m \mid j) * P(h \mid j) \end{aligned}$$

Notice that we're expanding out all possible values for three variables (q,j,p). This is potentially a problem, because k binary variables have 2^k possible value assignments.

Notice that we're expanding out all possible values for three variables (q,j,p). This is potentially a problem, because k binary variables have 2^k possible value assignments.



How to compute efficiently

Let's look at how we can organize our work. First, we can simplify the equation, e.g. move variables outside summations like this

$$P(q,m,h) = P(q) * \sum_{P=p} P(p) * \sum_{J=j} P(j \mid p,q) * P(m \mid j) * P(h \mid j)$$

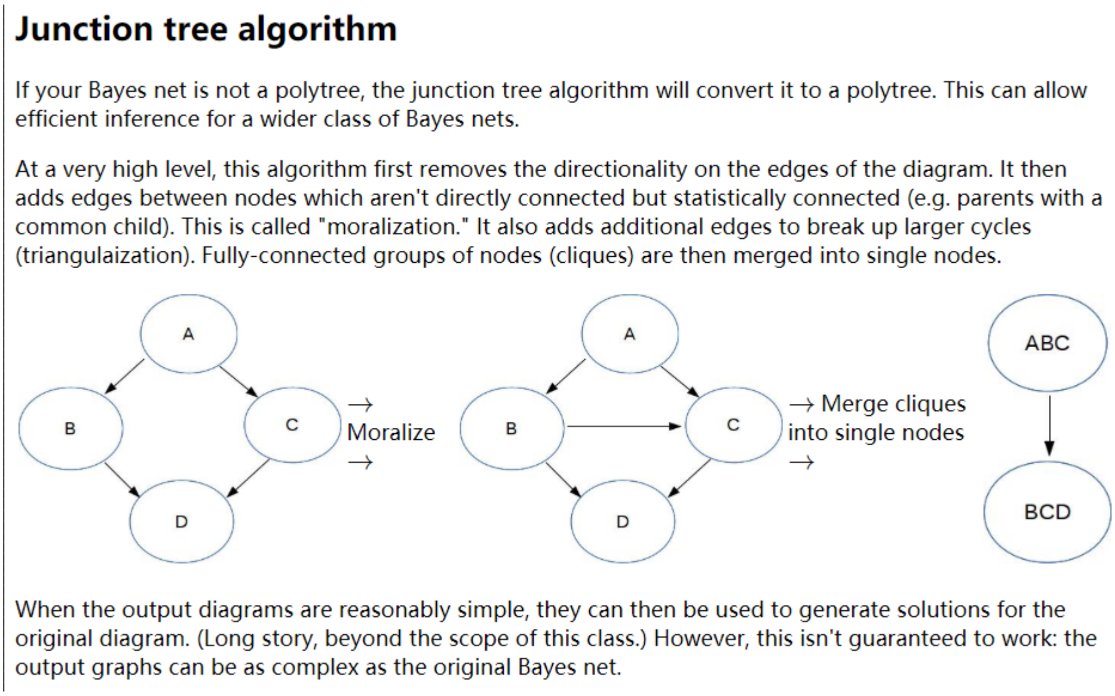
Second, if you think through that computation, we'll have to compute this quantity four times, i.e. once for each of the possible combinations of p and j values.

$$P(j \mid p,q) * P(m \mid j) * P(h \mid j)$$

But some of the constituent probabilities are used by more than one branch. E.g. $P(h \mid J = true)$ is required by the branch where P=true and separately by the branch where P=false. Se we can save significant work by caching such intermediate values so they don't have to be recomputed. This is called memoization or dynamic programming (depending on the author).

If work is organized carefully (a long story), inference takes polynomial time for Bayes nets that are "polytrees." A polytree is a graph that has no cycles if you consider the edges to be non-directional (i.e. you can follow the arrows backwards or forwards).

Bayes nets can be computed in polynomial time when there are polytrees - which means that when you remove the direction from the graph you're still left with no cycles



Junction tree algorithm

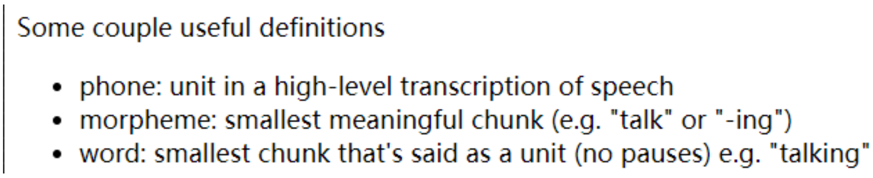
If your Bayes net is not a polytree, the junction tree algorithm will convert it to a polytree. This can allow efficient inference for a wider class of Bayes nets.

At a very high level, this algorithm first removes the directionality on the edges of the diagram. It then adds edges between nodes which aren't directly connected but statistically connected (e.g. parents with a common child). This is called "moralization." It also adds additional edges to break up larger cycles (triangulaization). Fully-connected groups of nodes (cliques) are then merged into single nodes.

When the output diagrams are reasonably simple, they can then be used to generate solutions for the original diagram. (Long story, beyond the scope of this class.) However, this isn't guaranteed to work: the output graphs can be as complex as the original Bayes net.

At a very high level, this algorithm first removes the directionality on the edges of the diagram. It then adds edges between nodes which aren't directly connected but statistically connected (e.g. parents with a common child). This is called "moralization." It also adds additional edges to break up larger cycles (triangulaization). Fully-connected groups of nodes (cliques) are then merged into single nodes.

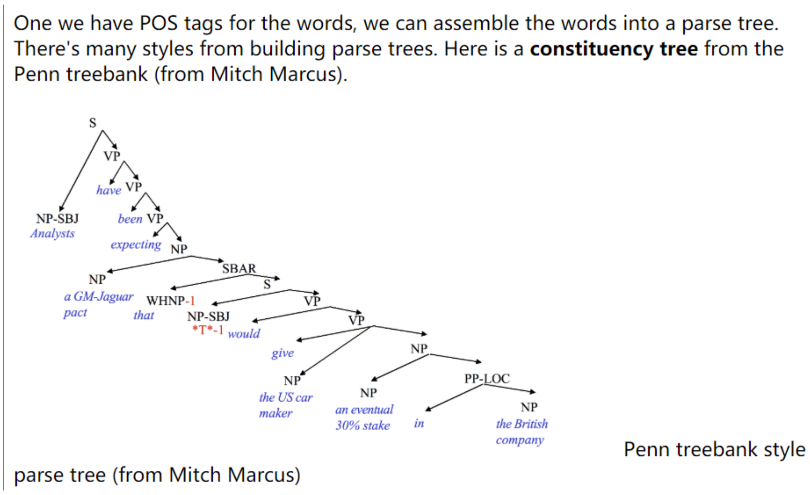
NLP



Some couple useful definitions

- phone: unit in a high-level transcription of speech
- morpheme: smallest meaningful chunk (e.g. "talk" or "-ing")
- word: smallest chunk that's said as a unit (no pauses) e.g. "talking"

constituency tree & dependency tree

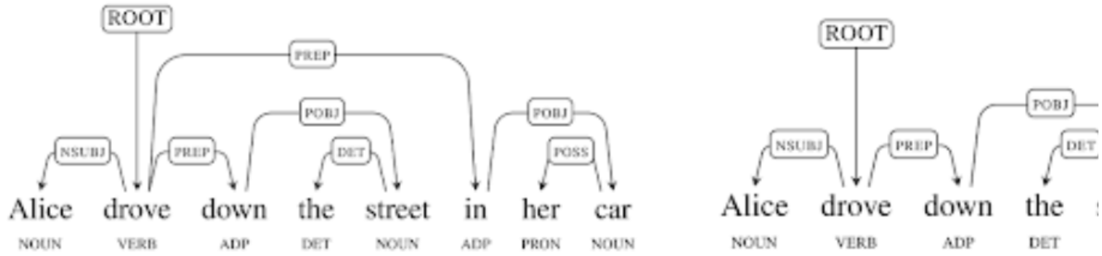


One we have POS tags for the words, we can assemble the words into a parse tree. There's many styles from building parse trees. Here is a **constituency tree** from the Penn treebank (from Mitch Marcus).

Penn treebank style

parse tree (from Mitch Marcus)

An alternative is a **dependency tree** like the following ones below from Google labs. A fairly recent parser from them is called "Parsey McParseface," after the UK's Boaty McBoatface shown above.



Parsers fall into two categories

- Unlexicalized: use only the POS tags to build the tree.
- Word classes: beyond the part of speech, identify the general type of object or action (e.g. a person vs. a vehicle)
- Lexicalized: also include some information about word identity/meaning

baseline algorithm might pick the most common tag for each word, ignoring context. In the table above, the most common tag for each word is shown in bold: it made one error out of six words.

The test/development data will typically contain some words that weren't seen in the training data. The baseline algorithm guesses that these are nouns, since that's the most common tag for infrequent words. An algorithm can figure this out by examining new words that appear in the development set, or by examining the set of words that occur only once in the training data ("hapax") words.

Hapax： words only occurs once

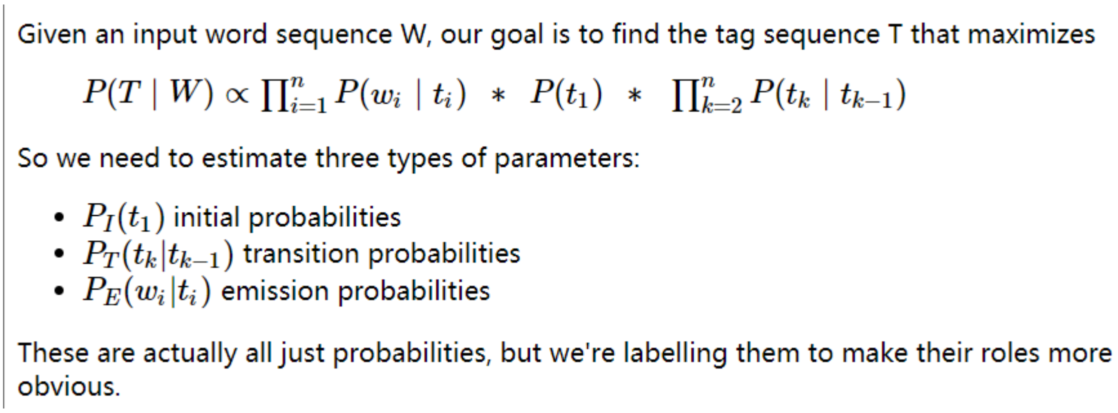
A Markov assumption is that the value of interest depends only on a short window of context.

$P(w_k)$ depends only on $P(t_k)$
$P(t_k)$ depends only on $P(t_{k-1}$

Therefore

$$P(W \mid T) = \prod_{i=1}^{n} P(w_i \mid t_i)$$
$$P(T) = P(t_1|START) * \prod_{i=1}^{n} P(t_k \mid t_{k-q})$$

where $P(t_1|START)$ is the probability of tag t_1 at the start of a sentence.



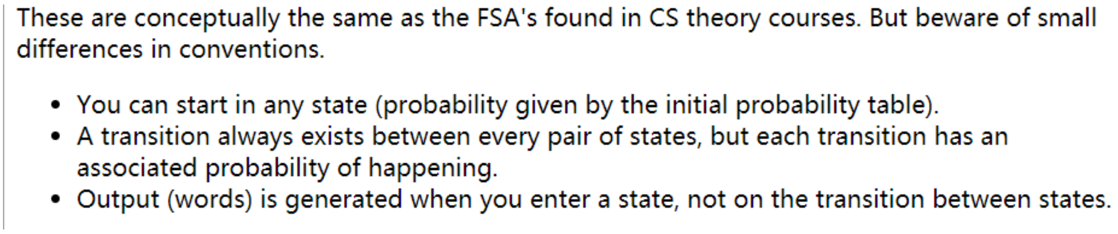
Given an input word sequence W, our goal is to find the tag sequence T that maximizes

$$P(T \mid W) \propto \prod_{i=1}^{n} P(w_i \mid t_i) \ * \ P(t_1) \ * \ \prod_{k=2}^{n} P(t_k \mid t_{k-1})$$

So we need to estimate three types of parameters:

- $P_I(t_1)$ initial probabilities
- $P_T(t_k|t_{k-1})$ transition probabilities
- $P_E(w_i|t_i)$ emission probabilities

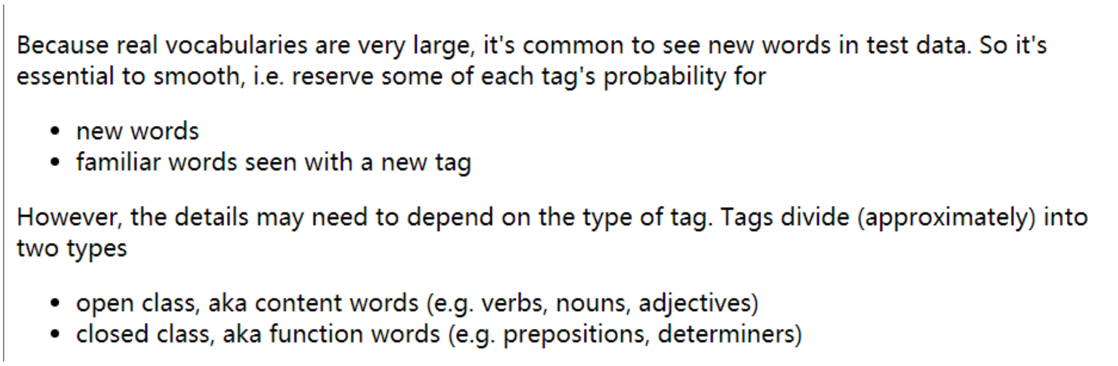
These are actually all just probabilities, but we're labelling them to make their roles more obvious.



These are conceptually the same as the FSA's found in CS theory courses. But beware of small differences in conventions.

- You can start in any state (probability given by the initial probability table).
- A transition always exists between every pair of states, but each transition has an associated probability of happening.
- Output (words) is generated when you enter a state, not on the transition between states.

Output (words) is generated when you enter a state, not on the transition between states.



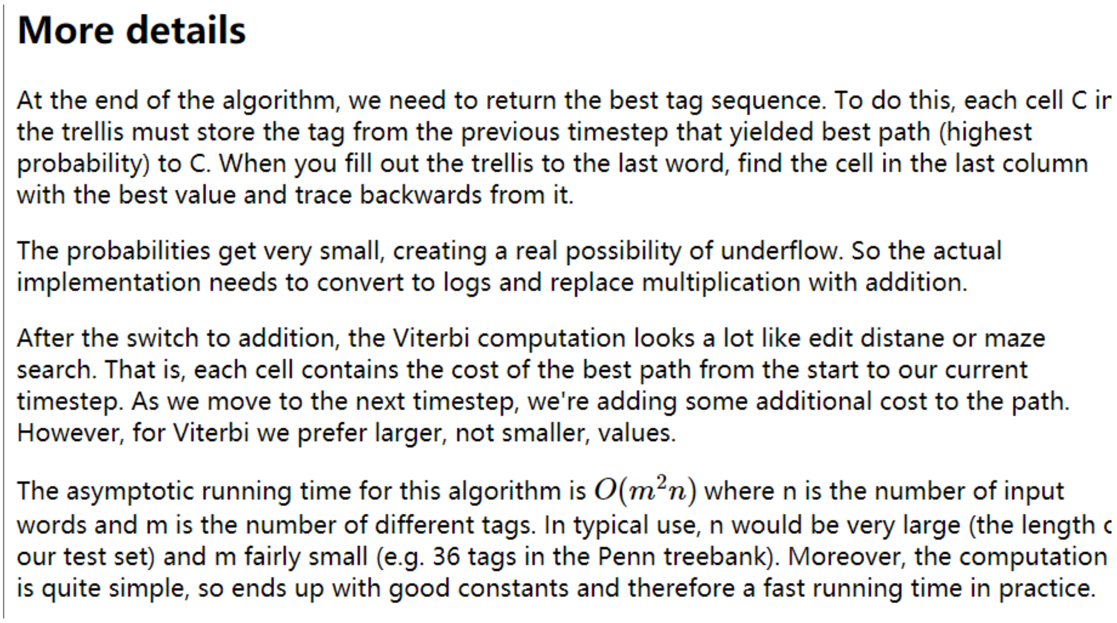
Because real vocabularies are very large, it's common to see new words in test data. So it's essential to smooth, i.e. reserve some of each tag's probability for

- new words
- familiar words seen with a new tag

However, the details may need to depend on the type of tag. Tags divide (approximately) into two types

- open class, aka content words (e.g. verbs, nouns, adjectives)
- closed class, aka function words (e.g. prepositions, determiners)

Smoothing for emission probabilities needs to take these differences into account. Suppose that use Laplace smoothing, separately for each tag, to fill in missing emission probabilities. Then each tag will need a different Laplace constant, i.e. a larger Laplace constant for tags that are more likely to generate unseen words.



More details

At the end of the algorithm, we need to return the best tag sequence. To do this, each cell C ir the trellis must store the tag from the previous timestep that yielded best path (highest probability) to C. When you fill out the trellis to the last word, find the cell in the last column with the best value and trace backwards from it.

The probabilities get very small, creating a real possibility of underflow. So the actual implementation needs to convert to logs and replace multiplication with addition.

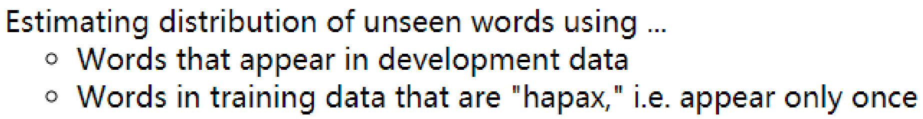
After the switch to addition, the Viterbi computation looks a lot like edit distane or maze search. That is, each cell contains the cost of the best path from the start to our current timestep. As we move to the next timestep, we're adding some additional cost to the path. However, for Viterbi we prefer larger, not smaller, values.

The asymptotic running time for this algorithm is $O(m^2n)$ where n is the number of input words and m is the number of different tags. In typical use, n would be very large (the length c our test set) and m fairly small (e.g. 36 tags in the Penn treebank). Moreover, the computation is quite simple, so ends up with good constants and therefore a fast running time in practice.

At what points does an HMM need smoothing?

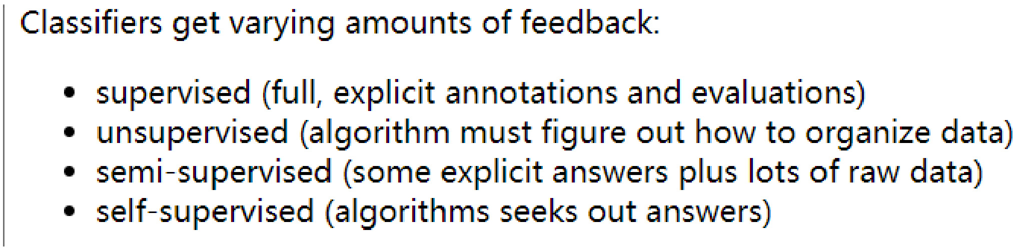
near-zero probability of a transition between tags e.g. from Det to Verb

emission probabilities, also need consider open class & closed class



Estimating distribution of unseen words using ...

- Words that appear in development data
- Words in training data that are "hapax," i.e. appear only once



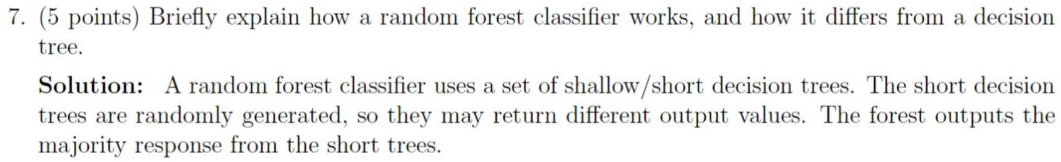
Classifiers get varying amounts of feedback:

- supervised (full, explicit annotations and evaluations)
- unsupervised (algorithm must figure out how to organize data)
- semi-supervised (some explicit answers plus lots of raw data)
- self-supervised (algorithms seeks out answers)

Decision trees

each node in the tree represents a pool of examples. Ideally, each leaf node contains only objects of the same type. However, that isn't always true if the classes aren't cleanly distinguished by the available features (see the vowel picture above). So, when we reach a leaf node, the algorithm returns the most frequent label in its pool.

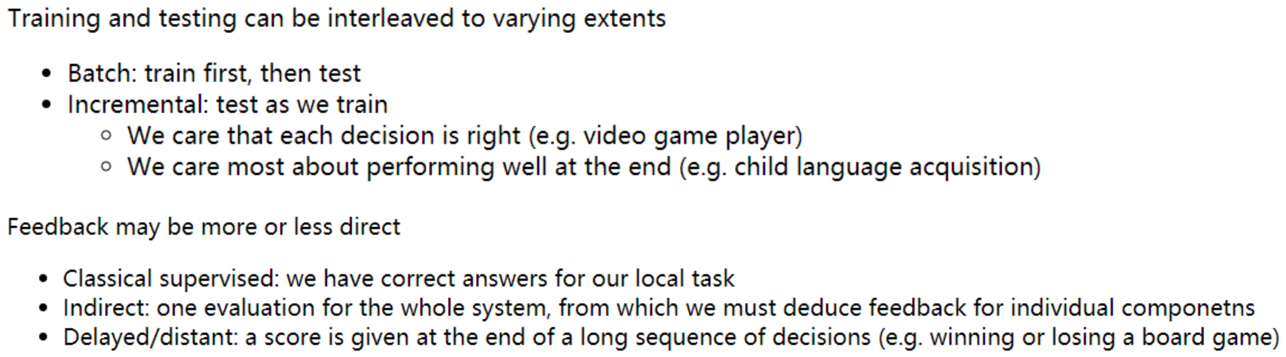
If we have enough decision levels in our tree, we can force each pool to contain objects of only one type. However, this is not a wise decision. Deep trees are difficult to construct (see the algorithm in Russell and Norvig) and prone to overfitting their training data. It is more effective to create a "forest" of shallow trees (bounded depth) and have these trees vote on the best label to return. **The set of shallow trees is created by choosing variables and possible split locations randomly.**



7. (5 points) Briefly explain how a random forest classifier works, and how it differs from a decision tree.

Solution: A random forest classifier uses a set of shallow/short decision trees. The short decision trees are randomly generated, so they may return different output values. The forest outputs the majority response from the short trees.

A good split should produce subpools that are less diverse than the original pool. If we're choosing between two candidate splits (e.g. two thresholds for the same numerical variable), it's better to pick the one that produces subpools that are more uniform. **Diversity/uniformity is measured using entropy.**



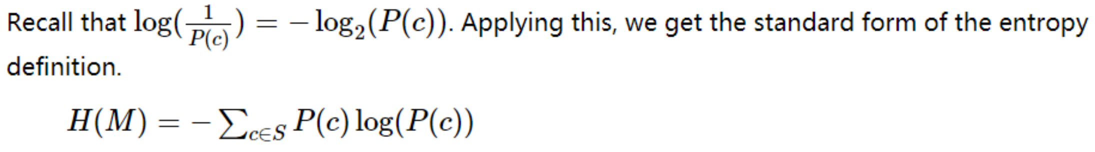
Training and testing can be interleaved to varying extents

- Batch: train first, then test
- Incremental: test as we train
  - We care that each decision is right (e.g. video game player)
  - We care most about performing well at the end (e.g. child language acquisition)

Feedback may be more or less direct

- Classical supervised: we have correct answers for our local task
- Indirect: one evaluation for the whole system, from which we must deduce feedback for individual componetns
- Delayed/distant: a score is given at the end of a long sequence of decisions (e.g. winning or losing a board game)

A good split should produce subpools that are less diverse than the original pool. If we're choosing between two candidate splits (e.g. two thresholds for the same numerical variable), it's better to pick the one that produces subpools that are more uniform. **Diversity/uniformity is measured using entropy.**



Recall that $\log(\frac{1}{P(c)}) = -\log_2(P(c))$. Applying this, we get the standard form of the entropy definition.

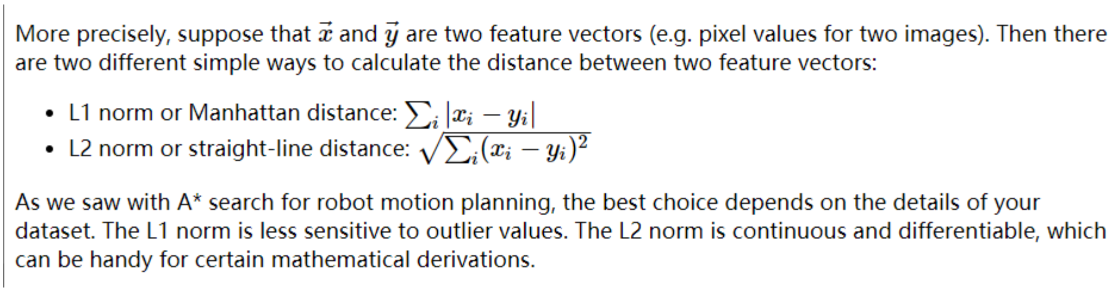
$$H(M) = -\sum_{c\in S} P(c)\log(P(c))$$

Where P(c) is the possible of letter c occurs in set S

K nearest neighbors

Another way to classify an input example is to find the most similar training example and copy its label.

This "nearest neighbor" method is prone to overfitting, especially if the data is messy and so examples from different classes are somewhat intermixed. However, performance can be greatly improved by finding the k nearest neighbors for some small value of k (e.g. 5-10).

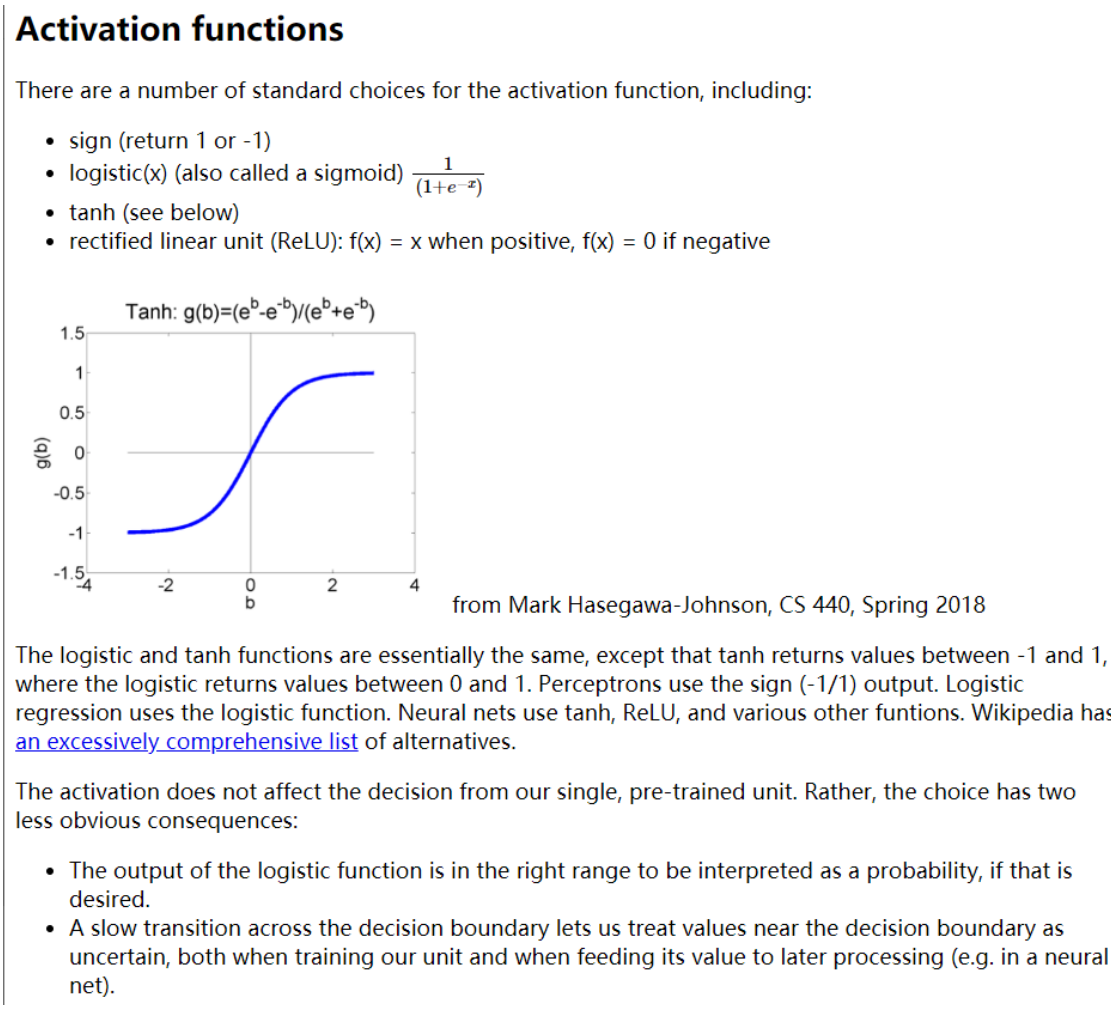


More precisely, suppose that $\vec{x}$ and $\vec{y}$ are two feature vectors (e.g. pixel values for two images). Then there are two different simple ways to calculate the distance between two feature vectors:

- L1 norm or Manhattan distance: $\sum_i |x_i - y_i|$
- L2 norm or straight-line distance: $\sqrt{\sum_i (x_i - y_i)^2}$

As we saw with A* search for robot motion planning, the best choice depends on the details of your dataset. The L1 norm is less sensitive to outlier values. The L2 norm is continuous and differentiable, which can be handy for certain mathematical derivations.

The L1 norm is less sensitive to outlier values. The L2 norm is continuous and differentiable, which can be handy for certain mathematical derivations.



Activation functions

There are a number of standard choices for the activation function, including:

- sign (return 1 or -1)
- logistic(x) (also called a sigmoid) $\frac{1}{(1+e^{-x})}$
- tanh (see below)
- rectified linear unit (ReLU): f(x) = x when positive, f(x) = 0 if negative

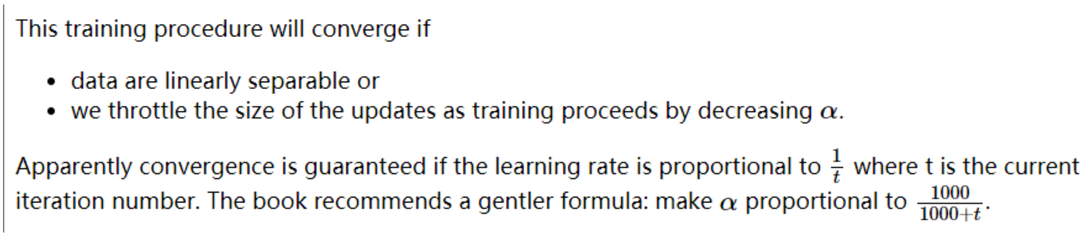
from Mark Hasegawa-Johnson, CS 440, Spring 2018

The logistic and tanh functions are essentially the same, except that tanh returns values between -1 and 1, where the logistic returns values between 0 and 1. Perceptrons use the sign (-1/1) output. Logistic regression uses the logistic function. Neural nets use tanh, ReLU, and various other funtions. Wikipedia has an excessively comprehensive list of alternatives.

The activation does not affect the decision from our single, pre-trained unit. Rather, the choice has two less obvious consequences:

- The output of the logistic function is in the right range to be interpreted as a probability, if that is desired.
- A slow transition across the decision boundary lets us treat values near the decision boundary as uncertain, both when training our unit and when feeding its value to later processing (e.g. in a neural net).

The output of logistic function is in the right range to be interpreted as a probability
A slow transition across the decision boundary lets us treat values near the boundary as uncertain.



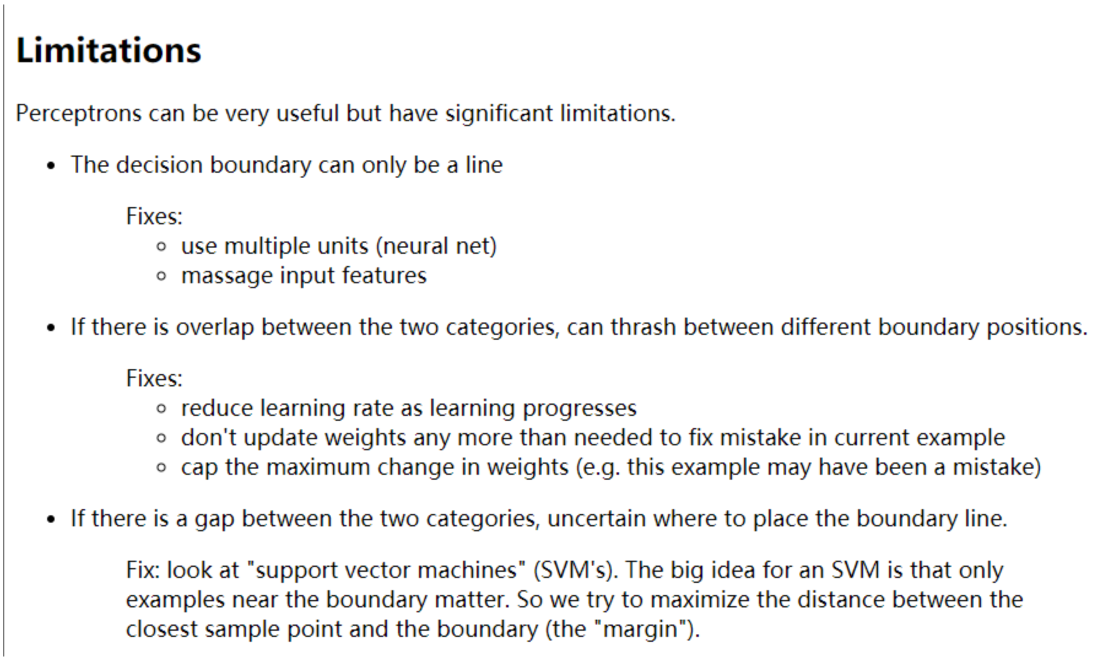
This training procedure will converge if

- data are linearly separable or
- we throttle the size of the updates as training proceeds by decreasing α.

Apparently convergence is guaranteed if the learning rate is proportional to $\frac{1}{t}$ where t is the current iteration number. The book recommends a gentler formula: make α proportional to $\frac{1000}{1000+t}$.

One run through training data through the update rule is called an "epoch."
One generally uses multiple epochs, i.e. the algorithm sees each training pair several times.

Datasets often have strong local correlations, because they have been formed by concatenating sets of data (e.g. documents) that each have a strong topic focus. **So it's often best to process the individual training examples in a random order.**



Limitations

Perceptrons can be very useful but have significant limitations.

- The decision boundary can only be a line

  Fixes:
  - use multiple units (neural net)
  - massage input features

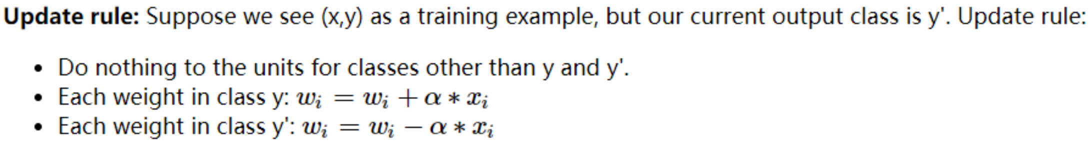
- If there is overlap between the two categories, can thrash between different boundary positions.

  Fixes:
  - reduce learning rate as learning progresses
  - don't update weights any more than needed to fix mistake in current example
  - cap the maximum change in weights (e.g. this example may have been a mistake)

- If there is a gap between the two categories, uncertain where to place the boundary line.

  Fix: look at "support vector machines" (SVM's). The big idea for an SVM is that only examples near the boundary matter. So we try to maximize the distance between the closest sample point and the boundary (the "margin").

Multi-class perceptrons



Update rule: Suppose we see (x,y) as a training example, but our current output class is y'. Update rule:

- Do nothing to the units for classes other than y and y'.
- Each weight in class y: $w_i = w_i + \alpha * x_i$
- Each weight in class y': $w_i = w_i - \alpha * x_i$