UiO : Department of Informatics
University of Oslo

INF4820

Algorithms for artificial intelligence and natural language processing

Joakim Myrvoll

2014

Contents

Part I
Point-wise categorization (geometric models)

Linear Structures

1 The Distributional Hypothesis

AKA The Contextual Theory of Meaning

The hypothesis: If two words share similar contexts, we can assume that they have similar meanings. Comparing meaning reduced to comparing contexts - no need for prior knowledge!

1.1 Context

- Neighborhood of $\pm n$ words left/right of the focus word.
- Bag-of-Words; include all co-occurring words, ignoring the linear ordering
- Grammatical context; the grammatical relations to other words.

1.2 Definition of a word

Raw:	The programmer's programs had been programmed.
Tokenized:	the programmer 's programs had been programmed.
Lemmatized:	the programmer 's program have be program.
W/ stop-list:	programmer program program
Stemmed:	program program program

Tokenization:
Splitting a text into sentences and words or other units.

Different levels of abstraction and morphological normalization:

- What to do with case, numbers, punctuation, compounds, . . . ?
- Full-form words vs. lemmas vs. stems . . .

Stop-list:
filter out closed-class words or function words. The idea is that only content words provide relevant context.

2 Vector Space Models

A general model for representing data based on a spatial metaphor. Often both sparse (low ratio of non-zero elements) and extremely high-dimensional

Each object is represented as a vector (or point) positioned in a coordinate system. Each coordinate (or dimension) of the space corresponds to some descriptive and measurable property (feature) of the objects.
To measure similarity of two objects, we can measure their geometrical distance/closeness in the model. Vector representations are foundational to a wide range of ML methods. There are multiple choices of data structures;

- The Euclidean norm or length of a vector $\vec{x}$ is defined as follows:

$$||\vec{x}|| = \sqrt{\sum_{i=1}^{n} \vec{x_i^2}}$$

- The cosine measure is defined as:

$$cos(\vec{x}, \vec{y}) = \frac{\sum_i \vec{x_i}\vec{y_i}}{\sqrt{\vec{x_i^2}}\sqrt{\vec{y_i^2}}} = \frac{\vec{x} \cdot \vec{y}}{||\vec{x}||||\vec{y}||}$$

It is often desirable to work with so-called unit vectors or length normalized vectors. One important reason for this is that we want to reduce bias effects caused by e.g. skewed frequency distributions. It also makes it possible to compute similarity functions such as the cosine much more efficiently. A vector has unit length if its Euclidean norm is 1:

- Euclidean distance:

$$||\vec{x}|| = \sqrt{\sum_{i=1}^{n} \vec{x_i^2}} = \sum_{i=1}^{n} \vec{x_i^2} = 1$$

- Cosine measure:

$$\vec{x} \cdot \vec{y} = \sum_{i=1}^{n} \vec{x_i}\vec{y_i}$$

2.1 Semantic spaces

AKA distributional semantic models or word space models.

A semantic space is a vector space model where

- points represent words
- dimensions represent context of use
- distance in the space represents semantic similarity

3 Vector space classification

Vector space classification is based on *the contiguity hypothesis*:

- Objects in the same class form a contiguous region, and regions of different classes do not overlap.
- Classification amounts to computing the boundaries in the space that separate the classes; the decision boundaries.
- How we draw the boundaries is influenced by how we choose to represent the classes.

3.1 Representing classes

3.1.1 Exemplar-based

- No abstraction. Every stored instance of a group can potentially represent the class.
- Used in so-called instance based or memory based learning (MBL).
- In its simplest form; the class = the collection of points.
- Another variant is to use medoids - representing a class by a single member that is considered central, typically the object with maximum average similarity to other objects in the group.

3.1.2 Centroid-based

- The average or center of mass in the region.
- Given a class c_i, where each object o_j being a member is represented as a feature vector $\vec{x_j}$, we can compute the class centroid $\vec{\mu_i}$ as:

$$\vec{\mu_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \vec{x_j}$$

3.1.3 Typicality

Centroids and medoids both represent a group of objects by a single point, a *prototype*. But while a medoid is an actual member of the group, a centroid is an abstract prototype; an average. The typicality of class members can be determined by their distance to the prototype.

The centroid could also be distance weighted; let each member's contribution to the average be determined by its average pairwise similarity to the other members of the group.

3.2 Representing class membership

3.2.1 Hard Classes

- Membership considered a Boolean property: a given object is either part of the class or it is not.
- A crisp membership function.
- A variant: disjunctive classes. Objects can be members of more than one class, but the memberships are still crisp.

3.2.2 Soft Classes

- Class membership is a graded property.
- Probabilistic. The degree of membership for a given restricted to [0, 1], and the sum across classes must be 1.
- Fuzzy: The membership function is still restricted to [0, 1], but without the probabilistic constraint on the sum.

3.2.3 Examples

- Named Entity Recognition:
  - Recognize Entities
  - Assign them a class (ex. Person Location and Organization)
- Sentiment Analysis:
  - Classify Sentences into classes Positive, Negative Neutral
- Textual Entailment:
  - Classify pair of sentences A and B into 2 classes: YES (A implies B) and NO (A does not imply B)

4 Classification

- Supervised learning, requiring labeled training data.
- Train a classifier to automatically assign new instances to predefined classes, given some set of examples.
- Two examples of classifiers that use a vector space representation: Rocchio and kNN.

4.1 Rocchio

Uses centroids to represent classes. Each class c_i is represented by its centroid $\vec{\mu_i}$, computed as the average of the normalized vectors $\vec{x_j}$ of its members;

$$\vec{\mu_i} = \frac{1}{|c_i|} \sum_{\vec{x_j} \in c_i} \vec{x_j}$$

To classify a new object o_j (represented by a feature vector $\vec{x_j}$);
- determine which centroid $\vec{\mu_i}$ that $\vec{x_j}$ is closest to,
- and assign it to the corresponding class c_i.

The centroids define the boundaries of the class regions. Defines the boundary between two classes by the set of points equidistant from the centroids.

4.1.1 Problems

Ignores details of the distribution of points within a class, only based on the centroid distance. Implicitly assumes that classes are spheres with similar radio. Does not work well for classes than cannot be accurately represented by a single prototype or "center" (e.g. disconnected or elongated regions). Because the Rocchio classifier defines a linear decision boundary, it is only suitable for problems involving linearly separable classes.

4.2 kNN

k Nearest Neighbor classification

- For k = 1: Assign each object to the class of its closest neighbor.
- For k > 1: Assign each object to the majority class among its k closest neighbors.

Rationale:
given *the contiguity hypothesis*, we expect a test object o_i to have the same label as the training objects located in the local region surrounding $\vec{x_i}$. The parameter k must be specified in advance, either manually or by optimizing on held-out data.

4.2.1 A non-linear classifier

Unlike Rocchio, the kNN decision boundary is determined locally. The decision boundary is defined by the Voronoi tessellation.

4.2.2 Voronoi tessellation

Assuming k = 1: For a given set of objects in the space, let each object define a cell consisting of all points that are closer to that object than to other objects. Results in a set of convex polygons; so-called *Voronoi cells*. Decomposing a space into such cells gives us the so-called *Voronoi tessellation*. In the general case of k $\geq$ 1, the Voronoi cells are given by the regions in the space for which the set of k nearest neighbors is the same.

4.2.3 "Softened" kNN-classification

4.2.3.1 A probabilistic version
Estimate the probability of membership in class c as the proportion of the k nearest neighbors in c.

4.2.3.2 A distance weighted version
The score for a given class c_i can be computed as:

$$score(c_i, o_j) = \sum_{\vec{x_n} \in knn(\vec{x_j})} I(c_i, \vec{x_n}) sim(\vec{x_n}, \vec{x_j})$$

where $knn(\vec{x_j})$ is the set of k nearest neighbors of $\vec{x_j}$, sim is whatever similarity measure we're using, and $I(c_i, \vec{x_n})$ is simply a membership function returning 1 if $\vec{x_n} \in c_i$ and 0 otherwise. Such distance weighted votes can often give more accurate results, and also help resolve ties.

4.2.4 Some peculiarities of kNN

- Not really any learning or estimation going on at all; simply memorizes all training examples.
- Example of so-called memory-based learning or instance-based learning.
- In general in machine learning, the more training data the better. But for kNN, large training sets comes with an efficiency penalty in classification.
- Test time is linear in the size of the training set, and independent of the number of classes.
- A potential advantage for problems with many classes.

4.3 Testing a classifier

We've seen how vector space classification amounts to computing the boundaries in the space that separate the class regions; the decision boundaries. To evaluate the boundary, we measure the number of correct classification predictions on unseeen test items. We want to test how well a model generalizes on a held-out test set (Or, if we have little data, by n-fold cross-validation). Labeled test data is sometimes refered to as the gold standard.

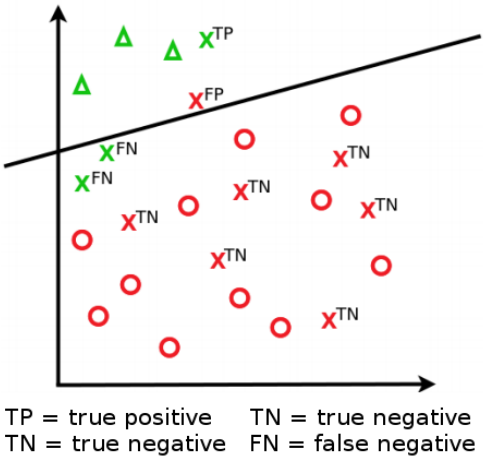


Figure 1:

- accuracy = $\frac{TP+TN}{N} = \frac{TP+TN}{TP+TN+FP+FN}$
  - The ratio of correct predictions.
  - Not suitable for unbalanced numbers of positive/negative examples
- precision = $\frac{TP}{TP+FP}$
  - The number of detected class members that were correct.
- recall = $\frac{TP}{TP+FN}$
  - The number of actual class members that were detected.
  - Trade-off: Positive predictions for all examples would give 100% recall but (typically) terrible precision.
- F-score = $\frac{2recision \times recall}{precision+recall}$
  - Balanced measure of precision and recall (harmonic mean).

4.4 Evaluating multi-class predictions

Macro-averaging
Sum precision and recall for each class, and then compute global averages of these. The macro average will be highly influenced by the small classes.

Micro-averaging
Sum TPs, FPs, and FNs for all points/objects across all classes, and then compute global precision and recall. The micro average will be highly influenced by the large classes.

5 Clustering

- Unsupervised learning from unlabeled data.
- Automatically group similar objects together.
- No predefined classes or structure, we only specify the similarity measure. Relies on "self-organization".
- General objective:
  Partition the data into subsets, so that the similarity among members of the same group is high (homogeneity) while the similarity between the groups themselves is low (heterogeneity).

5.1 Example applications of cluster analysis

- Visualization and exploratory data analysis.

- Many applications within IR (information retrieval). Examples:
  - Speed up search: First retrieve the most relevant cluster, then retrieve documents from within the cluster.
  - Presenting the search results: Instead of ranked lists, organize the results as clusters.
- Dimensionality reduction / class-based features.
- News aggregation / topic directories.
- Social network analysis; identify sub-communities and user segments.
- Image segmentation, product recommendations, demographic analysis,

5.2 Types of clustering methods

Different methods can be divided according to the memberships they create and the procedure by which the clusters are formed:

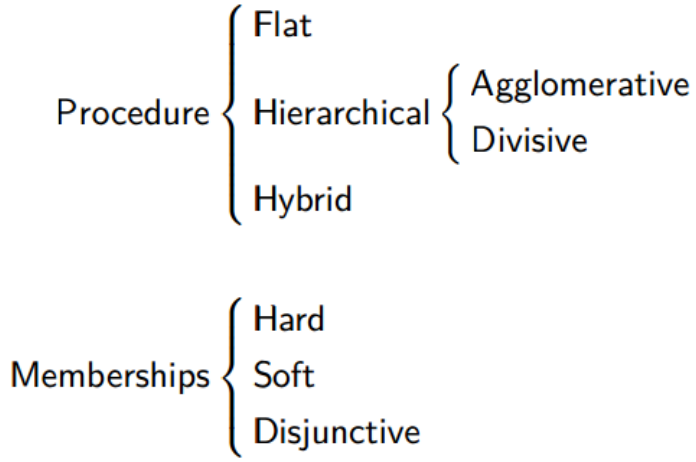


Figure 2:

5.2.1 Hierarchical clustering

Creates a tree structure of hierarchically nested clusters.

5.2.1.1 Agglomerative (bottom-up)

Let each object define its own cluster; then successively merge most similar clusters until only one remains.

- Initially; regards each object as its own singleton cluster.
- Iteratively "agglomerates" (merges) the groups in a bottom-up fashion.
- Each merge defines a binary branch in the tree.
- Terminates; when only one cluster remains (the root).
- At each stage, we merge the pair of clusters that are most similar, as defined by some measure of inter-cluster similarity; sim.

- Plugging in a different sim gives us a different sequence of merges T.

parameters: $\{o_1, o_2, \ldots, o_n\}$, sim
C = $\{\{o_1\}, \{o_2\}, \ldots, \{o_n\}\}$
T = [ ]

do for i = 1 to n - 1
    $\{c_j, c_k\} \leftarrow argmax_{\{c_j,c_k\} \subseteq C \wedge j \neq k}$ sim(c_j, c_k)
    C $\leftarrow$ C $\{c_j, c_k\}$
    C $\leftarrow$ C $\cup \{c_j \cup c_k\}$
    T[i] $\leftarrow \{c_j, c_k\}$

Dendrogram

A hierarchical clustering is often visualized as a binary tree structure known as a dendrogram. A merge is shown as a horizontal line. The y-axis corresponds to the similarity of the merged clusters. We here assume dot-products of normalized vectors (self-similarity = 1).

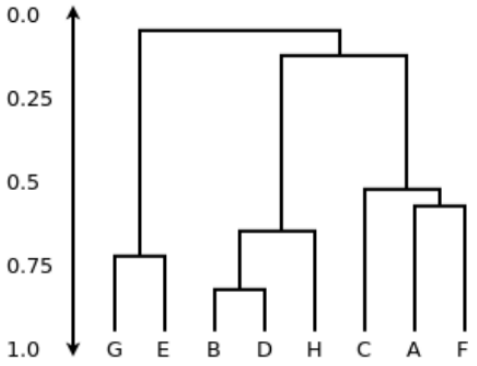


Figure 3: Dendrogram

Inversions

Centroid-linkage is non-monotonic. We risk seeing so-called inversions: similarity can increase during the sequence of clustering steps. Would show as crossing lines in the dendrogram. The horizontal merge bar is lower than the bar of a previous merge.

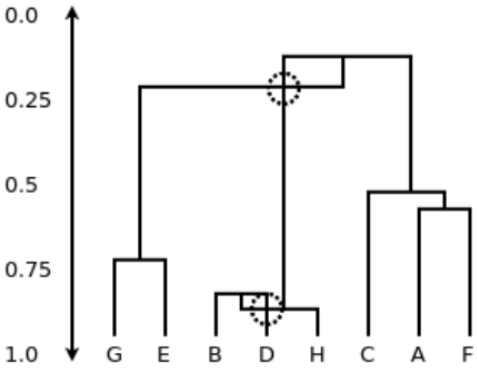


Figure 4:

Cutting the tree

The tree actually represents several partitions; one for each level. If we want to turn the nested partitions into a

single flat partitioning we must cut the tree. A cutting criterion can be defined as a threshold on e.g. combination similarity, relative drop in the similarity, number of root nodes, etc.

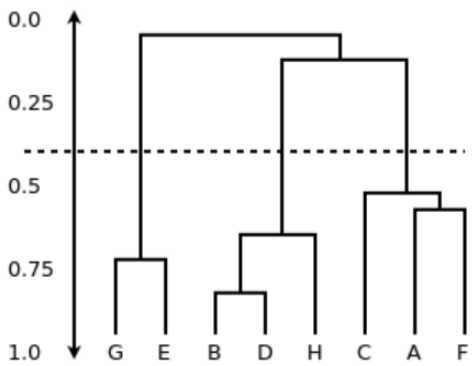


Figure 5:

5.2.1.2 Divisive (top-down)

Let all objects be members of the same cluster; then successively split the group into smaller and maximally dissimilar clusters until all objects is its own singleton cluster.

Generates the nested partitions top-down:

- Start: all objects considered part of the same cluster (the root).
- Split the cluster using a flat clustering algorithm (e.g. by applying k-means for k = 2).
- Recursively split the clusters until only singleton clusters remain (or some specified number of levels is reached).

Flat methods are generally very effective (e.g. k-means is linear in the number of objects). Divisive methods are thereby also generally more efficient than agglomerative, which are at least quadratic (single-link). Also able to initially consider the global distribution of the data, while the agglomerative methods must commit to early decisions based on local patterns.

5.2.2 Flat clustering

Often referred to as partitional clustering when assuming hard and disjoint clusters (but can also be soft). Tries to directly decompose the data into a set of clusters.

Given a set of objects $O = o_1, \ldots, o_n$, construct a set of clusters $C = c_1, \ldots, c_k$, where each object o_i is assigned to a cluster c_i.

Parameters:

- The cardinality k (the number of clusters).
- The similarity function s.

More formally, we want to define an assignment $\gamma : O \rightarrow C$ that optimizes some objective function $F_s(\gamma)$.

In general terms, we want to optimize for:

- High intra-cluster similarity
- Low inter-cluster similarity

Optimization problems are search problems
There's a finite number of possible partitionings of O. Naive solution: enumerate all possible assignments $\Gamma = \gamma_1, \ldots, \gamma_m$ and choose the best one, $\hat{\gamma} = argmin_{\gamma \in \Gamma} F_s(\gamma)$.

Problem: Exponentially many possible partitions. Approximate the solution by iteratively improving on an initial (possibly random) partition until some stopping criterion is met.

Pros

- Conceptually simple, and easy to implement.
- Efficient. Typically linear in the number of objects.

Cons

- The dependence on the random seeds makes the clustering non-deterministic.
- The number of clusters k must be pre-specified. Often no principled means of a priori specifying k.
- The clustering quality often considered inferior to that of the less efficient hierarchical methods.
- Not as informative as the more stuctured clusterings produced by hierarchical methods.

5.3 k-Means (flat clustering)

Unsupervised variant of the Rocchio classifier

Goal: Partition the n observed objects into k clusters C so that each point $\vec{x_j}$ belongs to the cluster c_i with the nearest centroid $\vec{\mu_i}$.

Typically assumes Euclidean distance as the similarity function s.

The optimization problem: For each cluster, minimize the within-cluster sum of squares, $F_s = WCSS$:

$$WCSS = \sum_{c_i \in C} \sum_{\vec{x_j} \in c_i} ||\vec{x_j} - \vec{\mu_i}||^2$$

Equivalent to minimizing the average squared distance between objects and their cluster centroids (since n is fixed); a measure of how well each centroid represents the members assigned to the cluster.

5.3.1 Algorithm

- Initialize: Compute centroids for k seeds.
- Iterate:
  - Assign each object to the cluster with the nearest centroid.
  - Compute new centroids for the clusters.
- Terminate: When stopping criterion is satisfied.

5.3.2 Properties

In short, we iteratively reassign memberships and recompute centroids until the configuration stabilizes. WCSS is monotonically decreasing (or unchanged) for each iteration. Guaranteed to converge but not to find the global minimum. The time complexity is linear, O(kn).

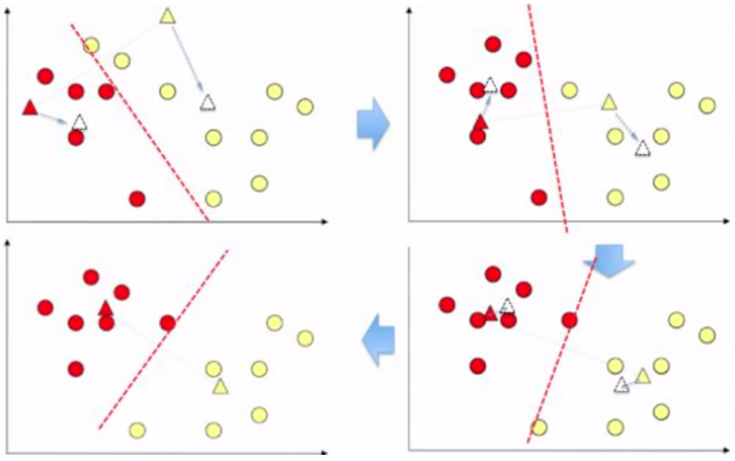


Figure 6:

5.3.3 "Seeding"

We initialize the algorithm by choosing random seeds that we use to compute the first set of centroids. Many possible heuristics for selecting the seeds:

- pick k random objects from the collection
- pick k random points in the space
- pick k sets of m random points and compute centroids for each set
- compute an hierarchical clustering on a subset of the data to find k initial clusters
- etc . . .

The initial seeds can have a large impact on the resulting clustering (because we typically end up only finding a local minimum of the objective function). Outliers are troublemakers.

5.3.4 Possible termination criterions

- Fixed number of iterations
- Clusters or centroids are unchanged between iterations.
- Threshold on the decrease of the objective function (absolute or relative to previous iteration)

5.3.5 Some Close Relatives of k-Means

- k-Medoids: Like k-means but uses medoids instead of centroids to represent the cluster centers.
- Fuzzy c-Means (FCM): Like k-means but assigns soft memberships in [0, 1], where membership is a function of the centroid distance. The computations of both WCSS and centroids are weighted by the membership function.

5.4 Definitions of inter-cluster similarity

- How do we define the similarity between clusters?
- In agglomerative clustering, a measure of cluster similarity $sim(c_i, c_j)$ is usually referred to as a *linkage criterion*:
  - Single-linkage
  - Complete-linkage
  - Centroid-linkage
  - Average-linkage
- Determines which pair of clusters to merge in each step.

5.4.1 Single-linkage

Merge the two clusters with the minimum distance between any two members (Nearest-Neighbors). Can be computed efficiently by taking advantage of the fact that it's best-merge persistent:

- Let the nearest neighbor of cluster c_k be in either c_i or c_j. If we merge $c_i \cup c_j = c_l$, the nearest neighbor of c_k will be in c_l.
- The distance of the two closest members is a local property that is not affected by merging.

I Undesirable chaining effect: Tendency to produce 'stretched' and 'straggly' clusters.

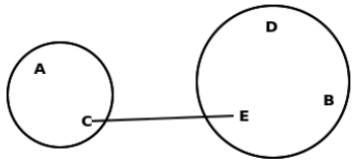


Figure 7: Single-linkage

5.4.2 Complete-linkage

Merge the two clusters where the maximum distance between any two members is smallest (Farthest-Neighbors) Amounts to merging the two clusters whose merger has the smallest diameter. Preference for compact clusters with small diameters. Sensitive to outliers. Not best-merge persistent: Distance defined as the diameter of a merge is a non-local property that can change during merging.

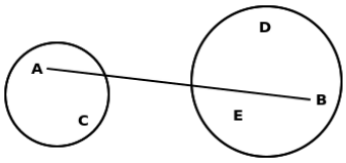


Figure 8: Complete-linkage

5.4.3 Centroid-linkage

Similarity of clusters c_i and c_j defined as the similarity of their cluster centroids $\vec{\mu_i}$ and $\vec{\mu_j}$. Equivalent to the average pairwise similarity between objects from different clusters:

$$sim(c_i, c_j) = \vec{\mu_i} \cdot \vec{\mu_j} = \frac{1}{|c_i||c_j|} \sum_{\vec{x} \in c_i} \sum_{\vec{y} \in c_j} \vec{x} \cdot \vec{y}$$

Not best-merge persistent. Not monotonic, subject to inversions: The combination similarity can increase during the clustering.

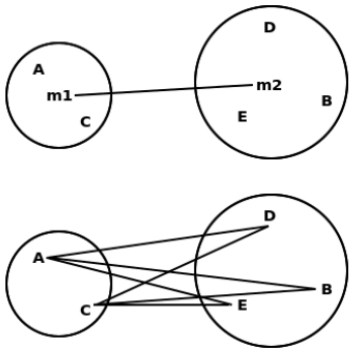


Figure 9: Centroid-linkage

5.4.4 Average-linkage

aka. group-average agglomerative clustering

Merge the clusters with the highest average pairwise similarities in their union. Aims to maximize coherency by considering all pairwise similarities between objects within the cluster to merge (excluding self-similarities). Compromise of complete- and single-linkage. Monotonic but not best-merge persistent. Commonly considered the best default clustering criterion. Can be computed very efficiently if we assume (i) the dot-product as the similarity measure for (ii) normalized feature vectors. Let $c_i \cup c_j = c_k$, and $sim(c_i, c_j) = W(c_i \cup c_j) = W(c_k)$, then:

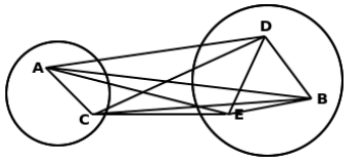


Figure 10: Average-linkage

$$W(c_k) = \frac{1}{|c_k|(|c_k|-1)} \sum_{\vec{x} \in c_k} \sum_{\vec{y} \neq \vec{x} \in c_k} \vec{x} \cdot \vec{y} = \frac{1}{|c_k|(|c_k|-1)} \left(\left(\sum_{\vec{x} \in c_k} \vec{x} \right)^2 - |c_k| \right)$$

The sum of vector similarities is equal to the similarity of their sums.

5.5 Monotinicity

A fundamental assumption in clustering: small clusters are more coherent than large. We usually assume that a clustering is monotonic; Similarity is decreasing from iteration to iteration. This assumpion holds true for all our clustering criterions except for centroid-linkage.

Part II
General programming

6 Dynamic programming

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting the properties of overlapping subproblems and optimal substructure (if an optimal solution can be constructed efficiently from optimal solutions of its subproblems). When applicable, the method takes far less time than naive methods that don't take advantage of the subproblem overlap (like depth-first search).

6.1 Benefits

- Subproblems only solved once, thus reducing the number of computations
- Often memoized (next time the same solution is needed, it is simply looked up)

6.2 Algorithms

- Viterbi
- Earley
- Cocke–Younger–Kasami (CYK/CKY)
- Dijkstra's shortest path
- Minimum edit distance
- Longest common subsequence

7 Diversion: Complexity and O(N)

Big-O notation describes the complexity of an algorithm. It describes the worst-case order of growth in terms of the size of the input. Only the largest order term is represented. Constant factors are ignored. Determined by looking at loops in the code.

Part III
Structured categorization (probabilistic models)

8 Probability

8.1 The chain rule

Since joint probability is symmetric:

$$P(A \cap B) = P(A|B)P(B) = P(B|A)P(A)$$

More generally, using the chain rule:

$$P(A_1 \cap \ldots \cap A_n) = P(A_1)P(A_2|A_1)P(A_3|A_1 \cap A_2)\ldots P(A_n|\cap_{i=1}^{n-1} A_i)$$

The chain rule allows us to break a complicated situation into parts and we can choose the breakdown that suits our problem.

8.2 Bayes' theorem

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Reverses the order of dependence. In conjunction with the chain rule, allows us to determine the probabilities we want from the probabilities we have

9 Language models

A probabilistic (also known as stochastic) language model M assigns probabilities $P_M(x)$ to all strings x in language L.

- L is the sample space
- $0 \leq P_M(x) \leq 1$
- $\sum_{x \in L} P_M(x) = 1$

Language models are used in machine translation, speech recognition systems, spell checkers, input prediction, etc. We can calculate the probability of a string using the chain rule:

$$P(w_1 \ldots w_n) = P(w_1)P(w_2|w_1)P(w_3|w_1 \cap w_2)\ldots P(w_n|\cap_{i=1}^{n-1} w_i)$$

9.1 N-grams

We simplify using the Markov assumption: the last n-1 elements can approximate the effect of the full sequence.

That is, instead of
- P(beach| I want to go to the)
selecting an n of 3, we use
- P(beach| to the)

We call these short sequences of words n-grams:

- bigrams: I want, want to, to go, go to, to the, the beach
- trigrams: I want to, want to go, to go to, go to the
- 4-grams: I want to go, want to go to, to go to the

9.1.1 N-gram models

A generative model models a joint probability in terms of conditional probabilities. We talk about the generative story:

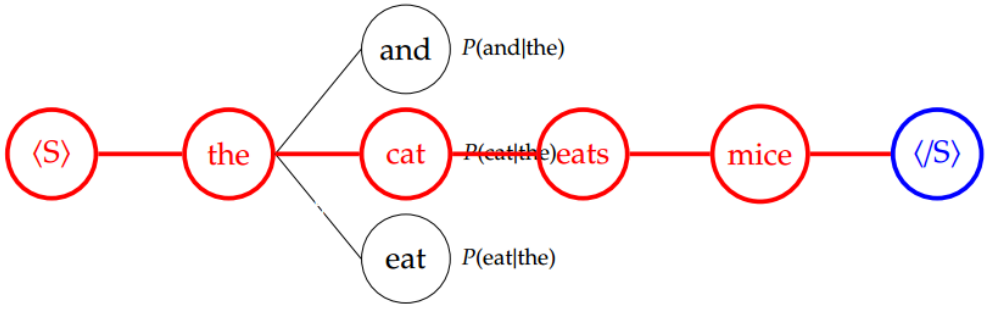


Figure 11:

An n-gram language model records the n-gram conditional probabilities:

P(I|<S>) = 0.0429	P(to|go) = 0.1540
P(want|I) = 0.0111	P(the|to) = 0.1219
P(to|want) = 0.4810	P(beach|the) = 0.0006
P(go|to) = 0.0131	

We calculate the probability of a sentence according to:

$$
\begin{aligned}
P(w_1^n) &\approx \textstyle\prod_{k=i}^{n} P(w_k|w_{k-1}) \\
&\approx P(I|<S>) \times P(want|I) \times P(to|want) \times P(go|to) \times P(to|go) \times P(the|to) \times P(beach|the) \\
&\approx 0.0429 \times 0.0111 \times 0.4810 \times 0.0131 \times 0.1540 \times 0.1219 \times 0.0006 \\
&= 3.38 \times 10^{-11}
\end{aligned}
$$

9.1.2 Maximum Likelihood Estimation (MLE)

Used for training an N-Gram Models

Estimate the probabilities of n-grams by counting (e.g. for trigrams):
$P(bananas|i\ like) = \frac{C(i\ like\ bananas)}{C(i\ like)}$

The probabilities are estimated using the relative frequencies of observed outcomes. This process is called Maximum Likelihood Estimation.

9.1.2.1 Bigram MLE Example

"I want to go to the beach"

w_1	w_2	C($w_1 w_2$)	C(w_1)	P($w_2|w_1$)
<S>	I	1039	24243	0.0429
I	want	46	4131	0.0111
want	to	101	210	0.4810
to	go	128	9778	0.0131
go	to	59	383	0.1540
to	the	1192	9778	0.1219
the	beach	14	22244	0.0006

9.1.3 Problems with MLE of N-Grams

Data sparseness: many perfectly acceptable n-grams will not be observed. Zero counts will result in a estimated probability of 0. Remedy - reassign some of the probability mass of frequent events to less frequent (or unseen) events. Known as *smoothing* or *discounting*. The simplest approach is Laplace ('add-one') smoothing:

$$P_L(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+V}$$

9.1.3.1 Example

"Others want to go to the beach"

w_1	w_2	C(w_1w_2)	C(w_1)	P($w_2|w_1$)	$P_L(w_2|w_1)$
<S>	I	1039	24243	0.0429	0.01934
<S>	Others	17	24243	0.0007	0.00033
I	want	46	4131	0.0111	0.00140
Others	want	0	4131	0	0.00003
want	to	101	210	0.4810	0.00343
to	go	128	9778	0.0131	0.00328
go	to	59	383	0.1540	0.00201
to	the	1192	9778	0.1219	0.03035
the	beach	14	22244	0.0006	0.00029

$$P_L(w_n|w_{n-1}) = \frac{C(w_{n-1}w_n)+1}{C(w_{n-1})+29534}$$

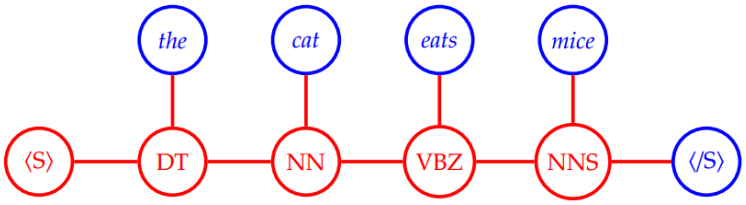
10 POS (part-of-speech) tagging

We are interested in the probability of sequences like:

flies	like	the	wind	**or**	flies	like	the	wind
nns	vb	dt	nn		vbz	p	dt	nn

In normal text, we see the words, but not the tags. Consider the POS tags to be underlying skeleton of the sentence, unseen but influencing the sentence shape. A structure like this, consisting of a hidden state sequence, and a related observation sequence can be modelled as a *Hidden Markov Model*.

10.1 Hidden Markov Models (HMM)



$$P(S,O) = P(\text{DT}|\langle S\rangle)\,P(\text{the}|\text{DT})\,P(\text{NN}|\text{DT})\,P(\text{cat}|\text{NN})\,P(\text{VBZ}|\text{NN})\,P(\text{eats}|\text{VBZ})\,P(\text{NNS}|\text{VBZ})\,P(\text{mice}|\text{NNS})\,P(\langle/S\rangle|\text{NNS})$$

Figure 12: The generative story

For a bi-gram HMM, with O_1^N:

$$P(S,O) = \prod_{i=1}^{N+1} P(s_i|s_{i-1})P(o_i|s_i)$$

where s_0 = <S>, s_{N+1} = </S>

The *transition probabilities* model the probabilities of moving from state to state. The *emission probabilities* model the probability that a state emits a particular observation.

10.1.1 Estimation

As so often in NLP, we learn an HMM from labelled data:

- Transition probabilities:
  Based on a training corpus of previously tagged text, with tags as our state, the MLE can be computed from the counts of observed tags:
  $$P(t_i|t_{t-1}) = \frac{C(t_{i-1}, t_i)}{C(t_{i-1})}$$
- Emission probabilities:
  Computed from relative frequencies in the same way, with the words as observations:
  $$P(w_i|t_j) = \frac{C(t_i, w_j)}{C(t_i)}$$

10.1.2 Implementation Issues

P(S,O) = P(s_1|<S>)P($o_1|s_1$)P($s_2|s_1$)P($o_2|s_2$)P($s_3|s_2$)P($o_3|s_3$) ...
= 0.0429 $\times$ 0.0031 $\times$ 0.0044 $\times$ 0.0001 $\times$ 0.0072 $\times$ ...

- Multiplying many small probabilities $\rightarrow$ underflow
- Solution: work in log(arithmic) space:
  - log(AB) = log(A) + log(B)
  - hence P(A)P(B) = exp(log(A) + log(B))
  - log(P(S, O)) = -1.368 + -2.509 + -2.357 + -4 + -2.143 + ...

The issues related to MLE/smoothing that we discussed for n-gram models also applies here.

10.1.3 Example

Jason likes to eat ice cream. He records his daily ice cream consumption in his diary. The number of ice creams he ate was influenced, but not entirely determined by the weather. Today's weather is partially predictable from yesterday's. A Hidden Markov Model with:
- Hidden states: {H,C} (plus pseudo-states <S> and </S>)
- Observations: {1, 2, 3}

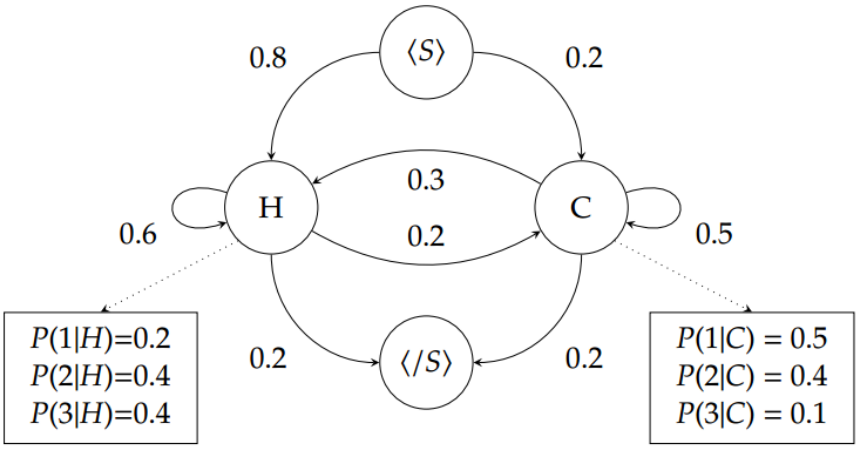


Figure 13:

10.1.4 Find tag sequence

We want to find the tag sequence, given a word sequence. With tags as our states and words as our observations, we know:

$$P(S,O) = \prod_{i=1}^{N+1} P(s_i|s_{i-1})P(o_i|s_i)$$

We want: $P(S|O) = \frac{P(S,O)}{P(O)}$. Actually, we want the state sequence that maximises P(S|O):

$$S_{best} = argmax_S \frac{P(S,O)}{P(O)}$$

Since P(O) always is the same, we can drop the denominator.

10.1.4.1 Example

What is the most likely state sequence S, given an observation sequence O and an HMM.

HMM		if O = {3 1 3}					P
P(H|<S>) = 0.8	P(C|<S>) = 0.2	<S>	H	H	H	</S>	0.0018432
P(H|H) = 0.6	P(C|H) = 0.2	<S>	H	H	C	</S>	0.0001536
P(H|C) = 0.3	P(C|C) = 0.5	<S>	H	C	H	</S>	0.0007680
P(</S>|H) = 0.2	P(</S>|C) = 0.2	<S>	H	C	C	</S>	0.0003200
P(1|H) = 0.2	P(1|C) = 0.5	<S>	C	H	H	</S>	0.0000576
P(2|H) = 0.4	P(2|C) = 0.4	<S>	C	H	C	</S>	0.0000048
P(3|H) = 0.4	P(3|C) = 0.1	<S>	C	C	H	</S>	0.0001200
		<S>	C	C	C	</S>	0.0000500

10.2 Tagger Evaluation

To evaluate a part-of-speech tagger (or any classification system) we:

- train on a labelled training set
- test on a separate test set

For a POS tagger, the standard evaluation metric is tag accuracy:

$$\text{Acc} = \frac{\text{number of correct tags}}{\text{number of words}}$$

The other metric sometimes used is error rate:

$$\text{error rate} = 1 - \text{Acc}$$

11 Viterbi

Most likely state sequence

Recall our problem:

$$\text{maximise } P(s_1 \dots s_n | o_1 \dots o_n) = P(s_1|s_0)P(o_1|s_1)P(s_2|s_1)P(o_2|s_2) \dots$$

Our recursive sub-problem:

$$v_i(x) = max_{k=1}^{L}[v_{i-1}(k) \cdot P(x|k) \cdot P(o_i|x)]$$

The variable $v_i(x)$ represents the maximum probability that the i-th state is x, given that we have seen O_1^i. At each step, we record backpointers showing which previous state led to the maximum probability.

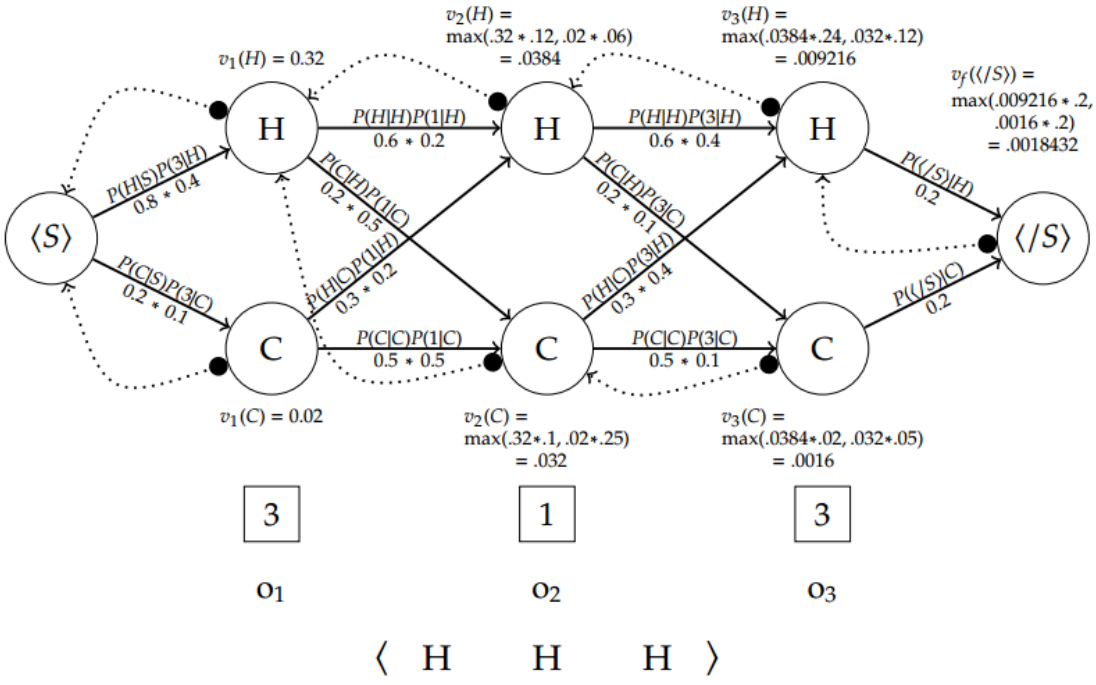


Figure 14: Example

11.1 Pseudocode

Input: observations of length N, state set of size L
Output: best-path
create a path probability matrix viterbi[N, L+2]
create a path backpointer matrix backpointer[N, L+2]
for each state s from 1 to L do
    viterbi[1,s] $\leftarrow$ trans(<S>,s) $\times$ emit(o_1,s)
    backpointer[1,s] $\leftarrow$ 0
end
for each time step i from 2 to N do
    for each state s from 1 to L do
        viterbi[i,s] $\leftarrow max_{s'=1}^{L} viterbi[i-1, s'] \times trans(s', s) \times emit(o_i, s)$
        backpointer[i,s] $\leftarrow argmax_{s'=1}^{L} viterbi[i-1, s'] \times trans(s', s)$
    end
end
viterbi[N, L+1] $\leftarrow max_{s=1}^{L} viterbi[s, N] \times trans(s, </S>)$
backpointer[N, L+1] $\leftarrow argmax_{s'=1}^{L} viterbi[N, s] \times trans(s, </S>)$
return the path by following backpointers from backpointer[N, L+1]

Complexity: $O(L^2 N)$

12 Forward algorithm

Probability for the observation

Again, we use dynamic programming - storing and reusing the results of partial computations in a trellis α. Each cell in the trellis stores the probability of being in state s_x after seeing the first i observations:

$$\alpha_i(x) = P(o_1 \dots o_i, s_i = x) = \sum_{k=1}^{L} \alpha_{i-1}(k) \cdot P(x|k) \cdot P(o_i|x)$$

Note $\sum$, instead of the max in Viterbi.

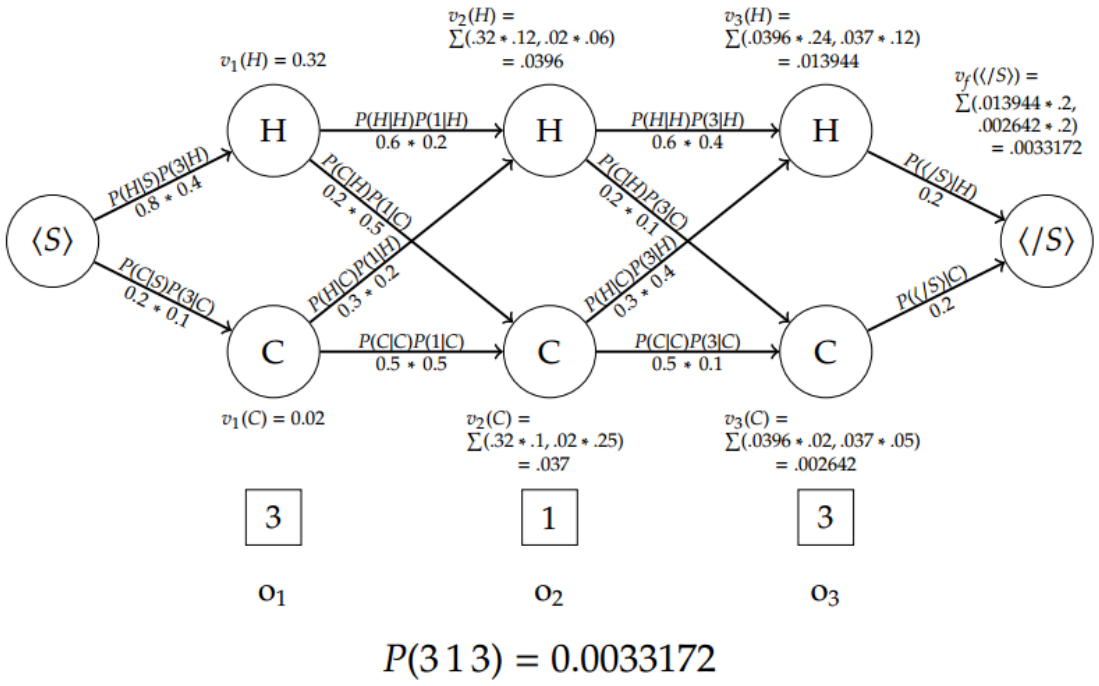


Figure 15: Example

12.1 Pseudocode

Input: observations of length N, state set of size L
Output: forward-probability
create a path probability matrix forward[N, L+2]
for each state s from 1 to L do
    forward[1,s] $\leftarrow$ trans(<S>,s) $\times$ emit(o_1,s)
end
for each time step i from 2 to N do
    for each state s from 1 to L do
        forward[i,s] $\leftarrow \sum_{s'=1}^{L} forward[i-1, s'] \times trans(s', s) \times emit(o_i, s)$
    end
end
forward[N, L+1] $\leftarrow \sum_{s=1}^{L} forward[N, s] \times trans(s, < /S >)$
return forward[N, L+1]

Hierchical Structures

13 Syntactic structure

13.1 Formal grammar

Formal grammar adds hierarchical structure. In NLP, being a sub-discipline of AI, we want our programs to 'understand' natural language. Finding the grammatical structure of sentences is an important step towards 'understanding'. Shift focus from sequences to syntactic structures.

13.2 Why we need structure

13.2.1 Constituency

Words tends to lump together into groups that behave like single units: we call them constituents. Constituency tests give evidence for constituent structure:

- interchangeable in similar syntactic environments.
- can be co-ordinated
- can be moved within a sentence as a unit

Example:

1. Kim read [a very interesting book about grammar]$_{NP}$. Kim read [it]$_{NP}$.
2. Kim [read a book]$_{VP}$, [gave it to Sandy]$_{VP}$, and [left]$_{VP}$.
3. You said I should read the book and [read it]$_{VP}$ I did

Constituents are theory-dependent, and are not absolute or language-independent. Language word order is often described in terms of constituents, and word order may be more or less free within constituents or between them. A constituent usually has a head element, and is often named according to the type of its head:

- A noun phrase (NP) has a nominal (noun-type) head: [a very interesting book about grammar]$_{NP}$
- A verb phrase (VP) has a verbal head: [gives books to students]$_{VP}$

13.2.2 Grammatical functions

Terms such as subject and object describe the grammatical function of a constituent in a sentence. Agreement is generally feature of the relationship between grammatical features.
- The decision of the Nobel committee members surprises most of us.

Why would a purely linear model have problems predicting this phenomenon? Verb agreement reflects the grammatical structure of the sentence, not just the sequential order of words.

13.3 Syntactic Ambiguity

Figure 16:

14 Context Free Grammars (CFGs)

Formal system for modeling constituent structure. Defined in terms of a lexicon and a set of rules. Formal models of 'language' in a broad sense; natural languages, programming languages, communication protocols, ... Can be expressed in the 'meta-syntax' of the *Backus-Naur Form (BNF)* formalism.

```
<symbol> ::= __expression__

Example, postal service:

<postal-address> ::= <name-part> <street-address> <zip-part>

<name-part> ::= <personal-part> <last-name> <opt-suffix-part> <EOL>
| <personal-part> <name-part>

<personal-part> ::= <first-name> | <initial> "."

<street-address> ::= <house-num> <street-name> <opt-apt-num> <EOL>

<zip-part> ::= <town-name> "," <state-code> <ZIP-code> <EOL>

<opt-suffix-part> ::= "Sr." | "Jr." | <roman-numeral> | ""

<opt-apt-num> ::= <apt-num> | ""
```

Powerful enough to express sophisticated relations among words, yet in a computationally tractable way.

14.1 Formally

Formally, a CFG is a quadruple: G = <C, Σ, P, S>

- C is the set of categories (aka non-terminals), {S, NP, VP, V}
- Σ is the vocabulary (aka terminals), {Kim, snow, adores, in}

- P is a set of category rewrite rules (aka productions):
  S $\rightarrow$ NP VP NP $\rightarrow$ Kim
  VP $\rightarrow$ V NP NP $\rightarrow$ snow
  V $\rightarrow$ adores
- S $\in$ C is the start symbol, a filter on complete results;
  for each rule $\alpha \rightarrow \beta_1, \beta_2, \ldots, \beta_n \in P : \alpha \in C$ and $\beta_i \in C \cup \Sigma$

14.2 Generative Grammar

Top-down view of generative grammars:

- For a grammar G, the language L_G is defined as the set of strings that can be derived from S.
- To derive w_1^n from S, we use the rules in P to recursively rewrite S into the sequence w_1^n where each $w_i \in \Sigma$.
- The grammar is seen as generating strings.
- Grammatical strings are defined as strings that can be generated by the grammar.
- The 'context-freeness' of CFGs refers to the fact that we rewrite non-terminals without regard to the overall context in which they occur.

15 Treebanks

Generally a treebank is a corpus paired with 'gold-standard' (syntactic) analyses. Can be created by manual annotation or selection among outputs from automated processing (plus correction).
Lexical rules: P(tag | word)
Syntactic rules: P(tag | previous tag)

15.1 Penn Treebank

(Marcus et al., 1993)

About one million tokens of Wall Street Journal text. Hand-corrected PoS annotation using 45 word classes. Manual annotation with (somewhat) coarse constituent structure.

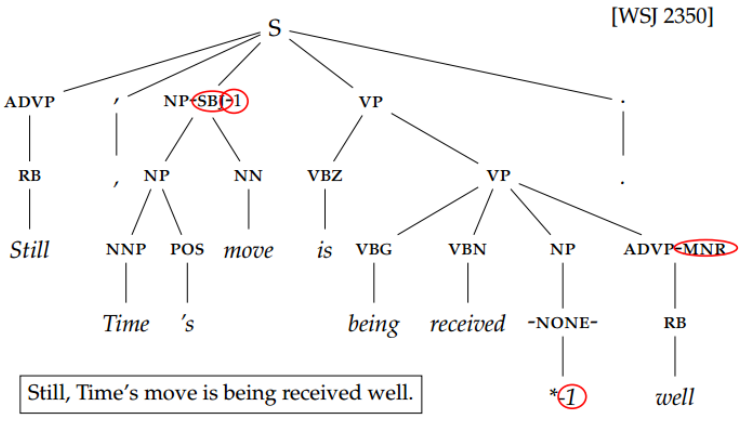


Figure 17: Example from the Penn Treebank

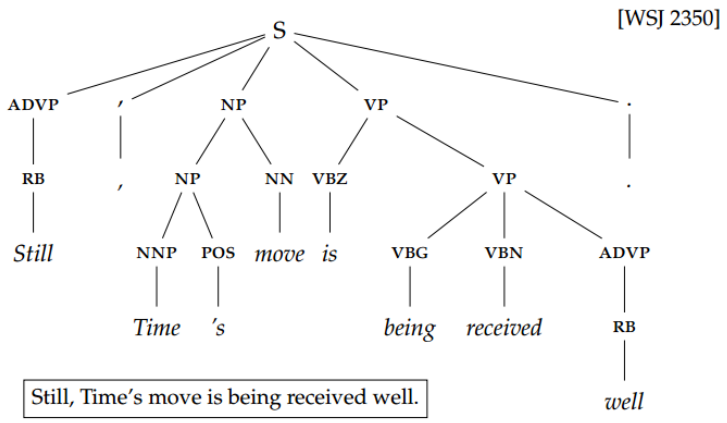


Figure 18: Elimination of Traces and Functions

16 Probabilitic Context-Free Grammars (PCFG)

We are interested, not just in which trees apply to a sentence, but also to which tree is most likely. Probabilistic context-free grammars (PCFGs) augment CFGs by adding probabilities to each production, e.g.

$S \rightarrow NP\ VP$ 0.6

$S \rightarrow NP\ VP\ PP$ 0.4

These are conditional probabilities - the probability of the right hand side (RHS) given the left hand side (LHS).
$P(S \rightarrow NP\ VP) = P(NP\ VP|S)$

We can learn these probabilities from a treebank, again using Maximum Likelihood Estimation.

16.1 Estimating PCFGs

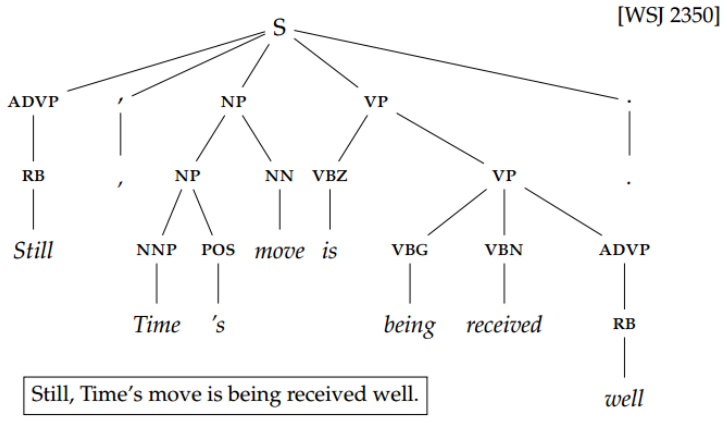


Figure 19:

```
(S
   (ADVP (RB "Still"))
   (|,| ",")
   (NP
      (NP (NNP "Time") (POS "'s"))
      (NN "move"))
    (VP
      (VBZ "is")
      (VP
         (VBG "being")
         (VP
           (VBN "received")
           (ADVP (RB "well")))))
   (\. "."))
```

RB → Still	1
AVP → RB	2
|,| → ,	1
NNP → Time	1
POS → 's	1
NP → NNP POS	1
NN → move	1
NP → NP NN	1
VBZ → is	1
VBG → being	1
VBN → received	1
RB → well	1
VP → VBN ADVP	1
VP → VBG VP	1
\. → .	1
S → ADVP |,| NP VP \.	1
START → S	1

Figure 20:

Once we have counts of all the rules, we turn them into probabilities.

S → ADVP |,| NP VP \.	50	S → NP VP \.	400
S → NP VP PP \.	350	S → VP !	100
S → NP VP S \.	200	S → NP VP	50

$$P(S \rightarrow \text{ADVP |,| NP VP \.}) \approx \frac{C(S \rightarrow \text{ADVP |,| NP VP \.})}{C(S)} = \frac{50}{1150} = 0.0435$$

17 Basic parsing strategies

17.1 Parsing with CFGs

S → NP VP
VP → V | V NP | VP PP
NP → NP PP
PP → P NP
NP → Kim | snow | Oslo
V → adores
P → in

All Complete Derivations

- are rooted in the start symbol S;
- label internal nodes with categories $\in C$, leafs with words $\in \Sigma$;
- instantiate a grammar rule $\in P$ at each local subtree of depth one.

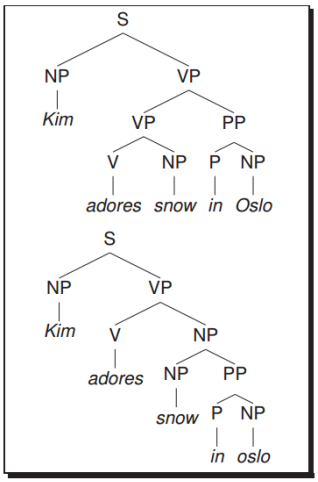


Figure 21: *Moving to a Procedural View*

17.2 Recursive Descend

A Naive Parsing Algorithm

17.2.1 Control Structure

- top-down: given a parsing goal α, use all grammar rules that rewrite α;
- successively instantiate (extend) the right-hand sides of each rule;
- for each β_i in the RHS of each rule, recursively attempt to parse β_i;
- termination: when α is a prefix of the input string, parsing succeeds.

17.2.2 (Intermediate) Results

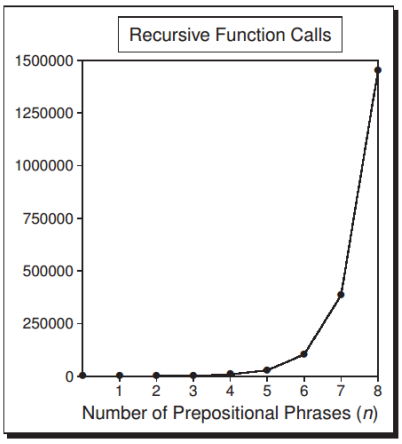
- Each result records a (partial) tree and remaining input to be parsed;
- complete results consume the full input string and are rooted in S;
- whenever a RHS is fully instantiated, a new tree is built and returned;
- all results at each level are combined and successively accumulated.

17.2.3 Pseudocode

```
(defun parse (input goal)
(if (equal (first input) goal)
(let ((edge (make-edge :category (first input))))
(list (make-parse :edge edge :input (rest input))))
(loop
for rule in (rules-deriving goal)
append (extend-parse (rule-lhs rule) nil (rule-rhs rule) input))))

(defun extend-parse (goal analyzed unanalyzed input)
(if (null unanalyzed)
(let ((tree (cons goal analyzed)))
(list (make-parse :tree tree :input input)))
(loop
for parse in (parse input (first unanalyzed))
append (extend-parse
goal (append analyzed (list (parse-tree parse)))
(rest unanalyzed)
(parse-input parse)))))
```

17.3 Quantifying the Complexity of the Parsing Task



Kim adores snow (in Oslo)^n

n	trees	calls
0	1	46
1	2	170
2	5	593
3	14	2,093
4	42	7,539
5	132	27,627
6	429	102,570
7	1430	384,566
8	4862	1,452,776
⋮	⋮	⋮

Figure 22:

17.4 Top-Down (Goal-Oriented)

- left recursion (e.g. a rule like 'VP $\rightarrow$ VP PP') causes infinite recursion;
- search is uninformed by the (observable) input: can hypothesize many unmotivated sub-trees, assuming terminals (words) that are not present.

17.5 Bottom-up (Data-Oriented)

- unary (left-recursive) rules (e.g. 'NP $\rightarrow$ NP') would still be problematic;
- lack of parsing goal: compute all possible derivations for, say, the input 'adores snow'; however, it is ultimately rejected since it is not sentential;
- availability of partial analyses desirable for, at least, some applications.

17.6 Local Ambiguity

- For many substrings, more than one way of deriving the same category;
- NPs: 1 | 2 | 3 | 6 | 7 | 9; PPs: 4 | 5 | 8; 9 ≡ 1 + 8 | 6 + 5;
- *parse forest* — a single item represents multiple trees [Billot & Lang, 89].

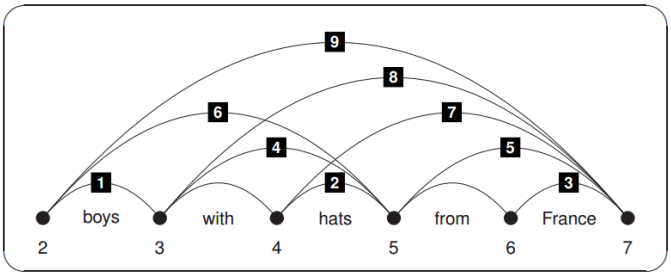


Figure 23:

17.7 CYK/CKY (Cocke, Kasami, & Younger)

17.7.1 Pseudocode

```
for (0 ≤ i < |input|) do
        chart[i,i+1] ≤ {α | α → input_i ∈ P };
end;
for (1 ≤ l < |input|) do
        for (0 ≤ i < |input| - l) do
                for (1 ≤ j ≤ l) do
                        if (α → β1β2 ∈ P ∧β1 ∈ chart[i,i+j] ∧ β2 ∈ chart[i+j,i+l+1]) then
                                chart[i,i+l+1] ← chart[i,i+l+1] ∪ {α};
                        end;
                end;
        end;
end;
```

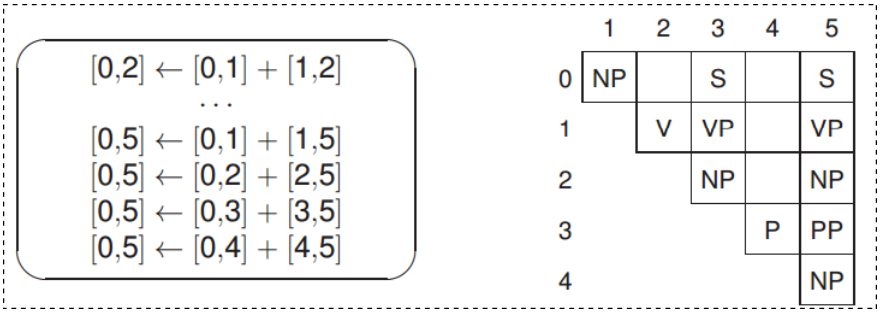


Figure 24:

17.7.2 Limitations

17.7.2.1 Built-In Assumptions

Chomsky Normal Form grammars:

$$\alpha \rightarrow \beta_1\beta_2 \text{ or } \alpha \rightarrow \gamma(\beta_i \in C, \gamma \in \Sigma)$$

Breadth-first (aka. exhaustive): always compute all values for each cell.
Rigid control structure: bottom-up, left-to-right (one diagonal at a time).

17.7.2.2 Generalized Chart Parsing

Liberate order of computation: no assumptions about earlier results. Active edges encode partial rule instantiations, 'waiting' for additional (adjacent and passive) constituents to complete: $[1, 2, VP \rightarrow V \cdot NP]$. Parser can fill in chart cells in any order and guarantee completeness.

17.8 Chart Parsing - Specialized Dynamic Programming

17.8.1 Basic Notions

- Use chart to record partial analyses, indexing them by string positions;
- count inter-word vertices; CKY: chart row is start, column end vertex;

- treat multiple ways of deriving the same category for some substring as equivalent; pursue only once when combining with other constituents.

17.8.2 Key Benefits

- Dynamic programming (memoization): avoid recomputation of results;
- efficient indexing of constituents: no search by start or end positions;
- compute parse forest with exponential 'extension' in polynomial time.

17.9 Generalized Chart Parsing

The parse chart is a two-dimensional matrix of edges (aka chart items). An edge is a (possibly partial) rule instantiation over a substring of input. The chart indexes edges by start and end string position (aka vertices). Dot in rule RHS indicates degree of completion:

$$\alpha \rightarrow \beta_1 \dots \beta_{i-1} \bullet \beta_i \dots \beta_n$$

Active edges (aka incomplete items) - partial RHS: [1, 2, VP $\rightarrow$ V • NP]
Passive edges (aka complete items) - full RHS: [1, 3,VP $\rightarrow$ V NP •]

17.9.1 Fundamental Rule

$$[i, j, \alpha \rightarrow \beta_1 \dots \beta_{i-1} \bullet \beta_i \dots \beta_n] + [j, k, \beta_i \rightarrow \gamma^+ \bullet] \mapsto [i, k, \alpha \rightarrow \beta_1 \dots \beta_i \bullet \beta_{i+1} \dots \beta_n]$$

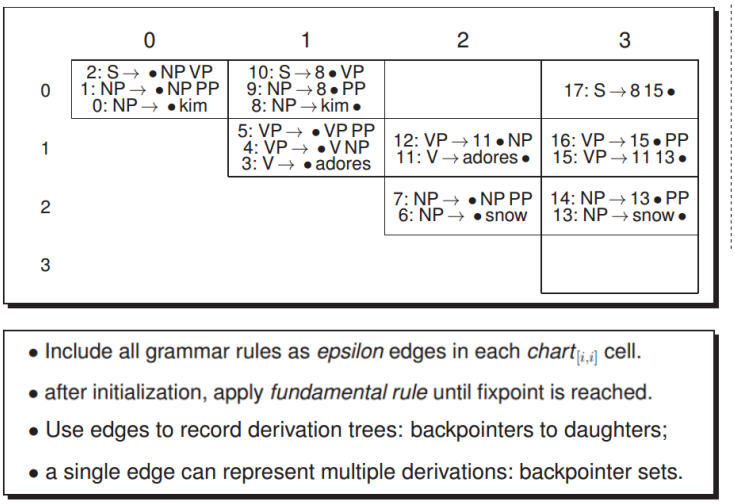


Figure 25:

17.9.2 Combinatorics: Keeping Track of Remaining Work

17.9.2.1 The Abstract Goal

Any chart parsing algorithm needs to check all pairs of adjacent edges.

17.9.2.2 A Naive Strategy

Keep iterating through the complete chart, combining all possible pairs, until no additional edges can be derived (i.e. the fixpoint is reached). Frequent attempts to combine pairs multiple times: deriving 'duplicates'.

17.9.2.3 An Agenda-Driven Strategy

Combine each pair exactly once, viz. when both elements are available. Maintain agenda of new edges, yet to be checked against chart edges. New edges go into agenda first, add to chart upon retrieval from agenda.

17.9.3 Pseudocode

Initialization:

- for each word in input string
  - add passive lexical edge <word• > to chart
  - for each $\alpha \rightarrow$ word $\in$ P
    - add passive $< \alpha \rightarrow$ word• $>$ edge to agenda

Main Loop:

- while edge $\leftarrow$ pop-agenda()
  - if equivalent edge in chart, pack; otherwise add edge
  - if edge is passive
    - for each active edge a to the left, fundamental-rule(a, edge)
    - predict new edges from P, and add to the agenda
  - else
    - for each passive edge p to the right, fundamental-rule(edge, p)

Termination:

- return all edges with category S that span the full input

17.10 Ambiguity Packing

17.10.1 General Idea

Maintain only one edge for each α from i to j (the 'representative'). Record alternate sequences of daughters for α in the representative.

17.10.2 Implementation

Group passive edges into equivalence classes by identity of α, i, and j. Search chart for existing equivalent edge (h) for each new edge (e). When h (the 'host' edge) exists, pack e into h to record equivalence. e not added to the chart, no derivations with or further processing of e. Unpacking multiply out all alternative daughters for all result edges.

17.11 Unpack Parse Forest

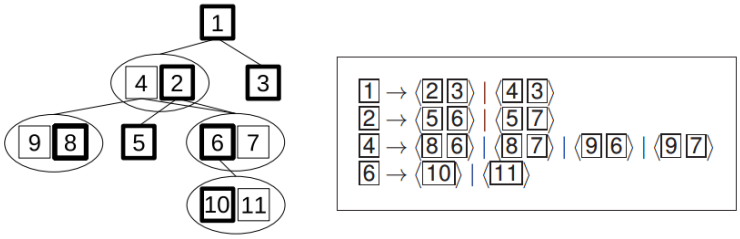


Figure 26: Ten complete trees in total

17.12 Viterbi Decoding over the Parse Forest

Recall the Viterbi algorithm for HMMs

$$v_i(x) = max_{k=1}^{L}[v_{i-1}(k) \cdot P(x|k) \cdot P(o_i|x)]$$

For our trees, we no longer have a linear order, but we still build up cached Viterbi values successively:

$$v(e) = max \left[P(\beta_1, \ldots, \beta_n|\alpha) \times \prod_i v(\beta_i) \right]$$

Similar to HMM decoding, we also need to keep track of the set of daughters that led to the maximum probability. Implementation: cache the highest-scoring edge within e, recording the maximum probability of its sub-tree, and the daughter sequence that led to it.

17.13 Chart Parsing Summary

- Organize edges in an agenda, and process them sequentially.
- A passive edge is complete, an active edge is still looking for daughters.
- Processing records the edge in the chart, and then may add other edges to the agenda, either through the fundamental rule, or through (active) edge prediction, or both.
- The edge data structure records:
  - category (LHS)
  - seen elements of rule RHS as daughter edges
  - unseen (unanalyzed) elements of rule RHS
  - input span covered (as start and end chart vertices)
  - probability of the sub-tree, reflecting rule probabilities
  - highest-scoring (cached) edge after Viterbi processing
- The agenda ordering and prediction strategy influence the search order (which can be varied fully freely).

18 Parser Evaluation

There are a number of aspects to consider in judging parser performance:

- **Coverage**:
  The percentage of inputs for which we we found an analysis.
- **Overgeneration**:
  The percentage of ungrammatical inputs (incorrectly) assigned an analysis.
- **Efficiency**:
  Time and memory used by the parser.
- **Accuracy**:
  Sentence accuracy measures the percentage of input sentences which received the right tree. Since full trees can be quite complex, this is a very strict metric, and so most statistical parsers report accuracy according to the granular ParsEval metric.

18.1 ParsEval

- The ParsEval metric (Black, et al., 1991) measures constituent overlap.
- The original formulation only considered the shape of the (unlabeled) bracketing.
- The modern 'standard' uses a tool called evalb, which reports precision, recall and F_1 score for labeled brackets, as well as the number of crossing brackets.

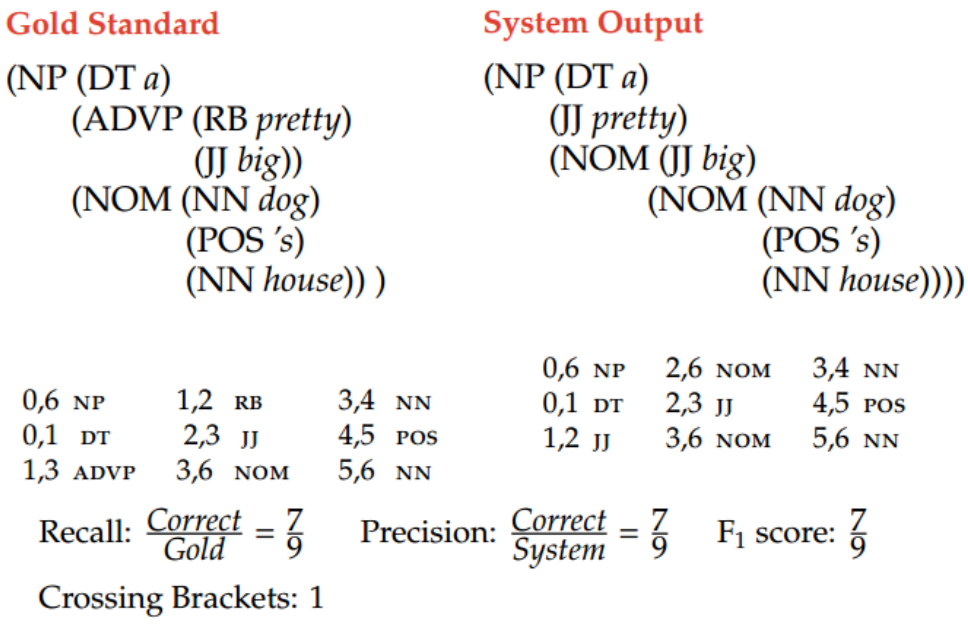


Figure 27: ParsEval

Part IV
Common lisp

19 def*

19.1 Functions

defun : used to define functions

19.1.1 Higher-Order Functions

Functions that accept functions as arguments or return values.

19.1.2 Anonymous Functions

We can also pass function arguments without first binding them to a name, using lambda expressions: (lambda (parameters) body)

19.2 Struct

defstruct : used to define structs

```
(defstruct (name option-1 option-2 ... option-m)
          doc-string
          slot-description-1
          slot-description-2
          ...
          slot-description-n)

(defstruct album
          (artist "unknown")
          (title "unknown"))

(defparameter foo (make-album :artist "Elvis"))
-> #S(album :artist "Elvis" :title "unknown")

(listp foo) -> nil
(album-p foo) -> t

(setf (album-title foo) "Blue Hawaii")
foo -> #S(album :artist "Elvis" :title "Blue Hawaii")
```

19.3 Parameter

defparameter : always assigns a value

```
(defparameter a 1)
(defparameter a 2)

a -> 2
```

defvar : assign value only once

```
(defvar b 1)
(defvar b 2)

b -> 1
```

setf : a macro which uses 'setq' internally, but has more possibilities. In a way it's a more general assignment operator. E.g. with 'setf' you can do:

```
(defparameter c '(1 2 3))
(setf (car c) 42)

c -> (42 2 3)
```

but you can't do that with 'setq':

```
(setq (car c) 42)

-> #ERROR
```

20 Parameter Lists: Variable Arities and Ordering

20.1 Optional Parameters

```
(defun foo (x &optional y (z 42))
  (list x y z))

(foo 1) -> (1 nil 42)
(foo 1 2 3) -> (1 2 3)
```

20.2 Keyword Parameters

```
(defun foo (x &key y (z 42))
  (list x y z))

(foo 1) -> (1 nil 42)
(foo 1 :z 3 :y 2) -> (1 2 3)
```

20.3 Rest Parameters

```
(defun avg (x &rest rest)
  (let ((numbers (cons x rest)))
    (/ (apply #'+ numbers)
       (length numbers))))

(avg 3) -> 3
(avg 1 2 3 4 5 6 7) -> 4
```

21 Equality

- **eq** tests object identity; it is not useful for numbers or characters.

- **eql** is like eq, but well-defined on numbers and characters.
- **equal** tests structural equivalence
- **equalp** is like equal but insensitive to case and numeric type.
- Also many type-specialized tests like **=**, **string=**, etc.

```
(eq (list 1 2 3) '(1 2 3)) -> nil
(equal (list 1 2 3) '(1 2 3)) -> t
(eq 42 42) -> ? [implementation-dependent]
(eql 42 42) -> t
(eql 42 42.0) -> nil
(equalp 42 42.0) -> t
(equal "foo" "foo") -> t
(equalp "FOO" "foo") -> t
```

22 Iteration

22.1 dolist

```
(let ((result nil))
  (dolist (x '(0 1 2 3 4 5))
    (when (evenp x)
      (push x result)))
    (reverse result))

-> (0 2 4)
```

22.2 dotimes

```
(let ((result nil))
  (dotimes (x 6)
    (when (evenp x)
      (push x result)))
    (reverse result))

-> (0 2 4)
```

22.3 loop

```
(loop
  for x below 6
  when (evenp x)
  collect x)

-> (0 2 4)
```

```
(loop
  for i from 10 to 50 by 10
  collect i)

-> (10 20 30 40 50)
```

- Iteration over lists or vectors: for symbol in | on | across list
- Counting through ranges:
  for symbol [from number] to | downto number [by number]
- Iteration over hash tables:
  for symbol being each hash-key | hash-value in hash table
- Stepwise computation:
  for symbol = sexp then sexp
- Accumulation:
  collect | append | sum | minimize | count | . . . sexp
- Control:
  while | until | repeat | when | unless | . . . sexp
- Local variables:
  with symbol = sexp
- Initialization and finalization:
  initially | finally sexp+
- All of these can be combined freely, e.g. iterating through a list, counting a range, and stepwise computation, all in parallel.
- Note: without at least one accumulator, loop will only return nil.

23 Input/Output

Reading and writing is mediated through streams. The symbol *t* indicates the default stream, the terminal.

```
(format t "~a is the ~a.~%" 42 "answer")
42 is the answer. ;; returns nil
```

- (read-line stream nil) reads one line of text from stream, returning it as a string.
- (read stream nil) reads one well-formed s-expression.
- The second reader argument asks to return nil upon end-of-file.

```
(with-open-file (stream "sample.txt" :direction :input)
  (loop
    for line = (read-line stream nil)
    while line do (format t "~a~%" line)))
```

24 Sequence

24.1 Arrays

Integer-indexed container (indices count from zero)

```
(defparameter array (make-array 5)) -> #(nil nil nil nil nil)
(setf (aref array 0) 42) -> 42
array -> #(42 nil nil nil nil)
```

Can be fixed-sized (default) or dynamically adjustable. Can also represent rectangular 'grids' of multiple dimensions:

```
(defparameter array (make-array '(2 5) :initial-element 0)) -> #((0 0 0 0 0) (0 0 0 0 0))

(incf (aref array 1 2)) -> 1
```

- Vectors = specialized type of arrays: one-dimensional.
- Strings = specialized type of vectors (similarly: bit vectors).
- Vectors and lists are subtypes of an abstract data type sequence.
- Large number of built-in sequence functions, e.g.:

```
(length "foo") -> 3
(elt "foo" 0) -> #\f
(count-if #'numberp '(1 a "2" 3 (b))) -> 2
(subseq "foobar" 3 6) -> "bar"
(substitute #\a #\o "hoho") -> "haha"
(remove 'a '(a b b a)) -> (b b)
(some #'listp '(1 a "2" 3 (b))) -> t
(sort '(1 2 1 3 1 0) #'<) -> (0 1 1 1 2 3)
```

Others: position, every, count, remove-if, find, merge, map, reverse, concatenate, reduce, ...

24.2 Plist (Property List)

I A property list is a list of alternating keys and values:

```
(defparameter plist (list :artist "Elvis" :title "Blue Hawaii"))
(getf plist :artist) -> "Elvis"
(getf plist :year) -> nil
(setf (getf plist :year) 1961) -> 1961
(remf plist :title) -> t
plist -> (:artist "Elvis" :year 1961)
```

getf and remf always test using eq (not allowing :test argument). Restricts what we can use as keys (typically symbols/keywords).

24.3 Alist (Association List)

An association list is a list of pairs of keys and values:

```
(defparameter alist (pairlis '(:artist :title)
                             '("Elvis" "Blue Hawaii")))
-> ((:artist . "Elvis") (:title . "Blue Hawaii"))

(assoc :artist alist) -> (:artist . "Elvis")

(setf alist (acons :year 1961 alist))
-> ((:artist . "Elvis") (:title . "Blue Hawaii") (:year . 1961))
```

Note: The result of cons'ing something to an atomic value other than nil is displayed as a dotted pair; (cons 'a 'b) -> (a . b) I With the :test keyword argument we can specify the lookup test function used by assoc; keys can be any data type. With look-up in a plist or alist, in the worst case, every element in the list has to be searched (linear complexity in list length).

24.4 Hash Table

While lists are inefficient for indexing large data sets, and arrays restricted to numeric keys, hash tables efficiently handle a large number of (almost) arbitrary type keys. Any of the four built-in equality tests can be used for key comparison.

```
(defparameter table (make-hash-table :test #'equal))
(gethash "foo" table) -> nil

(setf (gethash "foo" table) 42) -> 42
```

'Trick' to test, insert and update in one go (specifying 0 as the default):

```
(incf (gethash "bar" table 0)) -> 1
(gethash "bar" table) -> 1
```

Hash table iteration: use maphash or specialized loop directives.

25 Commands

- (car/first <list>): returns first element of list
- (cdr/rest <list>): returns all but first element of list
- (nth <N> <list>): return the Nth element of the list
- (cond ((<pred> <body>))): conditional
- (if <pred> <true> <false>): if-statement
- (let ((<var> <value>)) <body>)
- (let* ((<var> <value>)) <body>): may depend on other variables in same let
- (string-trim <list of symbols> <string>)
- (string-downcase <string>)

Index