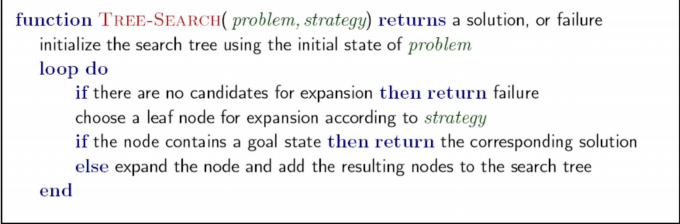
Reflex agents: § Choose action based on current percept (and maybe memory) § May have memory or a model of the world’s current state § Do not consider the future consequences of their actions § Consider how the world IS.

Planning agents: § Ask “what if” § Decisions based on (hypothesized) consequences of actions § Must have a model of how the world evolves in response to actions § Must formulate a goal (test) § Consider how the world WOULD BE

§ A search problem consists of: § A state space § A successor function (with actions, costs) § A start state and a goal test § A solution is a sequence of actions (a plan)

World state: every detail of the environment, State Space: only details needed for planning

DFS time: b^m, space: bm

b=branching factor, m=levels

BFS time: b^s, space b^s

UCS time: b^(C\*/ e) (exponential in effective depth)

space: b^(C\*/e)

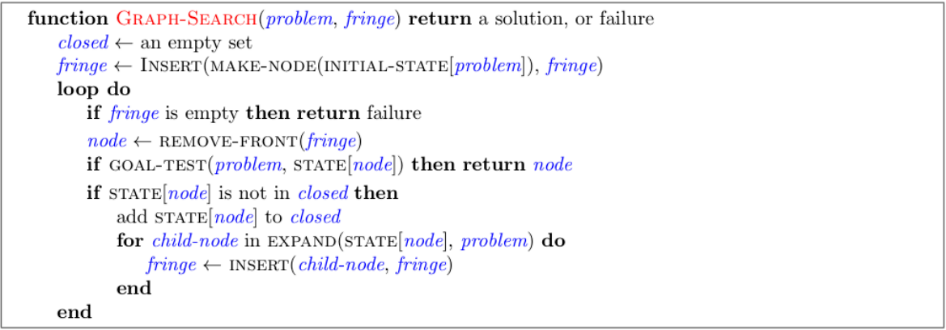
Iterative Deepening § Idea: get DFS’s space advantage with BFS’s time / shallow-solution advantages § Run a DFS with depth limit 1. If no solution… § Run a DFS with depth limit 2. If no solution… § Run a DFS with depth limit 3. …..

§ A heuristic is: § A function that estimates how close a state is to a goal § Designed for a particular search problem § Examples: Manhattan distance, Euclidean distance for pathing

§ A\* Search orders by the sum: f(n) = g(n) + h(n)

Admissible: h(A) <= actual cost A to G. Max of admissible heuristics is admissible.

Consistent: h(A) – h(C) <= cost(A to C). Consistency implies admissibility. f value along path never decrease



Constraint satisfaction problems (CSPs): § A special subset of search problems § State is defined by variables Xi with values from a domain D (sometimes D depends on i) § Goal test is a set of constraints specifying allowable combinations of values for subsets of variables

§ Backtracking search is the basic uninformed algorithm for solving CSPs § Idea 1: One variable at a time § Variable assignments are commutative, so fix ordering § I.e., [WA = red then NT = green] same as [NT = green then WA = red] § Only need to consider assignments to a single variable at each step § Idea 2: Check constraints as you go § I.e. consider only values which do not conflict previous assignments § Might have to do some computation to check the constraints § “Incremental goal test” § Depth-first search with these two improvements is called backtracking search (not the best name)

Filtering: Keep track of domains for unassigned variables and cross off bad options § Forward checking: Cross off values that violate a constraint when added to the existing assignment. § An arc X → Y is consistent iff for every x in the tail there is some y in the head which could be assigned without violating a constraint. § Forward checking: Enforcing consistency of arcs pointing to each new assignment

Arc consistency: enforce consistency of all arcs, delete from the tail

§ Variable Ordering: Minimum remaining values (MRV): § Choose the variable with the fewest legal left values in its domain

§ Value Ordering: Least Constraining Value § Given a choice of variable, choose the least constraining value (filter)

Theorem: if the constraint graph has no loops, the CSP can be solved in O(n\*d2) time

Algorithm for tree-structured CSPs: § Order: Choose a root variable, order variables so that parents precede children § Remove backward: For i = n : 2, apply RemoveInconsistent(Parent(Xi ),Xi ) § Assign forward: For i = 1 : n, assign Xi consistently with Parent(Xi )

§ Cutset conditioning: instantiate (in all ways) a set of variables such that the remaining constraint graph is a tree – could backtrack until cutset assigned

Iterative Improvement § Algorithm: While not solved, § Variable selection: randomly select any conflicted variable § Value selection: min-conflicts heuristic: § Choose a value that violates the fewest constraints § I.e., hill climb with h(n) = total number of violated constraints.

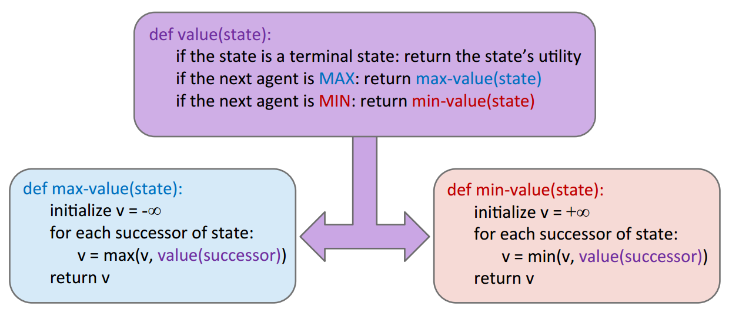
Hill Climbing Start wherever § Repeat: move to the best neighboring state § If no neighbors better than current, quit. Simulated Annealing § Idea: Escape local maxima by allowing downhill moves § But make them rarer as time goes on

Deterministic Games: § Many possible formalizations, one is: § States: S (start at s0) § Players: P={1...N} (usually take turns) § Actions: A (may depend on player / state) § Transition Function: SxA → S § Terminal Test: S → {t,f} § Terminal Utilities: SxP → R § Solution for a player is a policy: S → A

Zero-Sum Games § Agents have opposite utilities (values on outcomes) § Lets us think of a single value that one maximizes and the other minimizes § Adversarial, pure competition

Value of a state: The best achievable outcome (utility) from that state

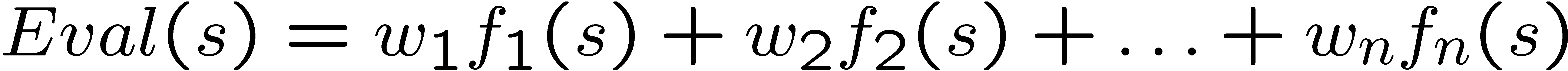
§ Minimax search: § A state-space search tree § Players alternate turns § Compute each node’s minimax value: the best achievable utility against a rational (optimal) adversary. efficiency? just like DFS



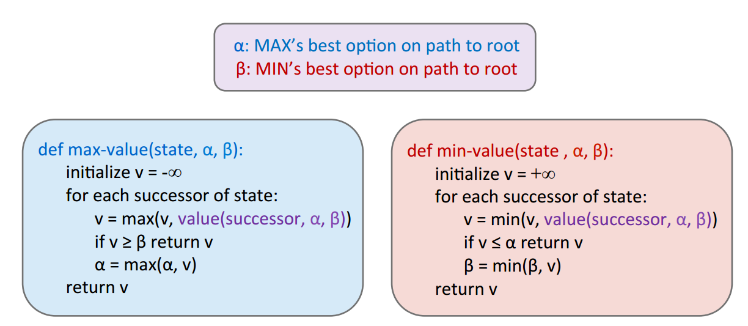
§ Problem: In realistic games, cannot search to leaves! § Solution: Depth-limited search § Instead, search only to a limited depth in the tree § Replace terminal utilities with an evaluation function for non-terminal positions

§ Evaluation functions are always imperfect § The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters § An important example of the tradeoff between complexity of features and complexity of computation

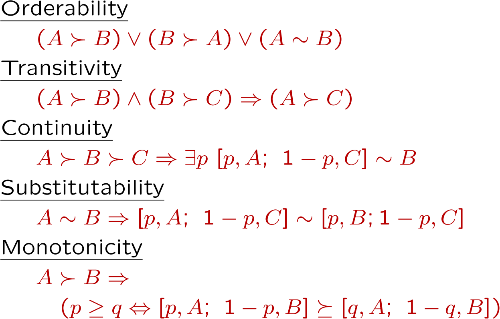
§ Ideal function: returns the actual minimax value of the position § In practice: typically weighted linear sum of features: § e.g. f1(s) = (num white queens – num black queens), etc.



Alpha Beta Pruning § General configuration (MIN version) § We’re computing the MIN-VALUE at some node n § We’re looping over n’s children § n’s estimate of the childrens’ min is dropping § Who cares about n’s value? MAX § Let a be the best value that MAX can get at any choice point along the current path from the root § If n becomes worse than a, MAX will avoid it, so we can stop considering n’s other children (it’s already bad enough that it won’t be played) § MAX version is symmetric



§ Why wouldn’t we know what the result of an action will be? § Explicit randomness: rolling dice § Unpredictable opponents: the ghosts respond randomly § Actions can fail: when moving a robot, wheels might slip § Values should now reflect average-case (expectimax) outcomes, not worst-case (minimax) outcomes § Expectimax search: compute the average score under optimal play § Max nodes as in minimax search § Chance nodes are like min nodes but the outcome is uncertain § Calculate their expected utilities § I.e. take weighted average (expectation) of children (txp_fig= preference)



Behavior is invariant under positive linear transformation

Given a lottery L = [p, $X; (1-p), $Y] § The expected monetary value EMV(L) is p\*X + (1-p)\*Y § U(L) = p\*U($X) + (1-p)\*U($Y)

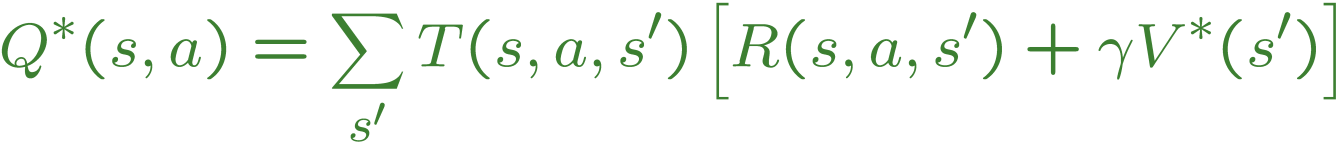
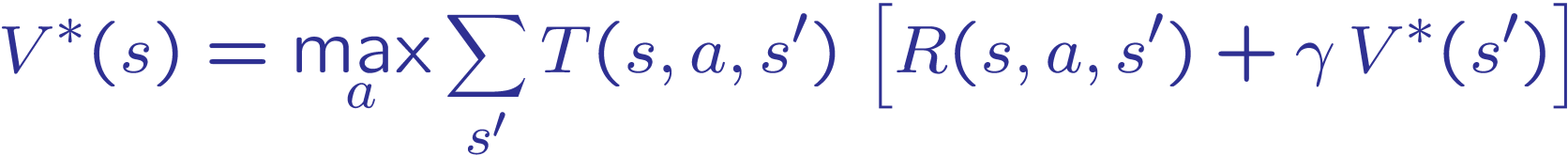
An MDP is defined by: § A set of states s ∈ S § A set of actions a ∈ A § A transition function T(s, a, s’) § Probability that a from s leads to s’, i.e., P(s’| s, a) § Also called the model or the dynamics § A reward function R(s, a, s’) § Sometimes just R(s) or R(s’) § A start state § Maybe a terminal state

§ For MDPs, we want an optimal policy π\*: S → A § A policy π gives an action for each state § An optimal policy is one that maximizes expected utility if followed § An explicit policy defines a reflex agent

How to discount? § Each time we descend a level, we multiply in the discount once § Why discount? § Sooner rewards probably do have higher utility than later rewards § Also helps our algorithms converge

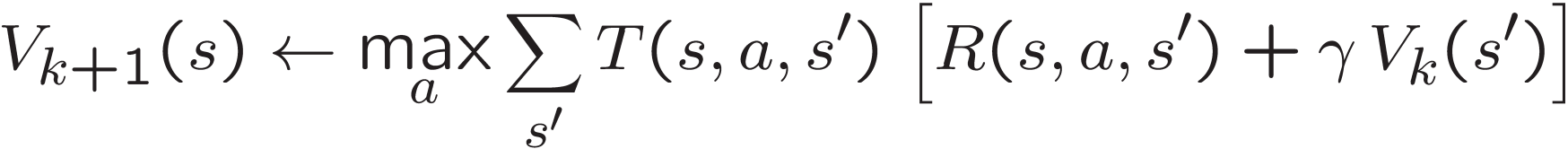
§ Problem: What if the game lasts forever? Do we get infinite rewards? § Solutions: § Finite horizon: (similar to depth-limited search) § Terminate episodes aver a fixed T steps (e.g. life) § Gives nonstationary policies (π depends on time lev) § Discounting: use 0 < γ < 1 § Smaller γ means smaller “horizon” – shorter term focus § Absorbing state: guarantee that for every policy, a terminal state will eventually be reached (like “overheated” for racing)

§ The value (utility) of a state s: V\*(s) = expected utility star)ng in s and acting optimally § The value (utility) of a q-state (s,a): Q\*(s,a) = expected utility starting out having taken action a from state s and (thereafter) acting optimally § The optimal policy: π \*(s) = optimal action from state s



Value Iteration

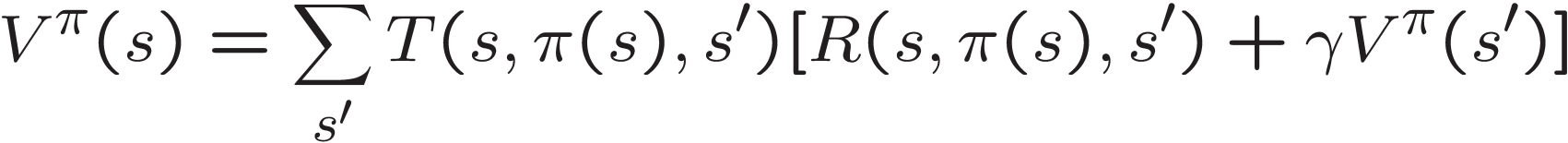
Start with V­0(s) = 0: no time steps left means an expected reward sum of zero, Given vector of Vk(s) values, do one ply of expectimax from each state:



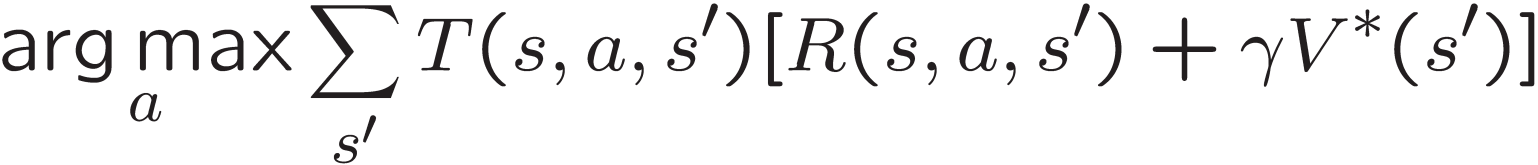
Repeat until convergence. Complexity of each iteration: O(S^2A). Theorem: will converge to unique optimal values

Fixed Policy

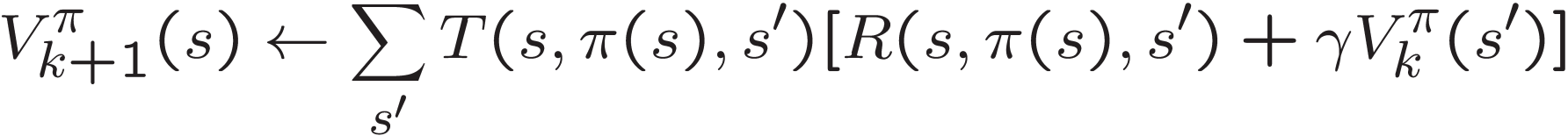
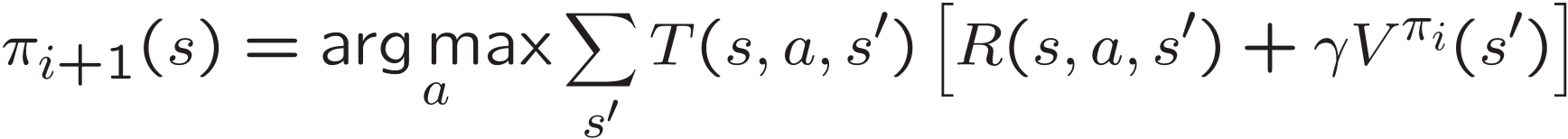
Vπ(s) = expected total discounted rewards starting in s and following



Policy Extraction



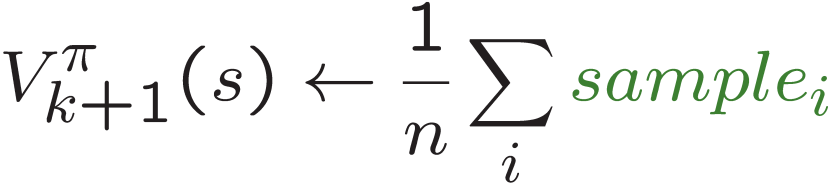
Policy iteration: policy evaluation, then policy improvement

Reinforcement learning: don’t know T or R

♣ Model-Based Idea: ♣ Learn an approximate model based on experiences ♣ Solve for values as if the learned model were correct ♣ Step 1: Learn empirical MDP model ♣ Count outcomes s’ for each s, a ♣ Normalize to give an estimate of ♣ Discover each when we experience (s, a, s’) ♣ Step 2: Solve the learned MDP ♣ For example, use value iteration, as before

Direct Evaluation ♣ Goal: Compute values for each state under π ♣ Idea: Average together observed sample values ♣ Act according to π ♣ Every time you visit a state, write down what the sum of discounted rewards turned out to be ♣ Average those samples

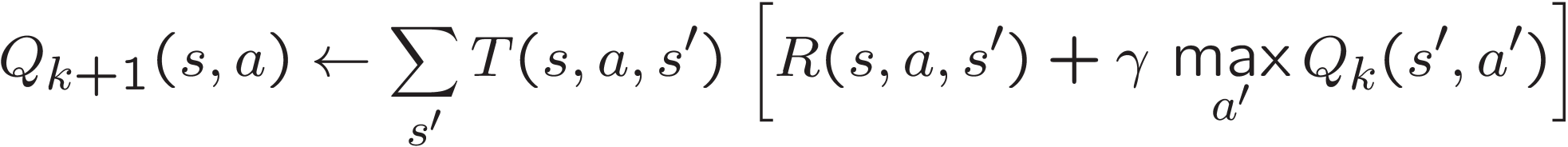
Sample based Policy Evaluation 

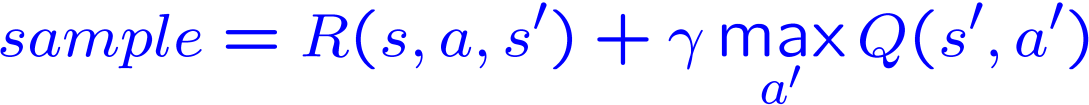
Temporal difference learning:♣ Big idea: learn from every experience! ♣ Update V(s) each time we experience a transition (s, a, s’, r) ♣ Likely outcomes s’ will contribute updates more often



Running interpolation update: \\.host\Shared Folders\Shared with PC\exp_moving_avg_update.png

TD learning can’t turn value into policy-> learn Q value, make action selection model free

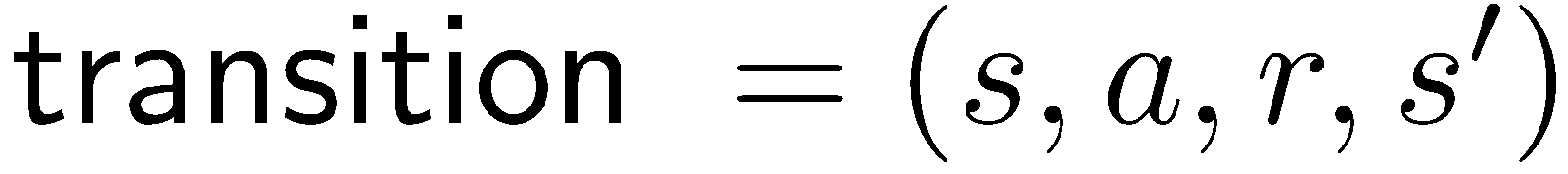
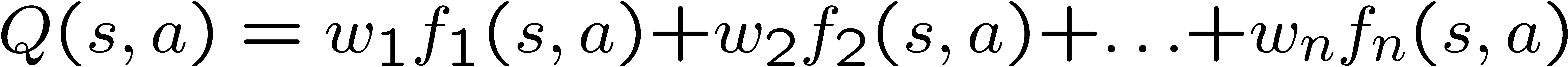
Q value iteration: 

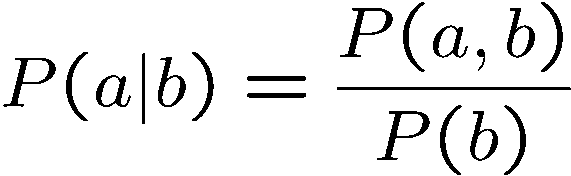
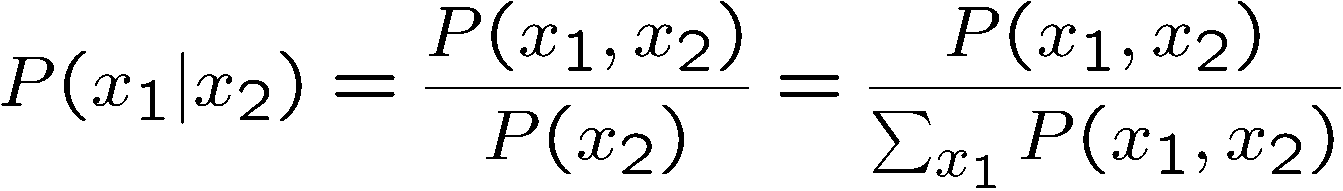
Q learning: 

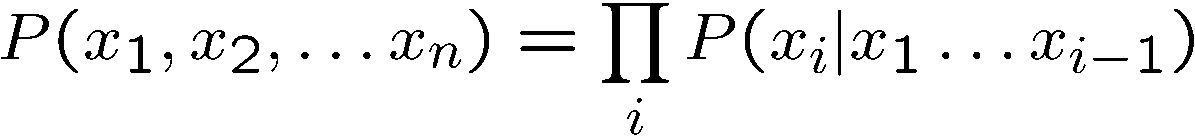
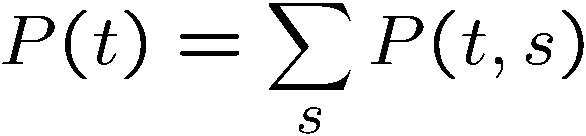
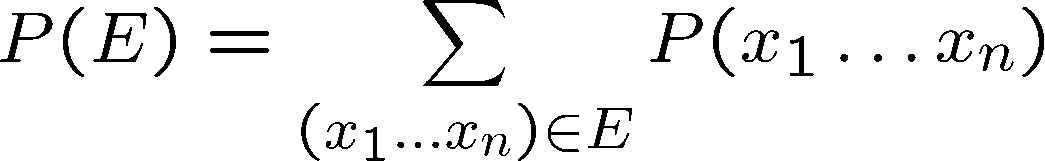


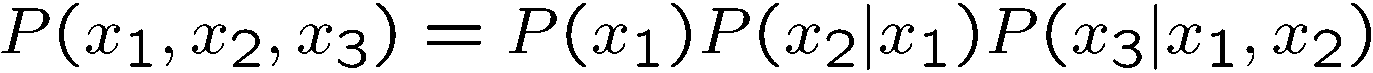
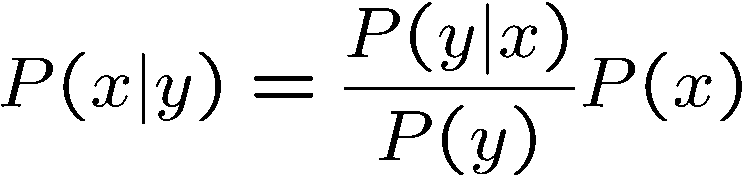
♣ Amazing result: Q-learning converges to optimal policy -- even if you’re acting suboptimally! ♣ This is called off-policy learning ♣ Caveats: ♣ You have to explore enough ♣ You have to eventually make the learning rate small enough ♣ … but not decrease it too quickly ♣ Basically, in the limit, it doesn’t matter how you select actions (!)

Modified Q update (exploration): 

Approximate Q learning: 



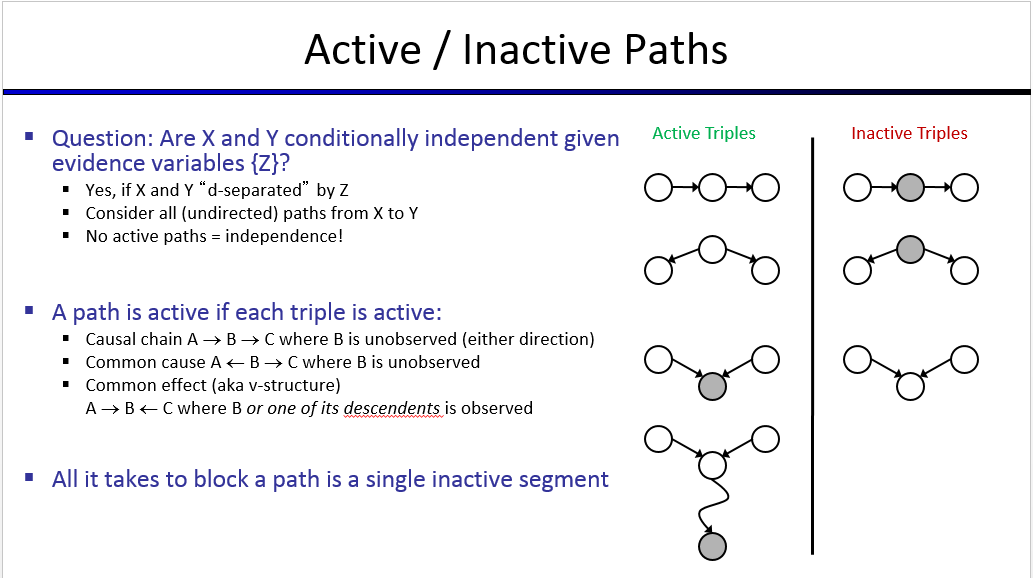
txp_fig.png



txp_fig.png if txp_fig.png or txp_fig.png

txp_fig.png if and only if txp_fig.png or txp_fig.png

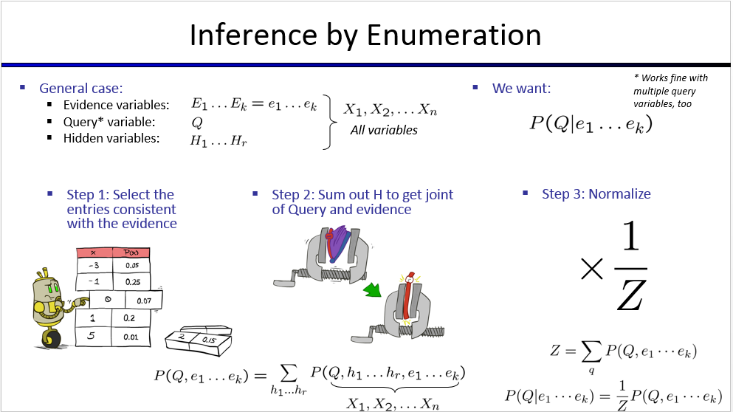
Bayes net A set of nodes, one per variable X § A directed, acyclic graph § A conditional distribution for each node § A collection of distributions over X, one for each combination of parents’ values § CPT: conditional probability table § Description of a noisy “causal” process. A Bayes net = Topology (graph) + Local Conditional Probabilities

Full assignment from BN: txp_fig.png

?

Check all undirected paths between Xi and Xj, if any are active the independence is not guaranteed

Independence is never guaranteed to be false

for bayes net to be able to represent, no additional ind. assumptions

Factors:

Joint distribution P(X,Y)

Selected Joint P(x, Y)

Single Conditional P(Y|x)

Family of Conditionals P(X|Y)

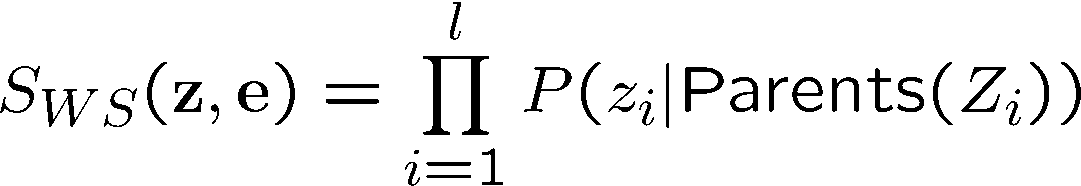
Specified family P(y|X)

# capitals = table dimensionality

Inference by Enumeration: Join all factors, then eliminate hidden variables. Variable Elimination: one at a time

Join: Get all factors over joining variable, build a new factor out of the union. Eliminate: Sum out a variable

Computational complexity critically depends on the largest factor being generated in this process. Size of factor = number of entries in table

Likelihood weighting

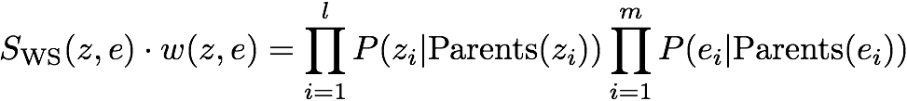
IN: evidence instantiation

w = 1.0

for i=1, 2, …, n

if Xi is an evidence variable

Xi = observation xi for Xi

Set w = w \* P(xi | Parents(Xi))

else

Sample xi from P(Xi | Parents(Xi))

return (x1, x2, …, xn), w

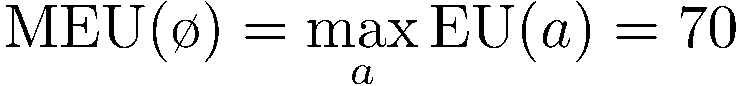
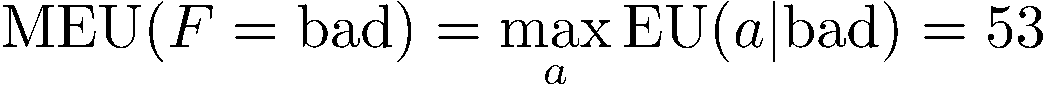
Gibbs Sampling Procedure: keep track of a full instantiation x1, x2, …, xn. Start with an arbitrary instantiation consistent with the evidence. Sample one variable at a time, conditioned on all the rest, but keep evidence fixed. Keep repeating this for a long time. § Property: in the limit of repeating this infinitely many times the resulting sample is coming from the correct distribution § Rationale: both upstream and downstream variables condition on evidence. § In contrast: likelihood weighting only conditions on upstream evidence, and hence weights obtained in likelihood weighting can sometimes be very small. Sum of weights over all samples is indicative of how many “effective” samples were obtained, so want high weight.

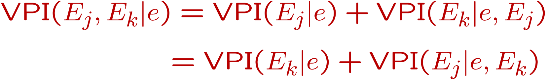
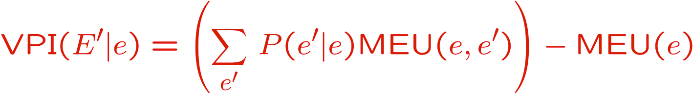
MEU: choose the action which maximizes the expected utility given the evidence

Circle: chance (like BN), rectangle: actions (no parents, observed evidence), diamond: utility (depend on chance and action)

§ Action selection § Instantiate all evidence § Set action node(s) each possible way § Calculate posterior for all parents of utility node, given the evidence § Calculate expected utility for each action § Choose maximizing action

Example with umbrella action, weather 🡪 forecast chance



txp_fig

txp_fig

HMM: underlying Markov chain over states, observe effects at each timestep

Base Case: txp_figtxp_figtxp_fig

txp_figtxp_fig

latex-image-1.pdflatex-image-1.pdflatex-image-1.pdflatex-image-1.pdfPassage of time

latex-image-1.pdf

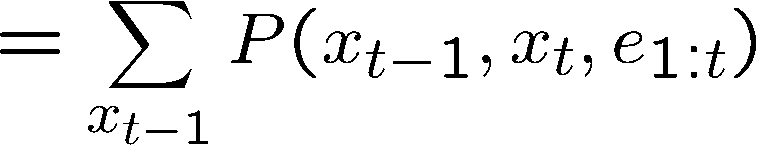
Compactly:

latex-image-1.pdflatex-image-1.pdflatex-image-1.pdfObservation

latex-image-1.pdflatex-image-1.pdf

latex-image-1.pdf

Compactly:

txp_figtxp_figlatex-image-1.pdfForward algorithm: given evidence, want to know txp_fig update time and observation at once

txp_fig

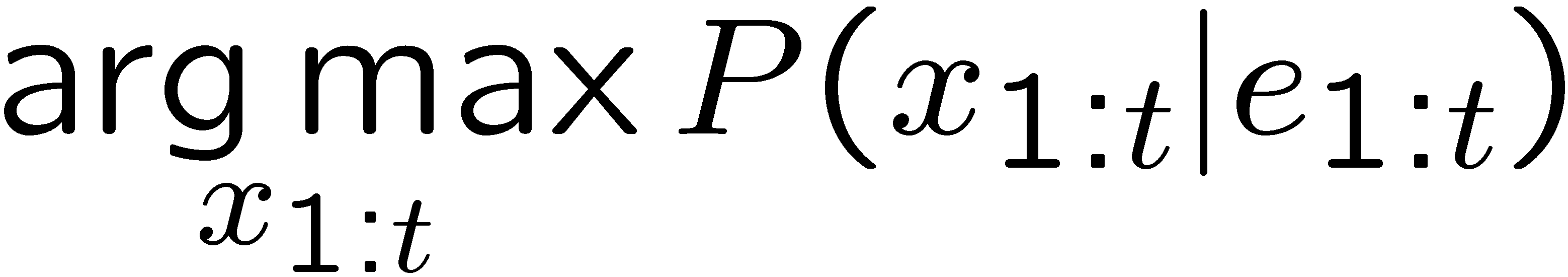
Particles: Our representation of P(X) is now a list of N particles (samples) § Generally, N << |X| § Storing map from X to counts would defeat the point § P(x) approximated by number of particles with value x § So, many x may have P(x) = 0! § More samples, more accuracy § For now, all particles have a weight of 1

Elapse time: txp_fig. Observe: txp_fig, txp_fig. Resample

§ We want to track multiple variables over time, using multiple sources of evidence § Idea: Repeat a fixed Bayes net structure at each time § Variables from time t can condition on those from t-1 § Dynamic Bayes nets are a generalization of HMMs

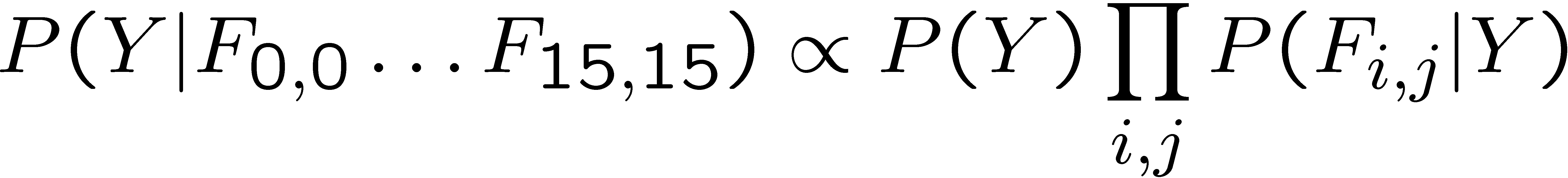
§ Variable elimination applies to dynamic Bayes nets § Procedure: “unroll” the network for T time steps, then eliminate variables until P(XT|e1:T) is computed § Online belief updates: Eliminate all variables from the previous 7me step; store factors for current time only

txp_figA particle is a complete sample for a time step § Initialize: Generate prior samples for the t=1 Bayes net § Example particle: G1 a = (3,3) G1 b = (5,3) § Elapse (me: Sample a successor for each particle § Example successor: G2 a = (2,3) G2 b = (6,3) § Observe: Weight each entire sample by the likelihood of the evidence conditioned on the sample § Likelihood: P(E1 a |G1 a ) \* P(E1 b |G1 b ) § Resample: Select prior samples (tuples of values) in proportion to their likelihood

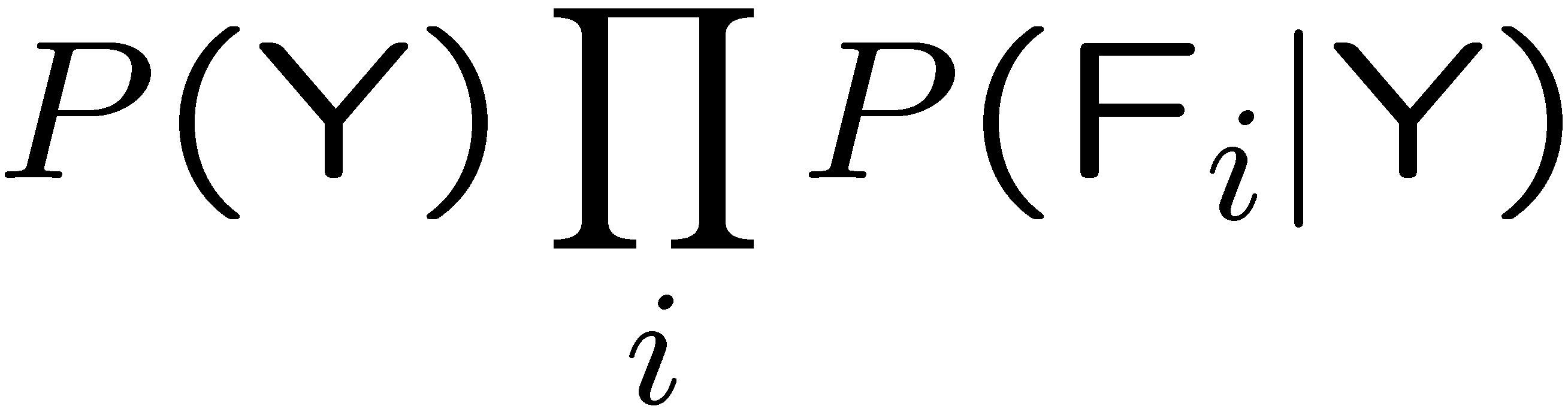
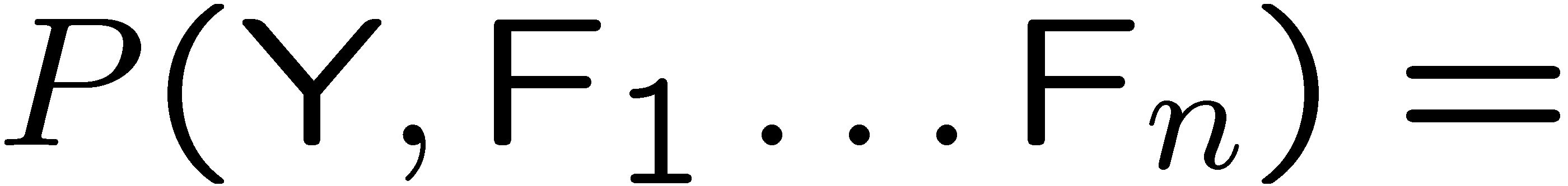
Most Likely Explanation: 

txp_figtxp_figVitterbi: Computes best paths

Naïve Bayes: Assume all features are independent effects of the label § Simple digit recognition version: § One feature (variable) Fij for each grid position § Feature values are on / off, based on whether intensity is more or less than 0.5 in underlying image § Each input maps to a feature vector

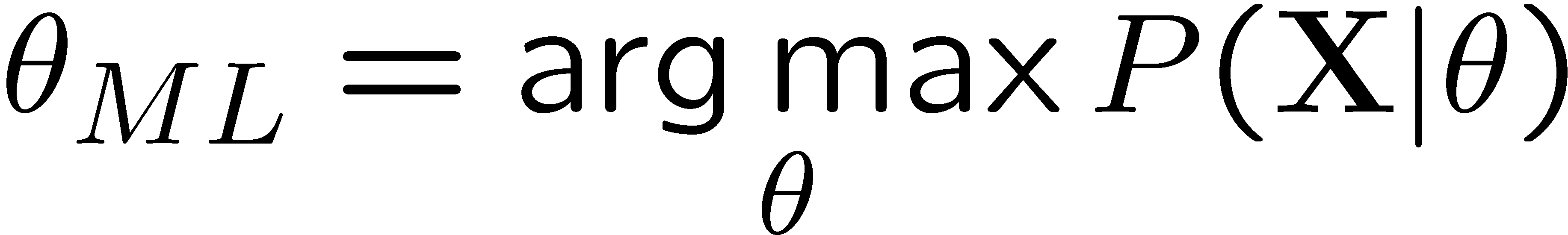
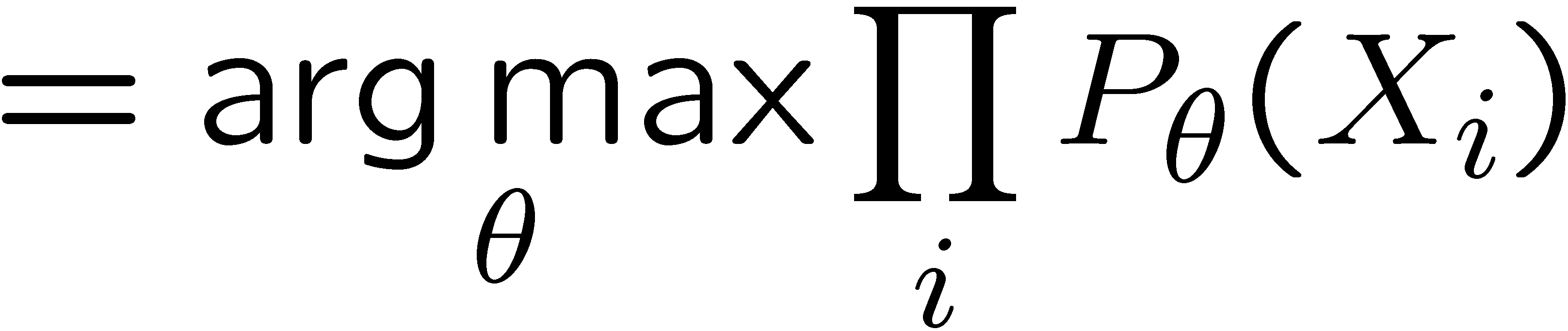
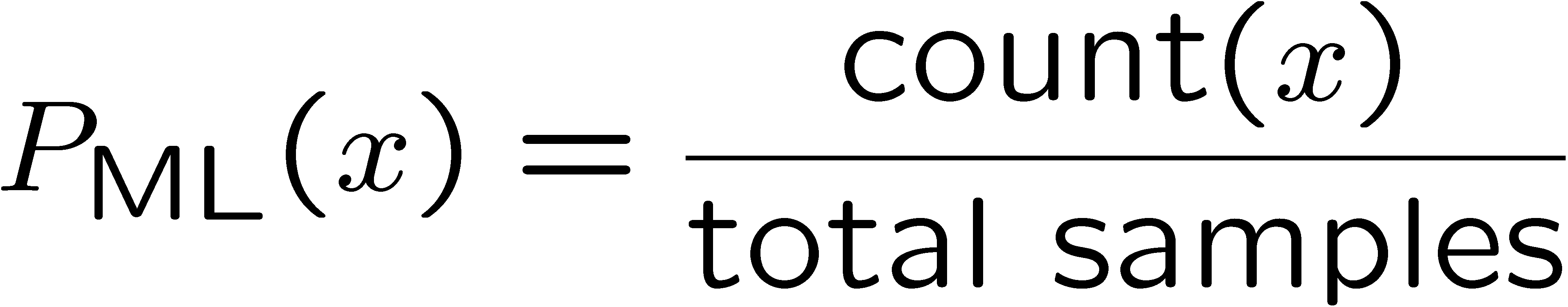
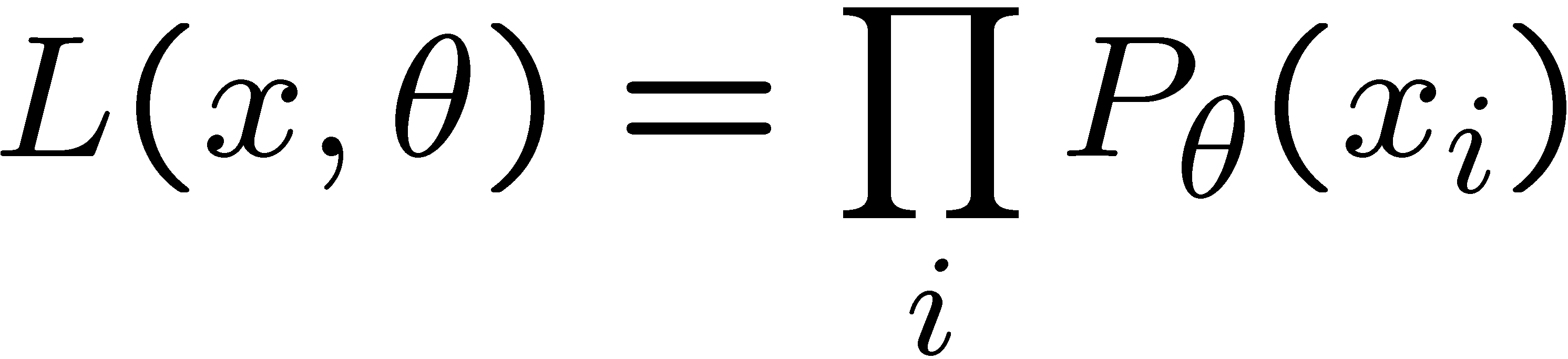


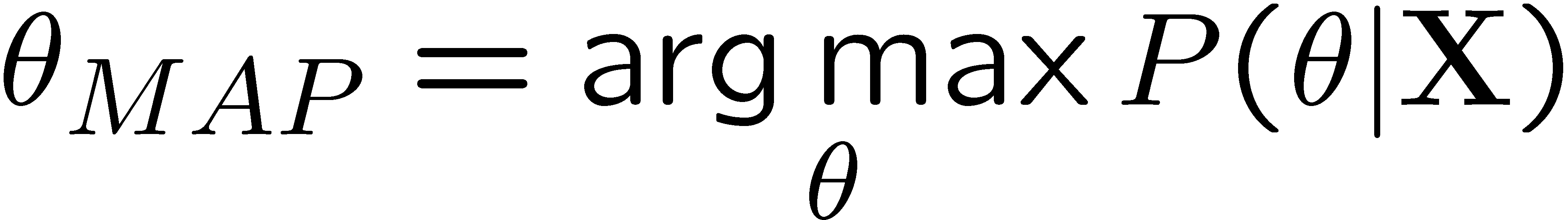
General Naïve Bayes: |Y| parameters

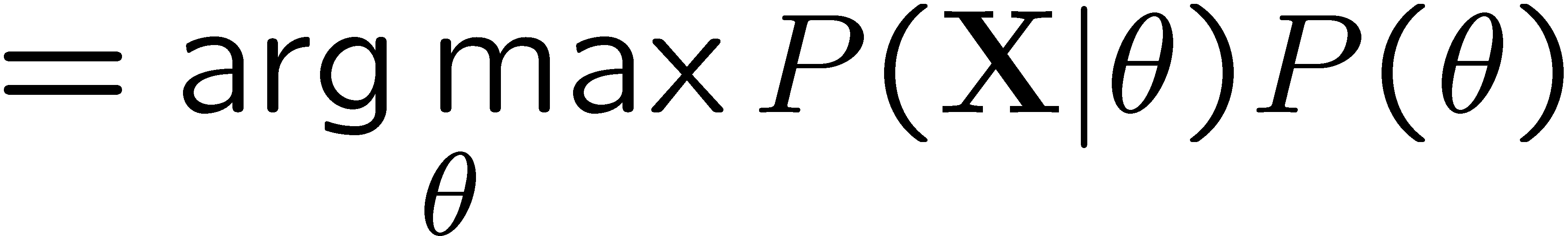


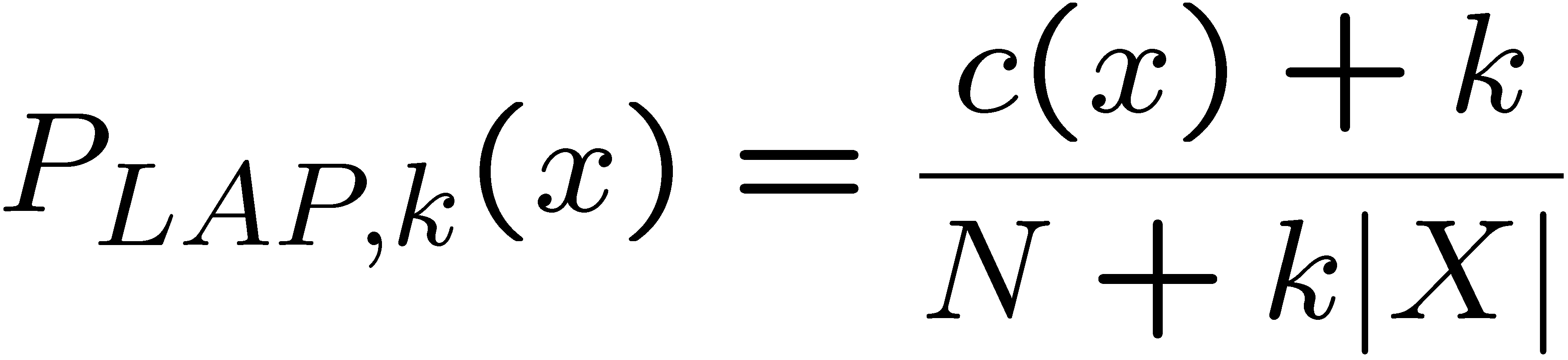
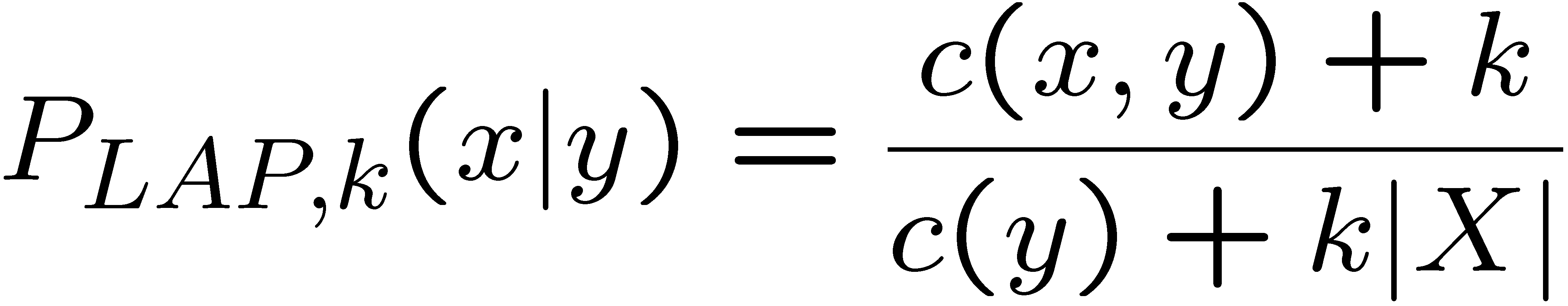
Inference: Step 1: get joint probability of label and evidence for each label § Step 2: sum to get probability of evidence § Step 3: normalize by dividing Step 1 by Step 2

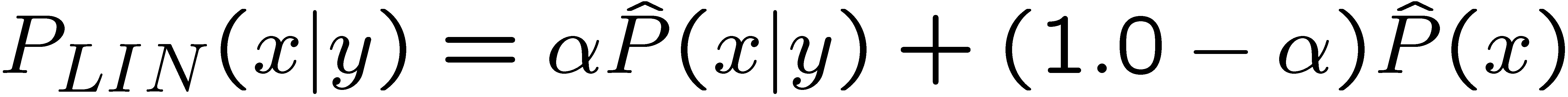
Bag of words: In a bag-of-words model § Each position is identically distributed § All positions share the same conditional probs P(W|Y)

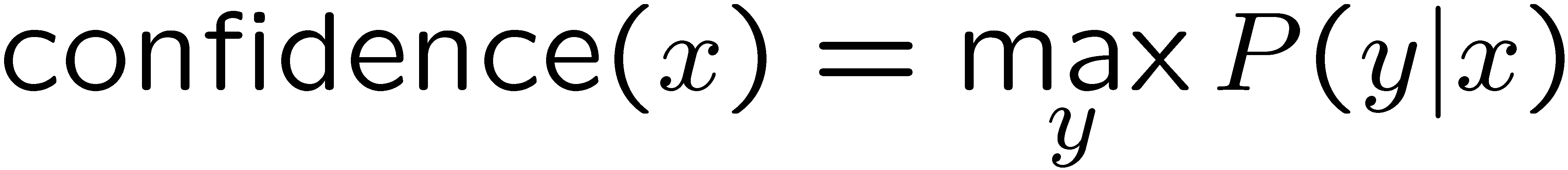
Parameter Estimation: , 

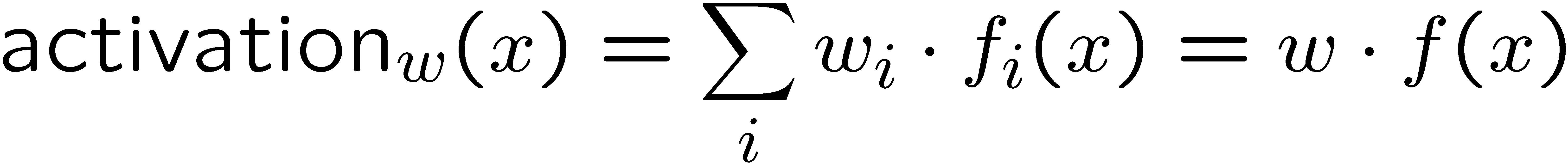
relative frequencies are maximum likelihood estimates:

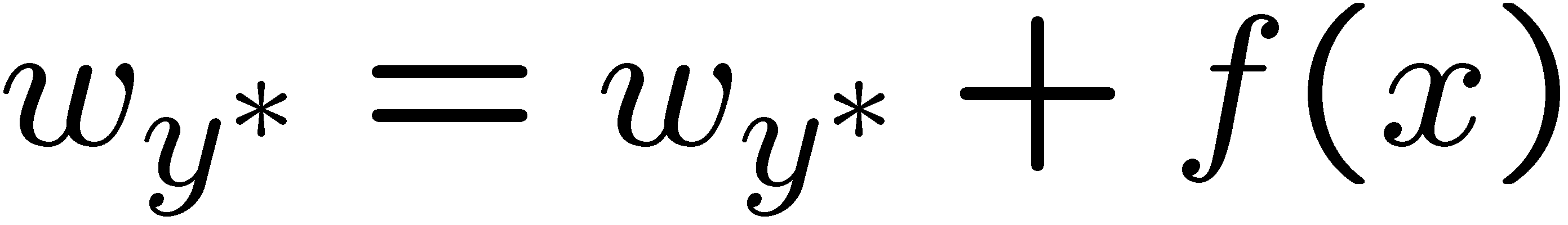
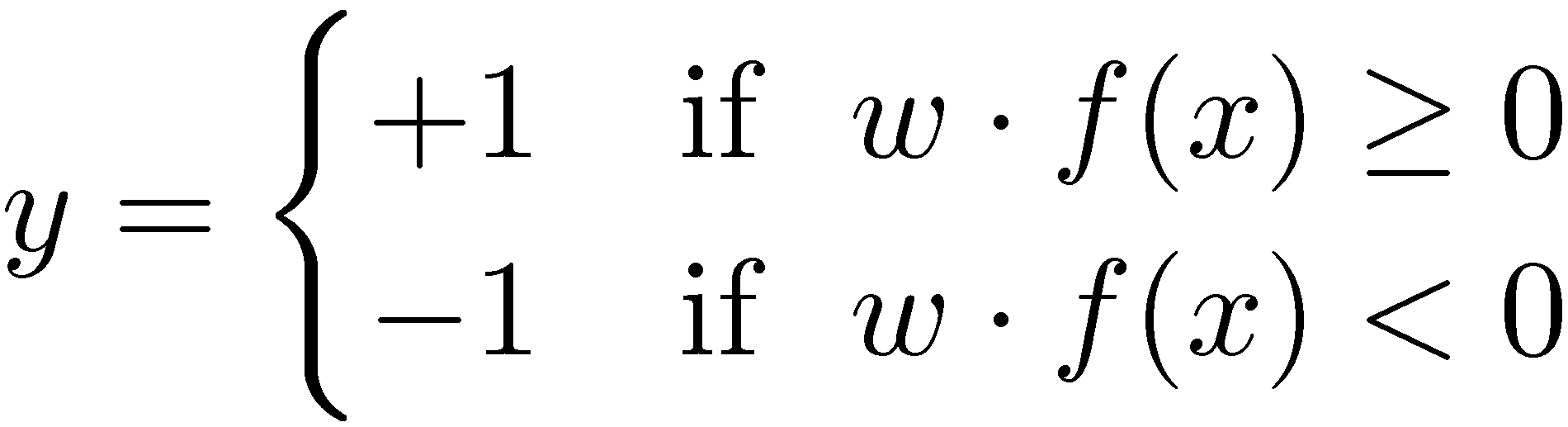
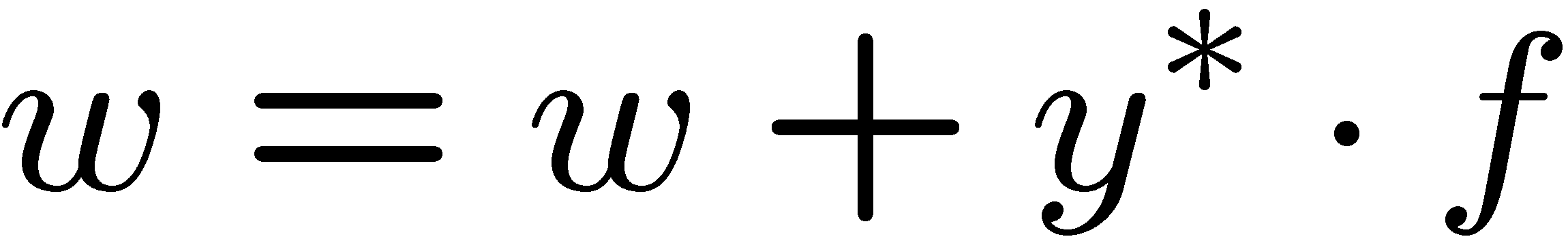
or consider most likely parameter value given data

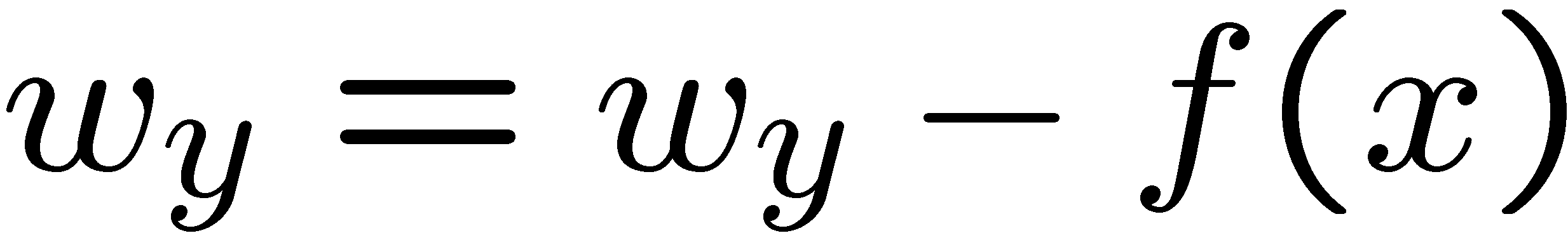
Laplace Smoothing  or for conditionals 

Linear interpolation 



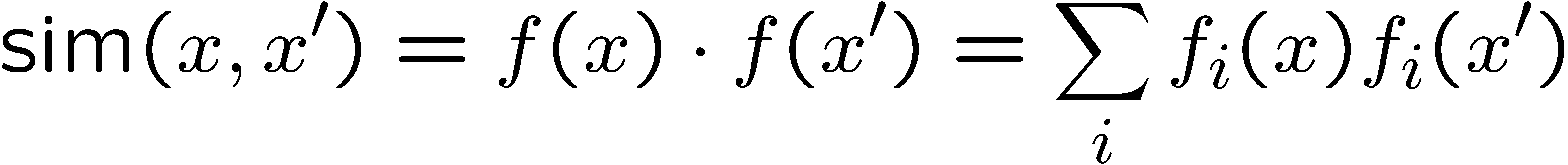
Linear Classifier: 

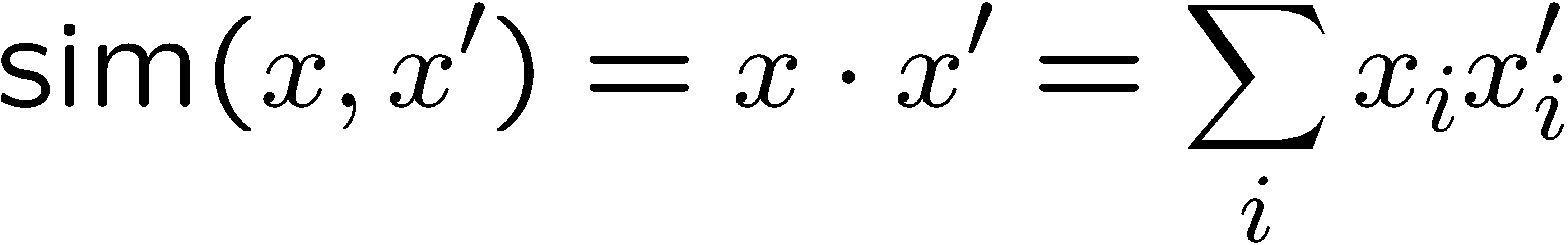
Binary Perceptron: Start with w=0, Classify  . If correct no change, if wrong: 

Multiclass perceptron: classify . If wrong:

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

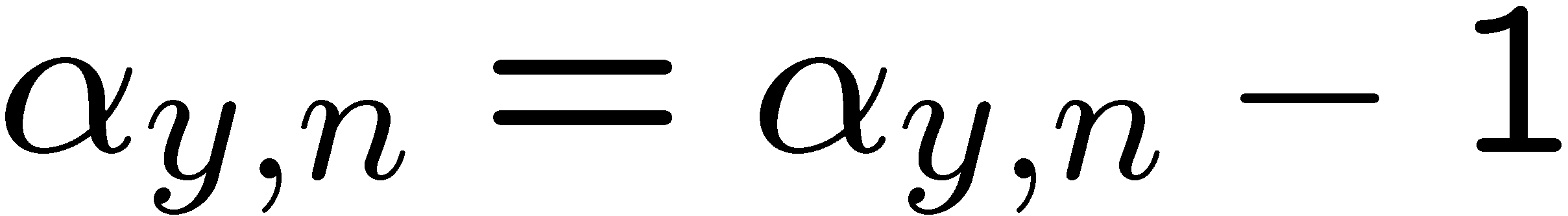
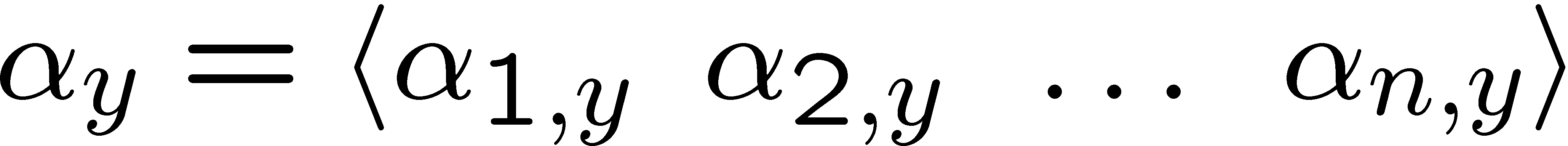
Nearest Neighbor Classification: Nearest neighbor for digits: § Take new image § Compare to all training images § Assign based on closest example

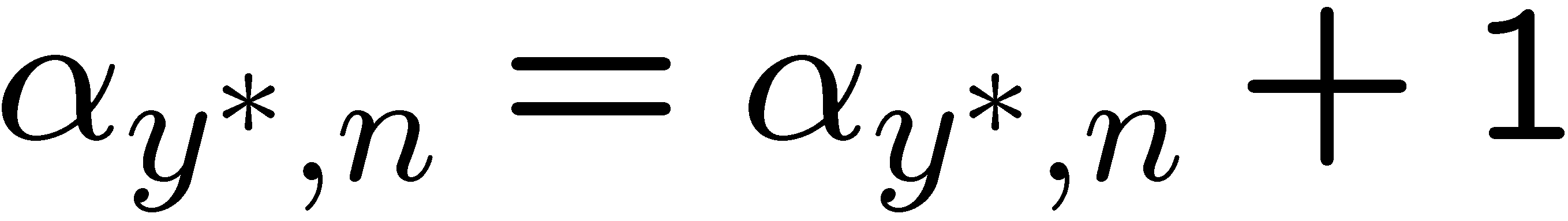
Many similarities based on feature dot products 

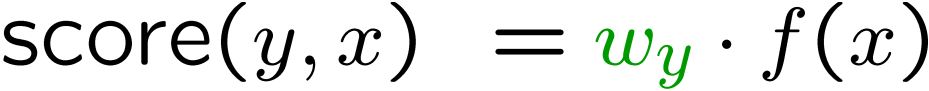
If features are just pixels 

Invariant metrics use knowledge about vision: Rotation, scaling, translation etc. Alternative: training data augmentation

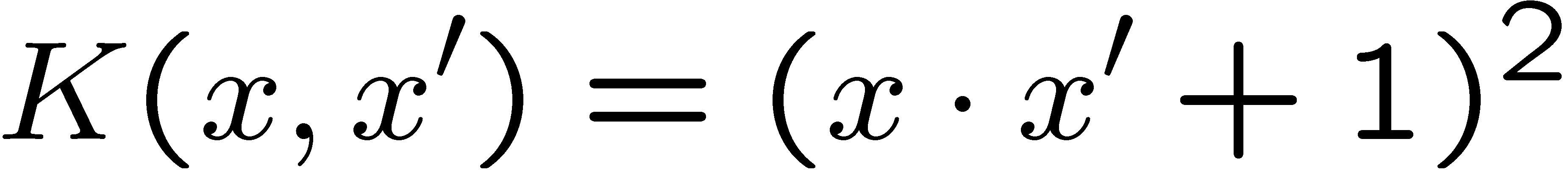
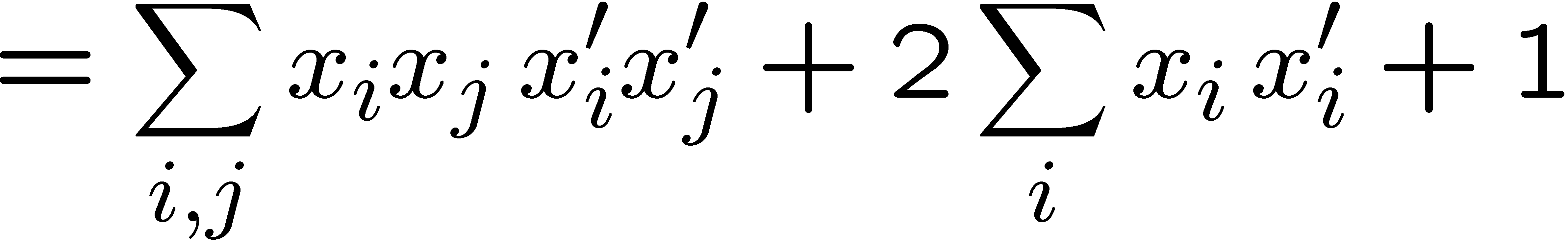
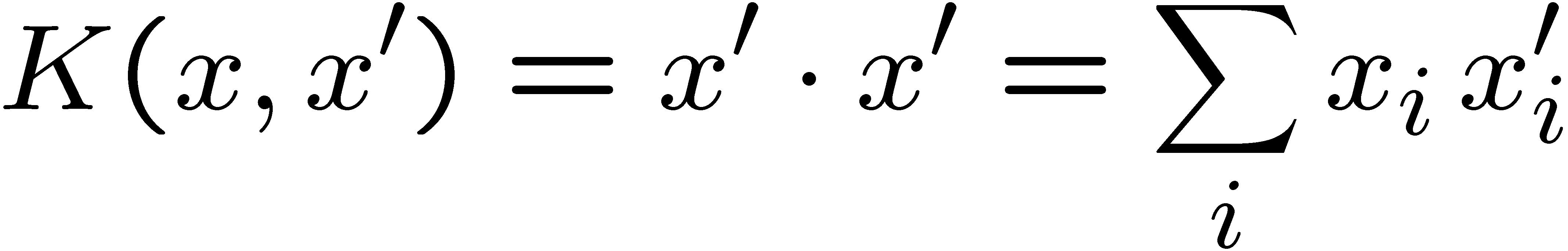
§ Deformable templates: § An “ideal” version of each category § Best-fit to image using min variance § Cost for high distortion of template § Cost for image points being far from distorted template § Used in many commercial digit recognizers

latex-image-1.pdfDual representation of perceptron: update counts 

latex-image-1.pdfStart with 0 counts, try to classify xn’ latex-image-1.pdfif wrong:

Kernelized Perceptron: If we had a black box (kernel) K that told us the dot product of two examples x and x’: Could work entirely with the dual representation

Non Linear separator: Map to higher dimensional space. Kernels implicitly map original vectors to higher dimensional spaces, take the dot product there, and hand the result back

Linear Kernel  Quadratic Kernel

Clustering: group together “similar” instances, i.e. small squared Euclidean distance

K Means: § An iterative clustering algorithm § Pick K random points as cluster centers (means) § Alternate: § Assign data instances to closest mean § Assign each mean to the average of its assigned points § Stop when no points’ assignments change

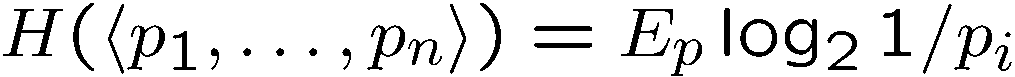
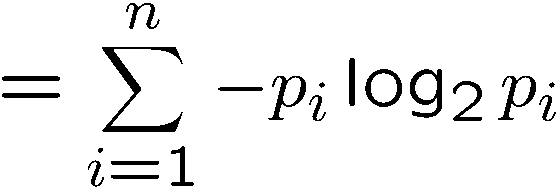
Agglomerative clustering: § First merge very similar instances § Incrementally build larger clusters out of smaller clusters § Algorithm: § Maintain a set of clusters § Initially, each instance in its own cluster § Repeat: § Pick the two closest clusters § Merge them into a new cluster § Stop when there’s only one cluster left

Inductive Learning § Simplest form: learn a function from examples § A target function: g § Examples: input-output pairs (x, g(x)). § Problem: § Given a hypothesis space H § Given a training set of examples xi § Find a hypothesis h(x) such that h ~ g

§ Fundamental tradeoff: bias vs. variance § Usually algorithms prefer consistency by default (why?) § Several ways to operationalize “simplicity” § Reduce the hypothesis space § Assume more: e.g. independence assumptions, as in naïve Bayes § Have fewer, better features / attributes: feature selection § Other structural limitations (decision lists vs trees) § Regularization § Smoothing: cautious use of small counts § Many other generalization parameters (pruning cutoffs today) § Hypothesis space stays big, but harder to get to the outskirts

Decision Trees § Compact representation of a function: § Truth table § Conditional probability table § Regression values

§ What is the expressiveness of a perceptron over these features? § For a perceptron, a feature’s contribution is either positive or negative § If you want one feature’s effect to depend on another, you have to add a new conjunction feature § E.g. adding “PATRONS=full ∧ WAIT = 60” allows a perceptron to model the interaction between the two atomic features § DTs automatically conjoin features / attributes § Features can have different effects in different branches of the tree! § Difference between modeling relative evidence weighting (NB) and complex evidence interaction (DTs) § Though if the interactions are too complex, may not find the DT greedily

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

Entropy

More uniform = higher entropy More values = higher entropy

More peaked = lower entropy Rare values almost “don’t count”

§ Pruning: § Build the full decision tree § Begin at the bottom of the tree § Delete splits in which pCHANCE > MaxPCHANCE § Continue working upward until there are no more prunable nodes § Note: some chance nodes may not get pruned because they were “redeemed” later

Controlling Overfitting: Limit the hypothesis space § E.g. limit the max depth of trees § Easier to analyze § Regularize the hypothesis selection § E.g. chance cutoff § Disprefer most of the hypotheses unless data is clear § Usually done in practice