Total words in spam = 3 + 4 + 6 = 13

Vocabulary size $V = 18$

Using **Laplace smoothing**:

$$P(word|Spam) = \frac{\text{Count(word in spam)} + 1}{\text{Total spam words} + V}$$

E.g.,

- win: appears 2 → $P(win|spam) = \frac{2+1}{13+18} = \frac{3}{31}$
- car: appears 1 → $P(car|spam) = \frac{2}{31}$
- new: appears 1 → $P(new|spam) = \frac{2}{31}$
- now: appears 2 → $P(now|spam) = \frac{3}{31}$

So:

$$P(X|Spam) = P(win|spam) \cdot P(new|spam) \cdot P(car|spam) \cdot P(now|spam)$$

Repeat the same for **Not Spam**, and compare:

$$P(Spam|Email) \propto P(Spam) \cdot P(X|Spam)$$

$$P(NotSpam|Email) \propto P(NotSpam) \cdot P(X|NotSpam)$$

Whichever is **higher** becomes the classification.

2. Depth-First Iterative Deepening (DFID)

Motivation

- Depth-limited search is complete (it will find a goal if it's at depth $\leq d$) but you must know a good bound.
- Breadth-first search is complete and finds the shallowest goal, but uses $O(b^d)$ memory.
- DFID combines the **memory efficiency** of DFS with the **completeness** and **optimality (in terms of depth)** of BFS.

How it works

1. For increasing limits $i = 0, 1, 2, \ldots$:
   a. Run a **Depth-Limited Search** with limit i.
   b. If it returns **success**, stop.
   c. Else if it returns **failure** (no cutoffs), the goal isn't in the tree—stop.
   d. Else (it returned cutoff), increment i and repeat.

1. Depth-Bounded (Depth-Limited) Search

What it is

- A standard DFS, but you **never descend more than** a fixed depth limit d.
- Any node at depth $> d$ is treated as if it has no successors.

How it works

1. Call `DLS(node, depth)`:
    - If `node` is the goal, return success.
    - Else if `depth == 0`, return "cutoff" (we hit the limit).
    - Else for each child c of `node`:
        - Recursively call `DLS(c, depth−1)`.
        - If any call returns success, propagate success.
        - Track if any call returned "cutoff" vs. "failure."
    - If at least one child was cutoff but none succeeded, return "cutoff."
    - Otherwise return "failure."

A **hill-climbing** algorithm maintains a single current state s and repeatedly:

1. **Generate** the "neighboring" states of s.
2. **Select** the best neighbor s' (the one with highest objective/fitness).
3. **If** s' is better than s, **move** $s \leftarrow s'$ and repeat.
4. **Else** (no neighbor is better), **terminate** at the current local maximum.

> ★ Can get stuck on a **local** maximum that isn't the true best.

---

2. The Iterative (Random-Restart) Trick

To overcome local maxima, **Iterative Hill Climbing** simply:

1. **Repeat** R times (for some budget R):
   a. **Pick** a new random **start state** s_0.
   b. **Run** standard hill climbing from s_0 → ends at some local peak s^*.
   c. **Record** the best s^* seen so far.
2. **Return** the overall best of those R final states.

By restarting from diverse points, you cover more of the search space and raise your odds of finding the **global** optimum.

Beam Search is a heuristic graph-search algorithm that explores a subset of the search space in a breadth-first manner, but only keeps the **best k** partial solutions (the “beam”) at each depth. It’s widely used in AI tasks like sequence decoding (machine translation, speech recognition) where the full search would be intractable.

How It Works

1. **Initialize** the beam with just the start state (or empty sequence) and score 0.
2. **Repeat** for each time-step or search depth:
   a. For each partial candidate in the beam, **expand** it to all possible one-step successors.
   b. **Score** each successor (e.g. log-probability under your model).
   c. **Sort** all successors by score, **descending**.
   d. **Prune**: keep only the top k (the beam width), discard the rest.
3. **Terminate** when you reach the desired depth or generate an end-of-sequence token.
4. **Return** the highest-scoring sequence in the beam.

1. Systematic Search: DPLL / CDCL

1.1 DPLL (Depth-First Backtracking with Inference)

1. **Pick** an unassigned variable, say x.
2. **Branch**: try $x = \text{True}$.
3. **Unit propagation**: whenever a clause becomes a "unit" (only one literal unassigned), set that literal so the clause is satisfied.
4. **Pure-literal elimination**: if a variable appears only as y (never $\neg y$) in all clauses, set $y = \text{True}$.
5. **Recurse** on the simplified formula.
6. **Backtrack** when you derive a contradiction (an empty clause), flip your last decision ($x = \text{False}$), repeat.
7. **Terminate** when either all clauses are satisfied (SAT) or you've backtracked past the first decision (UNSAT).

1.2 CDCL (Conflict-Driven Clause Learning)

An enhanced DPLL that, upon detecting a conflict:

- **Analyzes** the conflict to learn a new clause (a "lemma") that prevents the same bad partial assignment in the future.

2. Formula in Conjunctive Normal Form (CNF)

Most SAT solvers expect the formula in **CNF**: a conjunction (AND) of clauses, each clause is a disjunction (OR) of literals (a variable or its negation). Example:

$$(x_1 \vee \neg x_2 \vee x_3) \ \wedge \ (\neg x_1 \vee x_2) \ \wedge \ (x_2 \vee \neg x_3)$$

Here:

- Clause 1: $x_1 \vee \neg x_2 \vee x_3$
- Clause 2: $\neg x_1 \vee x_2$
- Clause 3: $x_2 \vee \neg x_3$

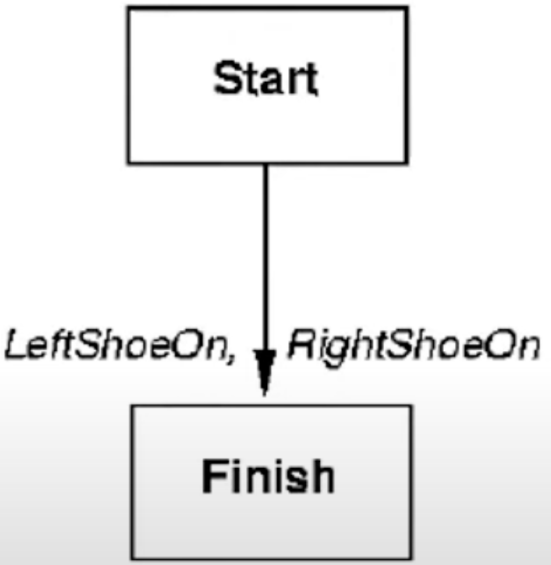
A CNF formula is **satisfiable** if and only if **every** clause can be made true by the same assignment of the x_i.

Next...

- Successor function
  - Arbitrarily picks one open precondition p on an action B and generates a successor plan, for every possible consistent way of choosing an action A, that achieves p
- Consistency:
  - Causal link $A \xrightarrow{p} B$ and the ordering constraint are added ( ) $(A \prec B \quad Start \prec A \quad A \prec Finish)$
  - Resolve conflict: add $B \prec C$ or $C \prec A$

The Initial Plan

- Initial plan contains:
- Start:
  - PRECOND: none
  - EFFECT: Add all propositions that are initially true
- Finish:
  - PRECOND: Goal state
  - EFFECT: none
- Ordering constraints: $Start \prec Finish$
- Causal links: {}
- Open preconditions:
  - {preconditions of Finish}



How to Define Partial Order Plan?

- A set of actions, that make up the steps of the plan
- A set of ordering constrain $A \prec B$
  - A before B
- A set of causal links: $A \xrightarrow{p} B$
  - A achieves P for B $RightSock \xrightarrow{RightSockOn} RightShoe$
  - May be conflicts if C has the effect of ¬P and if C comes after A and before B
- A set of open preconditions:
  - A precondition is open, if it is not achieved by some action in the plan

5. **Pop** → **Clear(P)** is true → do nothing.
6. **Pop** → **Holding(O)** false → choose **pickup(O)**, push it, then its preconds:

```mathematica
pickup(O)
[OnTable(O), Clear(O), ArmEmpty]
```

7. Pop/solve each literal in turn by executing when ready:
   - Pop ArmEmpty, Clear(O), OnTable(O): all true → do nothing.
   - Pop **pickup(O)** → all its preconds now hold → **execute** it, append to plan, update S.
8. Continue popping: next is **stack(O,P)** → preconds now hold → **execute** it.
9. Next pop is **On(T,O)**. It's still false → choose **stack(T,O)**, push its preconds and so on.
10. You may need to **unstack(P,T)** and/or **putdown(P)** to make **Clear(T)**, then **pickup(T)**, then **stack(T,O)**.

If you serialize your pushes carefully (e.g. tackle **On(O,P)** first, then **On(T,O)**, and within each action push the "armempty" precond last), you'll get exactly the six-step plan of part (a).

(c) Goal-Stack Planning (GSP)

We maintain:

- a **stack** of pending subgoals & actions
- the **current state** S
- the **plan** (actions in execution order)

We push the whole goal conjunction $\{\text{On}(T,O), \text{On}(O,P)\}$ and work depth-first:

1. **Push** $\{On(T,O), On(O,P)\}$.
2. **Pop** the conjunction → re-push it (to check later), then **push** its literals in some order. Suppose we push first **On(T,O)**, then **On(O,P)**.
3. **Pop** → gets **On(O,P)** which is false in S_0. **Choose** action **stack(O,P)**, push it, then push its preconditions:

   scss — Copy — Edit

   ```
   stack(O,P)
   [Holding(O), Clear(P), ArmEmpty]
   ```

4. **Pop** → **ArmEmpty** is true already → do nothing.
5. **Pop** → **Clear(P)** is true → do nothing.

- **Heuristic Forward Search** (Fast-Downward, FF)
  Uses *relaxed-plan* heuristics that implicitly reason about subgoal interactions and avoid paths that permanently destroy other subgoals.

In a Nutshell

> **Sussman's Anomaly** shows that in domains where actions can undo each other, you cannot simply pick a static order of subgoals. You need a planner that reasons about **which actions can safely interleave**, or explicitly records and enforces the minimal necessary ordering constraints between all actions.

3. What “Anomaly” Teaches Us

1. **Subgoals can interact.**
   You can’t treat them as independent—achieving one can *block* or *undo* another.
2. **No fixed linearization** of the goal conjuncts works.
   This breaks planners that simply push each literal in some order onto a stack and solve them one-by-one.
3. **Need for non-linear planning**:
   - **Interleaving** actions so you never fully “commit” to one subgoal in isolation.
   - **Planning graphs** (e.g. Graphplan) or **partial-order planners** that allow actions for different subgoals to be interwoven, with explicit “don’t interfere” constraints.

---

4. How Modern Planners Handle It

- **Partial-Order Planning (POP)**
  Builds a network of actions with only the necessary ordering constraints (“A before B if they conflict”), allowing independent subgoals to interleave freely.
- **Planning Graph & Mutex Analysis** (Graphplan/Blackbox)
  Detects pairs of subgoals or actions that *cannot* co-occur, forcing the planner to schedule them in a safe interleaving.

2. Why No Serial Ordering Works

A “serial” planner picks an order—say

1. **First** achieve On(A,B),
2. **Then** achieve On(B,C).

But:

1. To get **On(A,B)** you do
   - `pickup(A)`, `stack(A,B)`.
   - New state: A atop B, B no longer clear, C untouched.
2. Next goal **On(B,C)** needs you to do
   - `pickup(B)` (or `unstack(B,A)`) then `stack(B,C)`.
   - But **picking up B** necessarily **undoes** `On(A,B)` (you must remove A from B first!).

If you reverse the order—

1. First do **On(B,C)** by `pickup(B)`, `stack(B,C)`.
2. Then do **On(A,B)** by `pickup(A)`, `stack(A,B)`—

you undo **On(B,C)** in step 2 (because you must pick up B off C before stacking A on B).

In **both** serial orders, the second subgoal wrecks th[illegible]st.

Sussman's Anomaly is the canonical example in the Blocks-World that shows why **"linear"** or **"goal-stack"** planners that tackle one subgoal at a time can fail: the subgoals **interact** so that achieving one undoes the other, and *no* fixed ordering of "do A then B" or "do B then A" works.

---

1. The Setup

- **Blocks**: A, B, C
- **Initial state**

```scss
OnTable(A), OnTable(B), OnTable(C),
Clear(A), Clear(B), Clear(C), ArmEmpty
```

- **Goal**

```scss
On(A, B) ∧ On(B, C)
```

  i.e. make a stack A→B→C.

3. The “Extra Check” Trick

In pure GSP you only check *individual* literals for truth before popping them off the stack. To avoid endlessly undoing subgoals, one can insert an **extra global check** each time you finish solving all *n* subgoals:

> “Has the *entire* goal conjunction really been achieved in the current state S?
> If not, clear the stack and start again from S with the full conjunction.”

That guarantees termination in a **reversible** domain like Blocks World, because eventually you’ll reach a state from which the whole goal is true, and you’ll stop. But beware:

- If your goal conjunction is **inconsistent** (e.g. `On(A,B) ∧ On(B,A)`), you’ll loop forever, constantly retrying.

2.2 Next subgoal: On(B,C)

- GSP now pushes the preconditions of `stack(B,C)`: `Holding(B)`, `Clear(C)`, `ArmEmpty`.
- But to get `Holding(B)` the planner must `pickup(B)`, which in turn requires `Clear(B)` and `ArmEmpty`.
- **Pickup(B)** would disturb the partial solution `On(A,B)`—because to pick B off the table you have to *unstack* A from B or lift B from under A, thereby undoing On(A,B)!

What you see in **Figure 7.10** (and the accompanying trace) is exactly that: whichever of the two goals you try to satisfy first, the second one forces you to undo the first, and GSP on its own will get stuck in an infinite loop of "achieve A then B, then achieve B then A, ..."

1. `Holding(A)`
2. `Clear(B)`
3. `ArmEmpty`

It then works those subgoals one by one—ultimately doing
`pickup(A)`,
`stack(A,B)`
and marking **On(A,B)** solved.

- **But** that action has changed the world to:

```mathematica
S₁ = { On(A,B), OnTable(C), Clear(A), Clear(C), Clear(B)? no… }
```

so **Clear(B)** and **ArmEmpty** get re-established, but the other goal **On(B,C)** is still pending.

2.2 Next subgoal: On(B,C)

- GSP now pushes the preconditions of `stack(B,C)`:
`Holding(B)`, `Clear(C)`, `ArmEmpty`.
- But to get `Holding(B)` the planner must `pickup(B)`, which in turn requires `Clear(B)` and `ArmEmpty`.

2. A Worked Trace on Sussman's Anomaly

The **Anomaly** initial state is:

```mathematica
S₀ = { OnTable(A), OnTable(B), OnTable(C),
       Clear(A), Clear(B), Clear(C), ArmEmpty }
```

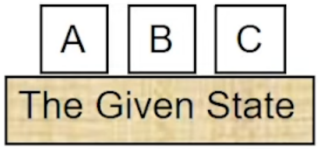
and the **goal** is

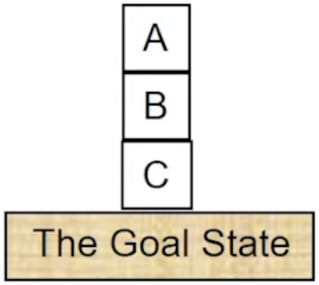
```ini
G = { On(A, B), On(B, C) }.
```

2.1 First Choice: Which subgoal first?

- Suppose we push and tackle **On(A,B)** before **On(B,C)**.
- GSP will *stack* up the preconditions of `stack(A,B)`:
  1. `Holding(A)`
  2. `Clear(B)`
  3. `ArmEmpty`

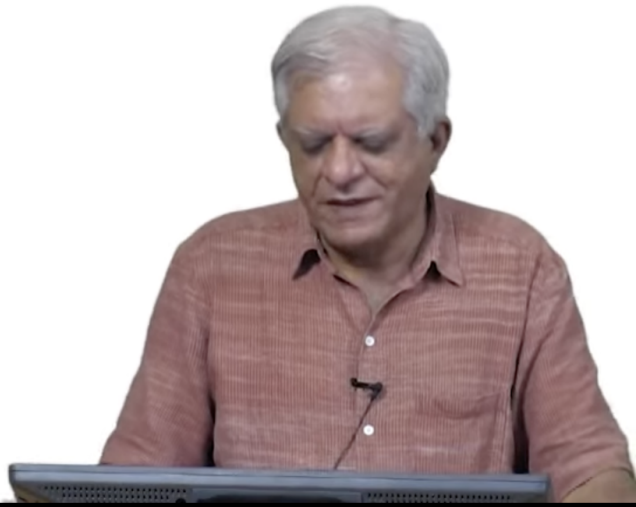
BSSP: Spurious Action Sequences





Search terminates because the regressed goal is true in the given state

<pickup(B), pickup(A), stack(B,C), stack(A,B)> is not a valid plan.



Backward State Space Planning (BSSP)

Relevant actions: Given a goal G an action *a* is *relevant to the goal* if it produces some positive effect in the goal, and deletes none. That is,

$$\{effect^{+}(a) \cap G\} \neq \phi \ \wedge \{effects^{-}(a) \cap G\} = \phi$$

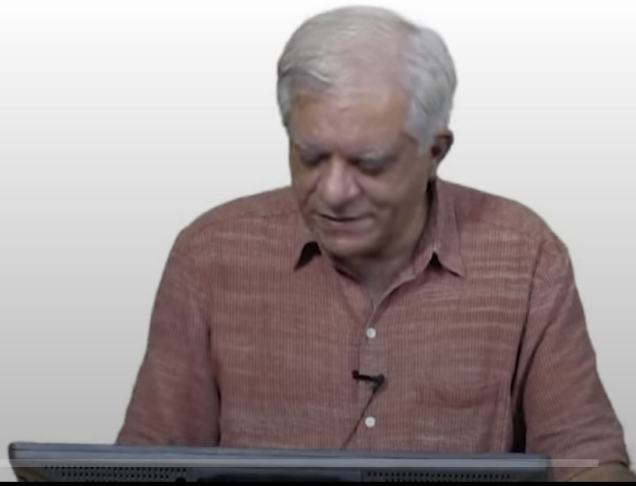
Regression: If a relevant action *a* is applied in a goal G then the goal *regresses* to a new goal G', defined as,

$$G' = \gamma^{-1}(G, a)$$
$$= \{G \setminus effects^{+}(a)\} \cup pre(a)$$

Plan: A plan π is a sequence of actions $<a_1, a_2, ..., a_n>$. A plan π is *relevant* in to a goal G_n if there are goals $G_1,...,G_{n-1}$ such that $G_{i-1} = \gamma^{-1}(G_i, a_i)$ for $i = 1,...,n$. The final goal is $G_1 = \gamma^{-1}(G_n, \pi)$

Valid plan: Let S_0 be the start state. Regression ends when $G_1 \subseteq S_0$
The plan π is a valid plan if $G_n \subseteq \gamma(S_0, \pi)$

note: validity still checked by progression



Forward State Space Planning (FSSP)

Applicable actions: Given a state S an action *a* is *applicable* in the state if its preconditions are satisfied in the state. That is,

$$pre(a) \subseteq S$$

Progression: If an applicable action *a* is applied in a state *S* then the state *transitions* or *progresses* to a new state S', defined as,

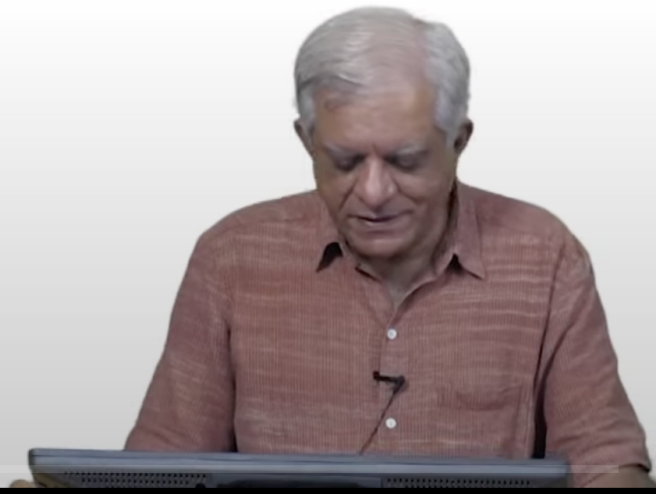
$$S' = \gamma(S, a)$$
$$= \{S \cup effects^{+}(a)\} \setminus effects^{-}(a)$$

Plan: A plan π is a sequence of actions $\langle a_1, a_2, \ldots, a_n \rangle$. A plan π is *applicable* in a state S_0 if there are states $S_1,\ldots,S_n$ such that $\gamma(S_{i-1},a_i)=S_i$ for $i = 1,\ldots,n$.

The final state is $S_n = \gamma(S_0,\pi)$.

Valid plan: Let G be a goal description.
Then a plan π is a valid plan in a state state S_0 if,

$$G \subseteq \gamma(S_0,\pi)$$



3. Key Differences

Aspect	Forward Planning	Backward (Regression) Planning
Direction	From **initial** to goal	From **goal** to initial
Search nodes	**World states** (complete assignments)	**Subgoal sets** (conjunctions of facts)
Action applicability	Must check action's preconditions in state	Must check that action's effects cover a subgoal literal
When useful	• Many irrelevant goals • Easily used with **state-based heuristics** (e.g. distance to goal) • Better when initial state is small	• When goal conjuncts are few • When actions have few effects • Can avoid exploring irrelevant parts of the state space
Plan extraction	Trace parent pointers from goal-state back	Record regression steps and then reverse

- **Regression step 2**:
  - Now handle `On(B,C)`. You need `Stack(B,C)`, which requires `Clear(B), Clear(C), HandEmpty`.
  - Remove `On(B,C)` and add its preconds:

```scss
{ Clear(A), Clear(B), Clear(C), HandEmpty }
```

- **Regression step 3**:
  - Every literal in this set holds in the **initial state** (`A,B,C` all start clear, hand is empty).
  - So we're done regressing.

To extract the plan, you note the reverse order of the actions you picked:

1. `Stack(B,C)`
2. `Stack(A,B)`

which is exactly the same two moves, but discovered "backwards."

5. Store the fact that you regressed over a (so you can build the plan in reverse).
6. Repeat until some subgoal set is covered by the initial state.

Block-World Example

- **Goal**: `On(A,B), On(B,C)`
- **Regression step 1**:
  - Pick subgoal `On(A,B)`. The only action that makes `On(A,B)` true is `Stack(A,B)`, whose preconditions are
    - `Clear(A)`
    - `Clear(B)`
    - `Holding(A)` *if we model a hand*, or "arm empty" + `PickUp(A)`.
  - Remove `On(A,B)` from the goal and add `Clear(A), Clear(B), HandEmpty` → new subgoal set:

```scss
{ On(B,C), Clear(A), Clear(B), HandEmpty }
```

Forward search might go:

1. From `OnTable(A),…` pick action `Stack(A,B)` → new state has `On(A,B)`.
2. From that state pick `Stack(B,C)` → new state has both `On(A,B)` and `On(B,C)` → goal reached.

It explores actions in the real order you'd execute them.

---

2. Backward (Regression) Planning

- **Idea**: Start from the **goal description**, "regress" subgoals over actions to find what **must** have been true one step earlier—until those preconditions are satisfied by the initial state.
- **Algorithm sketch**:
  1. Put the goal clause set G in your open list.
  2. Remove a subgoal set G'. If every literal in G' holds in the initial state, you've found a plan.
  3. Otherwise, pick some literal $l \in G'$ that isn't true initially, and pick an action a that achieves l.
  4. **Regress**: form a new subgoal set

$$G'' = (G' - \{l\}) \cup \text{Preconditions}(a).$$

1. Forward (Progression) Planning

- **Idea**: Start in the **initial state**, apply actions "forwards" until you reach a state that **satisfies the goal**.
- **Algorithm sketch** (e.g. BFS, A*):
  1. Put the initial state S_0 in your open list.
  2. Remove a state S, test if it meets the goal; if yes, reconstruct the action sequence.
  3. Otherwise, expand S: for each action a whose preconditions hold in S, generate successor state S' = result(S, a); enqueue S'.
  4. Repeat until you find a goal or exhaust the space.

Block-World Example

- **Blocks**: A, B, C
- **Initial**:
  - `OnTable(A), OnTable(B), OnTable(C), Clear(A), Clear(B), Clear(C)`
- **Goal**:
  - `On(A,B), On(B,C)`

Forward search might go:

- And similarly $P(a \mid \text{non-enzyme})$.

4. **Predicting** a new protein "MKTAYIAK…":
   - Count relative frequencies of each amino acid.
   - Compute

     $$\log P(\text{enzyme}) + \sum_{a \in \mathcal{A}} (\#a)\, \log P(a \mid \text{enzyme}) \quad \text{vs.} \quad \log P(\text{non-enzyme}) + \sum_{a} (\#a)\, \log P(a \mid \text{non-enzyme}).$$

   - Classify according to the larger score.

If you go beyond single-residue frequencies, you can define features as "dipeptides" ("MK," "KT," etc.), or even short motifs, increasing the feature dimension—and still apply exactly the same naïve Bayes formulas (with smoothing).

Why It Works

- **Speed & Scalability**: just counts and products (or sums of logs).
- **Robustness**: often performs competitively even when independence is violated.
- **Interpretable**: you can inspect which words or amino acids drive each class's likelihood.

Limitations

3. Example B: Protein Sequence Classification (Enzyme vs. Non-enzyme)

1. **Features**: amino-acid composition or dipeptide (pair) counts.
   - Let $\mathcal{A} = \{\mathrm{A}, \mathrm{C}, \ldots, \mathrm{Y}\}$ be the 20 amino acids.
   - A simple feature vector is

$$x_i = \frac{\text{count of amino acid } i \text{ in the protein}}{\text{sequence length}}.$$

2. **Classes**:
   - $c_1 =$ "enzyme,"
   - $c_2 =$ "non-enzyme."
3. **Training**:
   - Compute the prior $P(\text{enzyme})$ = #enzymes / total proteins.
   - For each amino acid a, estimate

$$P(a \mid \text{enzyme}) = \frac{\sum_{\text{seq} \in \text{enzyme}} \big[\#a \text{ in seq}\big] \; + \; 1}{\sum_{\text{seq} \in \text{enzyme}} \big[\text{length(seq)}\big] \; + \; 20}.$$

   - And similarly $P(a \mid \text{non-enzyme})$.
4. **Predicting** a new protein "MKTAYIAK...":

2. Example A: Text Classification (Spam vs. Ham)

1. **Features**: presence or count of words ("free," "win," "meeting," ...).
2. **Classes**:
   - $c_1 =$ "spam,"
   - $c_2 =$ "ham" (legitimate).
3. **Training**:
   - Compute $P(\text{spam})$ = #spam emails / total emails.
   - For each word w, count how often it appears in spam vs. ham, then compute

$$P(w \mid \text{spam}) = \frac{\text{count}_{w,\text{spam}} + 1}{\sum_{w'} \text{count}_{w',\text{spam}} + V},$$

     and similarly for $P(w \mid \text{ham})$.
4. **Predicting** a new email "Win money now":
   - Tokenize into words: $\{\text{"Win"}, \text{"money"}, \text{"now"}\}$.
   - Compute score for each class:

$$\log P(\text{spam}) + \sum_{w \in \{\ldots\}} \log P(w \mid \text{spam}) \quad \text{vs.} \quad \log P(\text{ham}) + \sum_{w} \log P(w \mid \text{ham}).$$

   - Pick whichever is larger. Often "win," "money," and "now" are much more likely under the spam model, so you label it spam.

1. The Math: Bayes' Theorem + "Naïve" Independence

For a datum x with features $(x_1, \ldots, x_n)$ and a set of classes $C = \{c_1, \ldots, c_K\}$, we compute

$$P(c_j \mid x_1, \ldots, x_n) \;\propto\; P(c_j) \prod_{i=1}^{n} P(x_i \mid c_j).$$

- **Prior**: $P(c_j)$ is the fraction of training examples in class c_j.
- **Likelihood**: $P(x_i \mid c_j)$ is the probability of feature i taking value x_i in class c_j.
- **Naïve assumption**: features x_i are independent given the class, so we multiply their probabilities.

At prediction time we pick the class with highest posterior:

$$\hat{c} = \arg\max_{c_j} P(c_j) \prod_{i} P(x_i \mid c_j).$$

To avoid zero probabilities, we typically use **Laplace (add-one) smoothing**:

$$P(x_i = v \mid c_j) = \frac{N_{i,v,c_j} + 1}{N_{c_j} + V_i},$$

where N_{i,v,c_j} = count of training examples in class c_j with feature i equal to v, N_{c_j} = total examples in c_j, and V_i = number of possible values feature i can take.

$$-x\cos(x) + \sin(x) + C$$

This output would be verified symbolically (via differentiation) to check correctness.

🔷 Limitations

- May sometimes produce incorrect results (especially on very complex or pathological functions)
- Requires post-processing or verification (like symbolic differentiation)

🔷 Related Systems

- **GPT-f** (OpenAI): A fine-tuned GPT model for formal mathematics
- **AlphaMath / MathGPT**: Emerging systems that combine symbolic math with large language models

---

- Input: mathematical expression to integrate
- Output: corresponding integral (antiderivative)
- Similar to how translation models work (e.g., English → French)

3. **Symbolic Output**:
   - Returns exact expressions (e.g., ∫sin(x) dx → -cos(x) + C)
   - Unlike numerical methods (like Simpson's Rule), which only approximate
4. **Generalization**:
   - Learns integration patterns and applies them even to unseen forms
   - Particularly effective in areas where traditional symbolic solvers might need extensive rule trees

---

🔷 Example Workflow

For the integral:

$$\int x \cdot \sin(x)\, dx$$

A SAINT model would process the tokenized input (like `x * sin(x)`) and output the antiderivative:

$$-x\cos(x) + \sin(x) + C$$

🔷 What is SAINT in the Context of Solving Integrals?

SAINT (Symbolic AI for INTegration) is an AI-based system designed to **solve symbolic integration problems** — integrals with algebraic, trigonometric, or transcendental functions that require symbolic manipulation rather than numeric evaluation.

It is based on:

- **Transformer-based neural networks** (like those used in language models)
- Trained on datasets of integration problems and solutions
- Performs integration by **learning symbolic patterns**, not by brute-force or numerical methods

---

🔷 Key Features

1. **Trained on Massive Datasets**:
    - Pairs of functions and their integrals
    - Enables learning of rules similar to human-derived methods
2. **Sequence-to-Sequence Architecture**:
    - Input: mathematical expression to integrat

3. Sparse-Memory Graph Search (SMGS)

SMGS is essentially DCFS + a “lazy” CLOSED-list pruner:

- It keeps **all three** sets initially (OPEN, boundary, kernel).
- When memory pressure builds up, it invokes **PruneClosed**:
  1. Scan OPEN and mark all CLOSED nodes that have an OPEN-child → these become **relay** nodes.
  2. For each boundary node, follow its back-pointers to the **most recent** relay on that path, and set an **ancestor** pointer to that relay.
  3. **Delete** every closed node that was *not* marked as a relay. Now CLOSED = kernel + boundary has been pruned to just the relay landmarks.
- Continue A* normally on the now-smaller CLOSED; if you run out of memory again you prune a second “layer” of relay nodes, and so on.

When the goal is reached, you end up with a **sparse solution path** through a handful of relay nodes. You then invoke SMGS recursively on each segment (Start→R_1, R_1→R_2, …, R_k→Goal) to fill in the “dense” path between each pair of relays—just as DCFS does.