Breaking substitution ciphers with Markov models

Algorithms description and implementation notes

Enrico Rollando (e.rollando@outlook.com)

August 6, 2021

Abstract

Cryptography is a useful playground for applying statistical methods. This document uses the simple substitution cipher as a toy problem and shows how to decrypt messages without knowing the key (i.e., without known plaintext). It focuses on Markov model-based approaches, reviews the underlying theory, and provides concrete implementations and usage examples.

Introduction

Can we break a simple substitution cipher automatically? "To break" here means discovering and exploiting weaknesses to restore—at least in part—the original text.

For scope, consider plaintext and ciphertext over uppercase English letters plus space (26+1 symbols). In the base setting, space maps to itself.

A *simple substitution cipher* is a mapping function $S_k(\text{text}) \to \text{ciphertext}$ where k is the key. It is a bijection on the alphabet (equivalently, a permutation of its symbols). Given the ciphertext only, the goal is to recover the inverse mapping $S_k^{-1}(\text{ciphertext})$ without any known plaintext.

To guide the search, define a function *fit(text)* that scores the current guess. Ideally this would answer "how likely is this the original plaintext?", but a practical proxy is: "how plausible is this as English text?". This score lets us compare different candidate decryptions and prefer the most plausible one.

A natural way to build such a score is to use language statistics: how often letters and *n*-grams appear in English, and how likely one character is to follow another. We then score candidate decryptions using these statistics.

There is still a problem: the number of possible keys is huge (26! for the English alphabet), far too many to try exhaustively. Other techniques are needed; this document presents a few of them.

Let us first define the cipher and briefly introduce the language model, which is the main tool used to exploit the cipher's weakness.

The simple substitution cipher

A simple substitution cipher [1] is a one-to-one mapping between two alphabets: the plaintext alphabet and the ciphertext alphabet. This mapping is the key of the cipher.

Spaces are sometimes removed from plaintext before encryption, as they provide information about word boundaries, which in turn helps recovery (the Patristocrat variant).

Related work also treats substitution decipherment as a statistical inference problem, where the plaintext structure and the key are latent and must be inferred jointly [2].

$P = [a_1, ..., a_{26}]$, example: ABCDEFGHIJKLMNOPQRSTUVWXYZ
$C = [c_1, ..., c_{26}]$, example: HQUVNCFYPXJDIMOBWLAZTGSRKE

$$S_k : P \to C \tag{1}$$

$S_k(\text{"THISISATEST"}) = \text{"ZYPAPAHZNAZ"}$
$S_k^{-1}(\text{"ZYPAPAHZNAZ"}) = \text{"THISISATEST"}$

The Language Model

In any language there are implicit rules we use when we write or interpret sentences. For example, the incomplete sentence "I like to write my letters by …" is likely to end with "hand". The preceding words provide context, which makes some continuations more plausible than others.
At the character level, the same idea lets us read an incomplete word like "uncomf.rt.b.y": plausibility can be interpreted statistically as "which character is most likely here, given its context?".

A language model is tied to a specific language and captures such regularities using frequencies at the character or word level. As an example, consider a bigram English language model LM_{EN}:

- bigram frequency:
    - how frequent is a bigram ab in English text? $P_{LM}(ab) = \frac{\text{count(occurrences of } ab)}{\text{len(text)}-1}$
    - for a bigram ab, how frequently does b follow a?

$$P_{LM}(b \mid a) = \frac{\text{count}(ab)}{\text{count}(a)} \tag{2}$$

The weakness

A simple substitution cipher always encrypts the same plaintext symbol in the same way. As a result, it preserves a surprising amount of structure from the plaintext: equal letters remain equal, repeated patterns remain repeated, and, more generally, the plaintext n-gram statistics are preserved **up to a relabeling of symbols** induced by the unknown key.

For example, if a ciphertext bigram is a double letter (e.g., `XX`), then the corresponding plaintext bigram must also be a double letter. By itself this is not enough to identify the letters, but across a whole message these constraints become informative, especially when combined with an English language model.

To exploit this, we can introduce a generative model G for English plaintext, controlled by parameters Θ, and search for a key (or plaintext) that makes the decoded text both (i) consistent with the ciphertext under a substitution mapping and (ii) plausible under G. In practice, this is done iteratively: we refine Θ and/or the key to increase the model score (likelihood or plausibility) of the observed ciphertext under the induced plaintext statistics.

Results at a glance

Method	Cipher length	Iterations/time	Outcome
EM (Baum-Welch)	~120 chars	500 iters (~1-2 min)	Readable plaintext; some letters may need manual swaps
MH (substitution)	~40 chars	1m swaps (~1 min)	Gradual convergence to legible text; prints best candidates
MH (patristocrat)	~30 chars	5k insert/remove (~seconds)	Recovers likely spacing; ambiguity remains on very short texts
LM pickle prep	corpus-dependent	varies	N-gram log-probability tables reused across runs

Reproducibility

Goal & Scope

- Ciphertext alphabet: A-Z + space. Spaces are preserved for the base examples; the patristocrat variant removes them.
- Corpus: plain-text English corpus (e.g., the Brown corpus [3]), preprocessed by uppercasing, keeping only A-Z and spaces, and collapsing whitespace.
- Success criteria: recover readable plaintext and a near-bijective substitution map; report convergence traces instead of only final text.
- Deliverables: pseudocode for EM and Metropolis-Hastings, runnable helpers in `src/crypto`, and a reproducible setup (commands + package versions).

How to reproduce

- Run from `crypto1/` so relative paths resolve (notably `resources/...`).
- Ensure the n-gram LM pickle exists (shipped as `resources/en/ngrams_space.pkl`). To regenerate it:
    - Default (Brown corpus; requires `nltk` + Brown data): `python3 -m src.crypto.demo prep-lm --out resources/en/ngrams_space.pkl`
    - Or from a local corpus file: `python3 -m src.crypto.demo prep-lm --corpus path/to/corpus.txt --out resources/en/ngrams_space.pkl`
- Run the demos:
    - EM: `python3 -m src.crypto.demo em --max-iter 700 --restarts 15 --verbose`
    - MH: `python3 -m src.crypto.demo mh --iters-sub 1000000 --iters-space 5000 --verbose`
    - All: `python3 -m src.crypto.demo all --max-iter 700 --restarts 15 --iters-sub 1000000 --iters-space 5000 --verbose`
- Runtime (typical laptop): EM decryption ~1-2 minutes at 500-700 iterations; MH on short texts ~seconds; pickle generation scales linearly with corpus size.

Reproducibility setup

- Environment: Python 3.x and `numpy`.
- Project layout: run from `crypto1/` and invoke `python3 -m src.crypto.demo ...` (or use `.venv/bin/python3` if you created a virtualenv in this directory).
- Data: the demos load the language model from `resources/en/ngrams_space.pkl` by default; regenerate it with `prep-lm` as shown above if needed.
- Optional: to regenerate the pickle from the Brown corpus, install `nltk` and ensure the Brown corpus data is available locally.
- Determinism: MH uses fixed internal seeds; EM is stochastic due to random initialization, so use multiple restarts and report representative traces.

Methods

Breaking the cipher with Expectation-Maximization (EM / Baum-Welch)

We cast the substitution cipher as a Hidden Markov Model (HMM): hidden plaintext states emit the observed ciphertext symbols, and plaintext-to-plaintext transitions follow an English bigram language model. EM (Baum-Welch) [4] [5] then adjusts the HMM parameters to increase the likelihood of the observed ciphertext. In our setup, we typically learn the transition model (A and π) from an English corpus and, for each ciphertext, re-estimate the emission model (B) while keeping A, π fixed.

Implementation details: scaled forward/backward passes prevent numerical underflow. We also avoid exact zeros in A (e.g., via smoothing) so transitions are never impossible and EM updates do not get stuck.

```
1 # scaled forward/backward to avoid underflow
```

```
 2 initialize A, B, PI
 3 repeat until convergence:
 4     alpha, scale = forward(A, B, PI, V)
 5     beta = backward(A, B, V, scale)
 6     gamma_state = expected_state_visits(alpha, beta)
 7     gamma_trans = expected_state_transitions(alpha, beta, A, B, V)
 8     if recompute_A: A = normalize_rows(gamma_trans)
 9     if recompute_B: B = normalize_rows(state_emissions(gamma_state, V))
10     if recompute_PI: PI = gamma_state[0]
11     B = constrain_space(B)  # enforce space->space
```

This highlights the scaled passes, the non-zero constraints on `A`, and the bijective tweak for the space character.

EM Implementation: the Baum-Welch algorithm Ciphertext symbols are modeled as emissions from a hidden Markov model (HMM). Let $H = (p_1, \ldots, p_N)$ be the hidden state sequence (e.g., plaintext symbols) and $V = (c_1, \ldots, c_N)$ the observed sequence (ciphertext symbols). Parameters are $\Theta = (\pi, A, B)$: * $\pi_i = P(p_1 = i)$ initial state probability. * $A_{ij} = P(p_{t+1} = j | p_t = i)$ transition probability. * $B_i(c) = P(c_t = c | p_t = i)$ emission probability.

Joint probability (one particular hidden path):

$$P(V, H|\Theta) = \pi_{p_1} B_{p_1}(c_1) \prod_{t=1}^{N-1} A_{p_t p_{t+1}} B_{p_{t+1}}(c_{t+1})$$

The likelihood $P(V|\Theta) = \sum_H P(V, H|\Theta)$ sums over all possible hidden paths, which is too expensive to do directly. Forward-backward computes it efficiently.

Forward pass (prefix probability). Define:

$$\alpha_t(i) = P(c_{1:t}, p_t = i|\Theta)$$

Initialize and recurse left to right:

$$\alpha_1(i) = \pi_i B_i(c_1)$$
$$\alpha_t(j) = B_j(c_t) \sum_i \alpha_{t-1}(i) A_{ij} (t = 2, \ldots, N)$$

Backward pass (suffix probability). Define:

$$\beta_t(i) = P(c_{t+1:N}|p_t = i, \Theta)$$

Start at the end and recurse right-to-left:

$$\beta_N(i) = 1$$
$$\beta_t(i) = \sum_j A_{ij} B_j(c_{t+1}) \beta_{t+1}(j) (t = N-1, \ldots, 1)$$

Likelihood (total probability of the observed ciphertext):

$$P(V|\Theta) = \sum_i \alpha_N(i)$$

Posterior marginals (E-step). Once we have α and β, we can ask: "how likely was the model in state i at time t?" and "how likely was the transition $i \to j$ at time t?"

$$\gamma_t(i) = P(p_t = i|V, \Theta) = \frac{\alpha_t(i)\beta_t(i)}{P(V|\Theta)},$$

$$\xi_t(i,j) = P(p_t = i, p_{t+1} = j|V, \Theta) = \frac{\alpha_t(i)A_{ij}B_j(c_{t+1})\beta_{t+1}(j)}{P(V|\Theta)} \quad (t = 1, ..., N - 1)$$

Parameter updates (M-step). Update parameters by normalizing expected counts:

$$\pi_i \leftarrow \gamma_1(i)$$

$$A_{ij} \leftarrow \frac{\sum_{t=1}^{N-1} \xi_t(i,j)}{\sum_{t=1}^{N-1} \gamma_t(i)}$$

$$B_i(c) \leftarrow \frac{\sum_{t=1}^{N} \mathbf{1}[c_t = c]\ \gamma_t(i)}{\sum_{t=1}^{N} \gamma_t(i)}$$

Iterate E/M steps until convergence (or for a fixed number of iterations).

Key implementation details (see [6]): - Transition matrix A avoids exact zeros; strict positivity implies the Markov chain is irreducible and aperiodic. - Forward/backward passes are scaled each step to prevent underflow on long ciphertexts. - First-order LM keeps sparsity manageable; higher orders need smoothing and more data.

EM implementation: a concrete example So far we have summarized the algorithm and its Forward-Backward passes. Let's use it to automatically decrypt some text.

EM usage sketch (see Appendix A for full code)

1. Build bigram LM (A, B_{id}, π)[7]: `A, B_id, PI = build_bigram_model(corpus_text, max_iter=50, limit=200k)`.
2. Initialize emission matrix `B` randomly, apply `constrain_space_mapping(B)`.
3. Run `BaumWelch.process(A, B, PI, V, max_iter=200, recompute_A=False, recompute_PI=False, ext_fun=constrain_space_mapping)`.
4. Decode with `decode_with_emissions(B_hat, ciphertext)` or feed emissions to Viterbi[8] for MAP decoding.

Log-likelihoods from a representative run are in the convergence table below.

EM convergence snapshot

Iteration	$\log P(V|\Theta)$	Note
0	-374.15	random init
2	-297.81	structure starts to appear
16	-242.69	readable plaintext emerges
500	-237.72	plateau; restart to escape local optima

Scaling prevents underflow, and strictly positive `A` keeps the chain irreducible and aperiodic. Run multiple random restarts (different `B` seeds) and keep the run with the best final log-likelihood.

EM decoded sample

- Iter 0: `BZN BQWNXLN JQBZ ...` (random initialization, gibberish)
- Iter 16: `THE TOXUPLE WITH HAZIND AN OQEN MING ...` (largely readable, a few bad mappings)
- Iter 500: `THE TROUPLE WITH HAZING AN OQEN MIND ...` (plateau; minor letter swaps remain)

These snapshots show how likelihood climbs and the mapping improves before stalling. Rerun with new seeds to escape plateaus.

EM usage example: computing statistics for the English bigrams LM Use `build_bigram_model` to estimate the bigram transition matrix A and initial state distribution π from a cleaned English corpus (e.g., the Brown corpus [7]). The helper trims the text to a configurable limit to keep runtimes reasonable.

EM usage: decrypt a text With A and π fixed, initialize an emission matrix `B`, apply `constrain_space_mapping`, and run `BaumWelch.process` with `recompute_A=False` and `recompute_PI=False`. Use `decode_with_emissions` for a quick readout or run Viterbi[8] with the learned emissions to obtain a MAP state sequence.

EM: Review of the Baum-Welch approach Baum-Welch can often recover readable plaintext quickly on sufficiently long ciphertexts, although runtime and quality depend on text length and initialization. In our setup we train A, π from an English corpus (bigram LM), then learn an emission matrix B per ciphertext while keeping A, π fixed.

Notes and limitations:

- the entropy of English implies a (rough) *unicity distance* - the minimum length needed for unique recovery in an idealized setting (Shannon, 1949) [9]. A common estimate uses per-character redundancy $D \approx 3.2$ bits:
$$U = \frac{H(K)}{D} = \frac{\log_2(26!)}{3.2} \approx 28$$

  In practice, Baum-Welch typically needs substantially longer ciphertexts for stable convergence.
- Using only a first-order (bigram) language model limits context to local dependencies.
- Higher-order models (e.g., trigrams) can improve accuracy but increase computation and memory.
- A substitution cipher is a bijection, but plain Baum-Welch does not enforce a one-to-one constraint on B; enforcing bijectivity requires constrained optimization or post-processing.
- We can bias initialization toward expected mappings (e.g., space→space, or other priors in B). Injecting higher-level knowledge (common words, names, greetings) is possible via constraints/priors or hybrid search, but it is not built into standard Baum-Welch.

Breaking the cipher with Metropolis-Hastings

Let's try a different angle.
Is there a way to enforce the bijective property of the cipher while solving the problem?
We briefly describe the Monte Carlo technique and the Metropolis-Hastings algorithm before using it with a new model.

Monte Carlo

If a quantity is intractable or hard to compute directly, we can approximate it via random sampling from a related distribution, avoiding the need for closed-form mathematics.

A typical example is estimating the value of π. Consider a circle of radius 1 inscribed in the square $[-1, 1] \times [-1, 1]$. If we repeatedly generate pseudo-random points (x, y) uniformly in the square and count the fraction that fall inside the circle ($x^2 + y^2 \leq 1$), that fraction estimates the area ratio:

$$\text{square area} = 2r \cdot 2r = 4$$
$$\text{circle area} = \pi \cdot r^2 = \pi$$
$$\frac{\text{circle area}}{\text{square area}} = \pi/4$$

Therefore $\pi \approx 4 \cdot$ (fraction of points inside the circle). By the law of large numbers, this estimate converges as the number of samples grows.

```
1  limit=1_000_000 # the more the better
2  print(sum(random.uniform(-1,1)**2 + random.uniform(-1,1)**2<=1 for i in range(limit)
       )*4.0/limit)
3  3.14188
```

Under "Monte Carlo" methods fall different techniques, which stochastically tweak the parameters of a model to explore unknown values and evaluate its outputs, such that the model can be improved. In our setting, this "tweaking" will take the form of a random-walk proposal over candidate cipher keys. This walk is mostly done by generating random samples and then accepting or discarding them according to a criterion, or by generating samples from some weighted distribution (importance sampling). For example, in the case of a substitution cipher, consider a random walk where at each step two letters of the cipher key are swapped and the resulting plaintext is evaluated for plausibility under a language model.

Metropolis-Hastings, brief introduction

Metropolis-Hastings is a Markov chain Monte Carlo (MCMC) algorithm that works well for the substitution cipher case.

- consider a target distribution $P(x)$ over solutions x (here: cipher keys) which we cannot evaluate exactly, but for which we can compute a score from the observations;
- we have a function $\ell(x) = \log \tilde{P}(x) + C$ which is equal to $\log P(x)$ up to an additive constant (log unnormalized score);
- we make a random walk by randomly making a change to the parameters (e.g., swapping two letters in the key) and then evaluating the result;
- the new value is accepted or rejected using an acceptance function that compares $\tilde{P}(x')$ with $\tilde{P}(x)$ (and, in general, also accounts for the proposal distribution);
- the proposal distribution $q(x'|x)$ specifies how we generate the candidate x' from the current state x (for swap moves, it is typically symmetric).

In the longer term (and under some basic conditions), this walk is guaranteed to sample from the target distribution $P(x)$ (more precisely: $P(x)$ is the stationary distribution of the Markov chain).

More about convergence of Markov models

A discrete-time Markov chain with transition probabilities $P(y|x)$ has a stationary distribution Π^* if

$$\Pi^*(y) = \sum_x \Pi^*(x) P(y|x)$$

In words: if the chain starts distributed as Π^*, then after one step it is still distributed as Π^*.

If the chain is irreducible and aperiodic (and the state space is finite), then a stationary distribution exists, is unique, and the chain converges to it. In particular, if we write the distribution at time t as a row vector Π_t, and the transition matrix as A with entries $A_{xy} = P(y|x)$, then

$$\Pi_{t+1} = \Pi_t A \qquad \Pi_t = \Pi_0 A^t$$

For large t,

$$\Pi_0 A^t \xrightarrow[t \to \infty]{} \Pi^*$$

and the stationary distribution satisfies the fixed-point equation

$$\Pi^* = \Pi^* A$$

Note that convergence does not mean the chain stops moving; it means the distribution of the state stabilizes.

A strong sufficient condition is strict positivity: if $A_{xy} > 0$ for all x, y, then the chain is automatically irreducible and aperiodic, hence it has a unique stationary distribution Π^*.

In Metropolis-Hastings, the goal is to construct a Markov chain whose stationary distribution is the desired target distribution over solution (e.g., cipher keys). In other words, the target distribution is the Π^* we want the chain to converge to.

A common way to guarantee that a chosen Π^* is stationary is detailed balance (time reversibility):

$$\Pi^*(x)P(y|x) = \Pi^*(y)P(x|y) \tag{3}$$

Summing both sides over x gives stationarity:

$$\sum_x \Pi^*(x)P(y|x) = \sum_x \Pi^*(y)P(x|y) = \Pi^*(y) \sum_x P(x|y) = \Pi^*(y)$$

since $\sum_x P(x|y) = 1$.

Some clues behind the inner workings of Metropolis-Hastings for the substitution cipher

So far, we modeled ciphertext with an HMM and learned an emission model with Baum-Welch. However, the substitution cipher key is bijective, and Baum-Welch does not naturally enforce a one-to-one constraint on the emission matrix. Also, using higher-order language models inside an HMM quickly becomes expensive. Let's rethink the model in a way that enforces the substitution mapping directly, and makes it easy to plug in an n-gram LM.

The technique discussed here is inspired by "The Markov Chain Monte Carlo Revolution" (Diaconis, 2009) [10] which gives deeper insights into MCMC and uses Metropolis-Hastings to (among other things) decode an encrypted message.

Consider the substitution mapping S (a bijection) and its inverse S^{-1}: $S(P) = C$ and $S^{-1}(C) = P$.

In Metropolis-Hastings, the state of the Markov chain is the key S itself. We move from one key to another by proposing small changes (e.g., swapping two letters in the key) and accepting/rejecting them. The chain is constructed so that its stationary distribution $\Pi(S)$ assigns higher probability to keys that produce more plausible English plaintext.

The stationary distribution $\Pi(S)$ does not need to be known in normalized form: it is enough to define an unnormalized score $\tilde{\Pi}(S) \propto \Pi(S)$, because MH only uses ratios of scores.

Using an n-gram language model (LM), we can score a candidate key S by decoding the ciphertext sequence $(c_1, \dots, c_N)$ into candidate plaintext symbols

$$p_i = S^{-1}(c_i)$$

and then computing the LM score of the resulting plaintext under the n-gram factorization:

$$Pl(S) = \sum_{i=n}^{N} \log P_{LM}(p_i \mid p_{i-n+1:i-1})$$

In implementation we use an unnormalized weighted sum of n-gram log-weights; MH only needs score differences.

Metropolis-Hastings (symmetric proposal): the algorithm

Metropolis et al. (1953) [11] is the special case of Metropolis-Hastings where the proposal is symmetric; it can be motivated via detailed balance.

It needs (i) a symmetric proposal distribution to generate candidate moves and (ii) a plausibility score (based on the language model) proportional to the target distribution. It works as follows:

0. Start with:

- a current state x (e.g., a key; can be random),
- a symmetric proposal $g(y|x)$ such that $g(y|x) = g(x|y)$,
- a log-score $Pl(x)$ (plausibility), defined so that $\exp(Pl(x)) \propto \Pi(x)$ (equivalently, $Pl(x) = \log \tilde{\Pi}(x) + C$ for some constant C).

1. From the current state x, draw a candidate $y \sim g(y|x)$.
2. Compute the acceptance probability

$$\alpha(x, y) = \min(1, \exp(Pl(y) - Pl(x)))$$

3. Accept/Reject:

- draw $u \sim \mathrm{Uniform}(0, 1)$;
- if $u \leq \alpha(x, y)$, accept and set $x \leftarrow y$;
- otherwise, reject and keep x.

4. Repeat from step 1 (for a fixed number of iterations, or after burn-in collect samples).

The acceptance rule comes from the detailed-balance condition (3). Let Π^* be the desired stationary distribution and let $T(y|x)$ be the actual transition kernel of the Metropolis chain. Condition (3) requires:

$$\Pi^*(x)T(y|x) = \Pi^*(y)\, T(x|y)$$

For $y \neq x$, the Metropolis transition can be written as:

$$T(y|x) = g(y|x)\alpha(x, y)$$

where $g(y|x)$ is the proposal distribution and $\alpha(x, y)$ is the acceptance probability. Plugging this into (3) gives:

$$\frac{g(y|x)\alpha(x, y)}{g(x|y)\alpha(y, x)} = \frac{\Pi^*(y)}{\Pi^*(x)}$$

If the proposal is symmetric, $g(y|x) = g(x|y)$, this simplifies to:

$$\frac{\alpha(x, y)}{\alpha(y, x)} = \frac{\Pi^*(y)}{\Pi^*(x)}$$

Since $Pl(x)$ is a log-score (i.e., $Pl(x) = \log \tilde{\Pi}(x)$ up to an additive constant), the condition is satisfied by choosing:

$$\alpha(x, y) = \min(1, \exp(Pl(y) - Pl(x))) = \min\left(1, \frac{\Pi^*(y)}{\Pi^*(x)}\right)$$

```
1 ct = normalize(ciphertext)
2 S = random_key(alphabet)                  # current bijection (state)
3 pt = decode(ct, S)                         # candidate plaintext under S
4 score = plausibility(pt, lm)               # log-score (recommended)
5
6 best_S, best_score = S, score
7
8 for _ in range(N):
9     a, b = pick_two_symbols(alphabet)
10    S2 = swap_in_key(S, a, b)               # symmetric proposal, still bijective
11    pt2 = decode(ct, S2)
12    score2 = plausibility(pt2, lm)Δ
13
```

```
14          = score2 - score                        # log acceptance ratio
15      if log(rand_uniform()) <= Δ:                # accept with prob min(1, expΔ())
16          S, pt, score = S2, pt2, score2
17
18      if score > best_score:
19          best_S, best_score = S, score
```

The proposal enforces a bijection by swapping two symbols; the plausibility function can mix multiple n-gram lengths.

Acceptance intuition: early on, many swaps are accepted; later, acceptance drops as the chain reaches higher-scoring regions. Restart from a fresh seed if trapped in low-quality states.

Plausibility should return a log score (sum of log n-gram probs) so Δ is stable and $\exp(\Delta)$ is well-defined.

Metropolis-Hastings implementation: a concrete example

With this algorithm, we can exploit an LM with multiple n-grams. This is straightforward because the plausibility function only needs to rank candidates; a fully normalized probability is not required. Intuitively, a candidate plaintext should score higher when it contains n-grams that are typical for English. A simple choice is the sum of weighted n-gram frequencies, where the weight increases with n because longer contexts carry more information.

Since "frequency" is a normalized quantity, the normalization step can be omitted as long as the normalization factor (e.g., the total number of extracted n-grams) is constant across candidates; it cancels out when comparing two states in the acceptance ratio. In practice, the weighted count is treated as a log-plausibility score, so the acceptance step uses $\exp(\Delta)$ with $\Delta = Pl(y) - Pl(x)$.

Furthermore, at each step an existing mapping is swapped; this maintains the bijection between the plaintext and ciphertext alphabets.

The n-gram model is sparse for larger n, so it would be inefficient to store it as a dense array. In this Python example we use a dictionary to store only the observed n-grams.

Finally, the acceptance rule allows occasional moves to lower-scoring states, which helps the chain jump between local optima. There is no single point where the algorithm is guaranteed to be "done", so for this project we set a maximum number of iterations; another option is to stop if there is no improvement in the best score for a fixed window of iterations.

Here is my full implementation of the Metropolis algorithm and a concrete example:

MH usage sketch (see Appendix B for full code)

- Build a multi-gram LM: `lm = count_ngrams(clean_corpus, min_len=2, max_len=5)`.
- Substitution cipher: `metropolis_substitution(ciphertext, lm, iterations=50k, seed=7)` tracks and prints intermediate best candidates during the run.
- Patristocrat spacing: `metropolis_patristocrat(text_without_spaces, lm, iterations=5k, seed=11)` proposes insert/remove-space moves and tracks the best spacing found.

MH trace highlights

- In this trace, the first accepted swaps quickly lock in a few high-signal bigrams (e.g., `TH`, `HE`), and readability starts improving early.
- The acceptance rate here tracks the plausibility spread: once the score separation between candidates grows, most proposed swaps are rejected and progress comes from occasional improving moves.
- In the spacing stage of this same trace, patristocrat spacing recovers as common trigrams reappear.

MH trace sample

- It#1: `K CXJO XI HBD BKIO XS ENJHB HEN XI HBD CLSB` (initial guess)
- It#15: `M CFJS FI TBR BMIS FO ENJTB TEN FI TBR CLOB` (bigrams emerging)
- It#219: `IF YOU LOOK FOR PERFECTION YOU LL NEVER BE CONTENT` (spacing recovered)

Patristocrats

To make a substitution cipher harder to break, spaces can be removed from the ciphertext. The American Cryptogram Association calls this kind of cryptogram a "Patristocrat". This increases difficulty, at the cost of added ambiguity when reconstructing word boundaries.

To solve a Patristocrat, it is convenient to score candidate plaintext without spaces using an LM trained on a corpus with spaces removed (alphabet A-Z only). The same algorithms as before can be used, but longer ciphertext is typically needed to provide enough statistical signal. Higher-order n-gram language models (as used in the Metropolis-Hastings scoring function) tend to work much better here than first-order/bigram models, as expected.

After the letters have been decoded, spaces can be reintroduced using an LM trained on a corpus with spaces. There are several ways to do this (e.g., dynamic programming), but with only minor changes to the previous implementation we use Metropolis-Hastings: proposals insert/remove a space at random positions and moves are accepted according to the spaced-LM plausibility score.

Practical usage: train the A-Z LM (spaces removed) and run `metropolis_patristocrat`; then train the spaced LM and run the spacing sampler.

Review of the Metropolis approach

With enough iterations and a few random restarts, Metropolis-Hastings can often recover readable plaintext even from relatively short ciphertexts, although success depends on text length and the strength of the language-model score. The algorithm is flexible: it can incorporate higher-order n-gram language models directly in the plausibility function without changing the sampler itself. In practice, the speed of improvement depends mainly on the proposal function (how new candidates are generated) and the acceptance rule (which controls how often worse moves are tolerated). Variants of Metropolis-Hastings typically modify these two components to trade off exploration, runtime, and solution quality.

Limitations & future work

- Short ciphertexts remain ambiguous; try multiple random seeds/restarts or incorporate partial known-plaintext cribs.
- Higher-order n-gram LMs can improve plausibility, but they require smoothing (to avoid zeros) and larger corpora to reduce sparsity.
- Other languages and alphabets (diacritics, punctuation, different whitespace rules) require retraining the LM and adjusting the symbol mapping/preprocessing.
- EM can stall in local optima; multiple restarts can help, and annealed/tempered variants of EM may improve robustness.

Appendices

Full Python listings referenced in the text. Place these in `src/crypto/` or on your import path when re-running the experiments.

Appendix A: EM helper code

```python
"""Baum-Welch helpers for bigram HMMs, including training from text and decoding utilities."""

from __future__ import annotations
```

```python
4
5  from typing import Callable, Tuple
6
7  import numpy as np
8
9  from .utils import (
10     ALPHABET,
11     SPACE_IDX,
12     char_to_index,
13     encode_text,
14     index_to_char,
15     normalize_text,
16     seed_everything,
17 )
18
19 EmissionHook = Callable[[np.ndarray], np.ndarray]
20
21
22 class BaumWelch:
23     """Minimal Baum-Welch implementation with scaling to avoid underflow."""
24
25     def __init__(self, num_hidden: int = len(ALPHABET)):
26         self.num_hidden = num_hidden
27         self.num_visible = num_hidden
28         self.T = 0
29
30     @staticmethod
31     def normalize(matrix: np.ndarray) -> Tuple[np.ndarray, np.ndarray]:
32         totals = np.sum(matrix, axis=min(1, len(matrix.shape) - 1), keepdims=True)
33         totals[totals == 0] = 1.0
34         return matrix / totals, totals
35
36     def alpha(self, A: np.ndarray, B: np.ndarray, PI: np.ndarray, V: np.ndarray):
37         alphas = np.zeros((self.T, self.num_hidden))
38         scale_facts = np.zeros(self.T)
39         alphas[0], total = self.normalize(PI * B[:, V[0]])
40         scale_facts[0] = total.item()
41         for t in range(1, self.T):
42             alphas[t], total = self.normalize(
43                 B[:, V[t]] * np.dot(alphas[t - 1], A)
44             )
45             scale_facts[t] = total.item()
46         return alphas, scale_facts
47
48     def beta(self, A: np.ndarray, B: np.ndarray, V: np.ndarray, scale_facts: np.ndarray):
49         betas = np.zeros((self.T, self.num_hidden))
50         betas[self.T - 1] = 1.0
51         for t in range(self.T - 2, -1, -1):
52             betas[t] = np.sum(betas[t + 1] * B[:, V[t + 1]] * A, axis=1) / scale_facts[t + 1]
53         return betas
54
55     def process(
56         self,
57         A: np.ndarray,
```

```python
        B: np.ndarray,
        PI: np.ndarray,
        V: np.ndarray,
        max_iter: int,
        recompute_A: bool = True,
        recompute_B: bool = True,
        recompute_PI: bool = True,
        ext_fun: EmissionHook | None = None,
        verbose_fun: Callable[[np.ndarray, np.ndarray, int, float], None] | None = None,
        stop_on_decrease: bool = True,
        min_delta: float = 0.0,
        patience: int | None = None,
    ):
        self.T = V.size
        log_scale_facts = None
        stale = 0
        for it in range(max_iter):
            alphas, scale_facts = self.alpha(A, B, PI, V)
            new_log_scale = float(np.sum(np.log(scale_facts)))
            if log_scale_facts is not None:
                delta = new_log_scale - log_scale_facts
                if stop_on_decrease and delta < -min_delta:
                    if verbose_fun:
                        print("Algorithm has converged (decrease)")
                    break
                if abs(delta) <= min_delta:
                    stale += 1
                    if patience is not None and stale >= patience:
                        if verbose_fun:
                            print("Algorithm has converged (plateau)")
                        break
                else:
                    stale = 0

            log_scale_facts = new_log_scale
            betas = self.beta(A, B, V, scale_facts)
            gammas = np.zeros((self.T, self.num_hidden, self.num_hidden))
            for t in range(self.T - 1):
                gammas[t] = (A.T * alphas[t, :]).T * (betas[t + 1] * B[:, V[t + 1]])
                denom = float(np.sum(gammas[t]))
                if denom != 0.0:
                    gammas[t] /= denom
            gammas_i = np.sum(gammas, axis=2)
            gammas_i[self.T - 1] = alphas[self.T - 1, :]

            if recompute_A:
                gammas_t = np.sum(gammas, axis=0)
                totals = np.sum(gammas_t, axis=1)[:, None]
                totals[totals == 0] = 1.0
                A = (gammas_t.T / totals).T

            if recompute_B:
                B_counts = np.zeros_like(B)
                for v in range(self.num_visible):
```

```python
112                    mask = V == v
113                    if np.any(mask):
114                        B_counts[:, v] = np.sum(gammas_i[mask], axis=0)
115                totals = np.sum(gammas_i, axis=0)
116                totals[totals == 0] = 1.0
117                B = (B_counts.T / totals).T

119            if recompute_PI:
120                PI = gammas_i[0]

122            if ext_fun is not None:
123                B = ext_fun(B)

125            if verbose_fun:
126                verbose_fun(V, B, it, log_scale_facts)

128        return A, B, PI, log_scale_facts


131 def constrain_space_mapping(B: np.ndarray) -> np.ndarray:
132     """
133     Force a 1-1 mapping for space:space in the emission matrix.
134
135     Ensures space can only map to space, keeping the substitution mapping bijective for that
            symbol.
136     """
137     B = B.copy()
138     B[SPACE_IDX, :] = 0.0
139     B[:, SPACE_IDX] = 0.0
140     B[SPACE_IDX, SPACE_IDX] = 1.0
141     row_sums = B.sum(axis=1, keepdims=True)
142     row_sums[row_sums == 0] = 1.0
143     return B / row_sums


146 def build_bigram_model(
147     corpus_text: str,
148     *,
149     max_iter: int = 150,
150     limit: int = 1_000_000,
151     seed: int = 13,
152 ):
153     """
154     Train a bigram language model from plain text using Baum-Welch.
155
156     Returns the transition matrix A, an identity emission matrix B, and the
157     initial state distribution PI. The text is upper-cased, cleaned, and
158     truncated to `limit` characters to keep runtimes reasonable.
159     """
160     seed_everything(seed)
161     normalized = normalize_text(corpus_text)[:limit]
162     V = np.array(encode_text(normalized))
163
164     num_states = len(ALPHABET)
```

```
165     PI = np.random.rand(num_states)
166     A = np.random.rand(num_states, num_states)
167     B = np.identity(num_states)
168
169     baum = BaumWelch(num_states)
170     A, _, PI, _ = baum.process(A, B, PI, V, max_iter=max_iter, recompute_B=False)
171     return A, B, PI
172
173
174 def decode_with_emissions(B: np.ndarray, observations: str) -> str:
175     """
176     Simple decoder: pick the highest emission probability per symbol.
177
178     For maximum a posteriori decoding, use the Viterbi algorithm instead.
179     """
180     V = [char_to_index(ch) for ch in observations]
181     decoded_indices = [int(np.argmax(B[:, v])) for v in V]
182     return "".join(index_to_char(idx) for idx in decoded_indices)
```

Appendix B: Metropolis-Hastings helper code

```
1  """Metropolis(-Hastings) helpers for substitution and spacing recovery."""
2
3  from __future__ import annotations
4
5  import math
6  import random
7  import string
8  from collections import Counter
9  from pathlib import Path
10 from typing import Dict
11
12 import pickle
13
14
15 def count_ngrams(text: str, min_dgram_len: int = 2, max_dgram_len: int = 9) -> Dict[int,
    Dict[str, int]]:
16     """Raw n-gram counts over the requested length range."""
17     return {
18         dlen: Counter(text[idx : idx + dlen] for idx in range(len(text) - dlen + 1))
19         for dlen in range(min_dgram_len, max_dgram_len + 1)
20     }
21
22
23 def load_pickled_language_model(path: Path) -> Dict[int, Dict[str, float]]:
24     """Load a precomputed n-gram language model from a pickle."""
25     data = pickle.loads(path.read_bytes())
26     return {int(k): v for k, v in data.items()}
27
28
29 def build_language_model(keep_spaces: bool = True, pickle_path: Path | None = None) ->
    Dict[int, Dict[str, float]]:
30     """
31     Create the multi-gram language model.
32
```

```python
33        Requires `pickle_path` pointing to an n-gram LM pickle.
34        The pickle is expected to contain a mapping `dict[int, dict[str, float]]` of n-gram
              length to log-weights.
35        """
36        if pickle_path and pickle_path.exists():
37            return load_pickled_language_model(pickle_path)
38        raise RuntimeError("Language model pickle not found.")
39
40
41  def plausibility(
42        text: str,
43        lm: Dict[int, Dict[str, float]],
44        *,
45        missing: float = math.log(1e-6),
46  ) -> float:
47        """
48        Log-plausibility under the LM: sum of count * log-weight, weighted by n-gram length.
49        """
50        min_n, max_n = min(lm), max(lm)
51        cnt = count_ngrams(text, min_n, max_n)
52        total = 0.0
53        for ngram_len, counts in cnt.items():
54            lm_row = lm.get(ngram_len, {})
55            for k, freq in counts.items():
56                weight = lm_row.get(k, missing)
57                total += freq * weight * ngram_len
58        return total
59
60
61  def metropolis_substitution(
62        ciphertext: str,
63        iterations: int = int(1e6),
64        seed: int = 13,
65        lm_pickle: Path | None = None,
66        verbose: bool = False,
67  ):
68        """
69        Recover simple substitution using Metropolis on swap proposals.
70        """
71        random.seed(seed)
72        mm = build_language_model(keep_spaces=True, pickle_path=lm_pickle)
73        best = pl_f = plausibility(ciphertext, mm)
74        best_text = ciphertext
75        fixed = set(ciphertext) - set(string.ascii_uppercase)
76        used_symbols = list(set(ciphertext) - fixed)
77        all_symbols = set(string.ascii_uppercase)
78        smallprob = math.log(1e-3)
79        for i in range(iterations):
80            c1 = random.choice(used_symbols)
81            c2 = random.choice([c for c in all_symbols if c != c1])
82            candidate = ciphertext.translate(str.maketrans({c1: c2, c2: c1}))
83            pl_f_star = plausibility(candidate, mm)
84            mu = math.log(random.uniform(0, 1))
85            if mu < max(min(pl_f_star - pl_f, 0), smallprob):
```

```python
86              if pl_f_star > best:
87                  best = pl_f_star
88                  best_text = candidate
89                  if verbose:
90                      print("It#", i, candidate)
91          ciphertext = candidate
92          pl_f = pl_f_star
93          used_symbols = list(set(ciphertext) - fixed)
94      return best_text


97  def metropolis_patristocrat(
98      ciphertext: str,
99      iterations: int = 5000,
100     seed: int = 21,
101     lm_pickle: Path | None = None,
102     verbose: bool = False,
103 ):
104     """
105     Restore spaces in a patristocrat ciphertext.
106     """
107     random.seed(seed)
108     mm = build_language_model(keep_spaces=True, pickle_path=lm_pickle)
109     best = pl_f = plausibility(ciphertext, mm) / len(ciphertext)
110     ret = ciphertext
111     for i in range(iterations):
112         c = random.choice(range(1, len(ciphertext) - 2))
113         candidate = list(ciphertext)
114         if ciphertext[c] == " ":
115             candidate = candidate[:c] + candidate[c + 1 :]
116             pl_f_star = plausibility("".join(candidate), mm) / len(candidate)
117         elif ciphertext[c - 1] != " " and ciphertext[c + 1] != " ":
118             candidate[c:c] = " "
119             pl_f_star = plausibility("".join(candidate), mm) / len(candidate)
120         else:
121             pl_f_star = pl_f
122         mu = math.log(random.uniform(0, 1))
123         if mu < min(pl_f_star - pl_f, 0):
124             ciphertext = "".join(candidate)
125             pl_f = pl_f_star
126             if best < pl_f_star:
127                 best = pl_f_star
128                 ret = ciphertext
129                 if verbose:
130                     print(ciphertext)
131     return ret, best
```

Appendix C: Utility functions

```python
1 import math
2 import pickle
3 import random
4 import re
5 from collections import Counter
6 from pathlib import Path
```

```python
 7 from typing import Iterable
 8
 9 ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
10 SPACE_IDX = ALPHABET.index(" ")
11
12
13 def seed_everything(seed: int) -> None:
14     """Seed Python's random generator and numpy if available."""
15     random.seed(seed)
16     try:
17         import numpy as np
18
19         np.random.seed(seed)
20     except Exception:
21         # numpy is optional
22         pass
23
24
25 def normalize_text(text: str) -> str:
26     """
27     Keep A-Z and spaces, collapse whitespace, and uppercase.
28
29     Ensures corpus and cipher text share the same character mapping for the LM.
30     """
31     cleaned = re.sub("[^A-Z ]", " ", text.upper())
32     return re.sub("\\s+", " ", cleaned).strip()
33
34
35 def char_to_index(ch: str) -> int:
36     """Map a character in ALPHABET to its ordinal index."""
37     return ALPHABET.index(ch)
38
39
40 def index_to_char(idx: int) -> str:
41     """Map an ordinal index back to a character in ALPHABET."""
42     return ALPHABET[idx]
43
44
45 def encode_text(text: Iterable[str]) -> list[int]:
46     """Convert a string of characters into indices."""
47     return [char_to_index(ch) for ch in text]
48
49
50 def decode_indices(indices: Iterable[int]) -> str:
51     """Convert indices back to a string of characters."""
52     return "".join(index_to_char(i) for i in indices)
53
54
55 def write_ngram_lm_pickle(
56     corpus_text: str,
57     out_path: str | Path,
58     *,
59     min_n: int = 1,
60     max_n: int = 7,
```

```
61      keep_spaces: bool = True,
62      limit_chars: int | None = None,
63  ) -> Path:
64      """
65      Build an n-gram language model from text and write it as a pickle.
66
67      The corpus is normalized to the same A-Z plus space alphabet as the cipher text.
68      The pickle contains `dict[int, dict[str, float]]`, mapping n-gram length to
69      n-gram log-probabilities (natural log) estimated by relative frequency (no smoothing).
70      Unseen n-grams are not stored and are handled by the scoring function via a fallback.
71      """
72      if min_n < 1 or max_n < min_n:
73          raise ValueError("Expected 1 <= min_n <= max_n")
74
75      normalized = normalize_text(corpus_text)
76      if not keep_spaces:
77          normalized = normalized.replace(" ", "")
78      if limit_chars is not None:
79          if limit_chars < 0:
80              raise ValueError("Expected limit_chars >= 0")
81          if limit_chars:
82              normalized = normalized[:limit_chars]
83
84      lm: dict[int, dict[str, float]] = {}
85      for n in range(min_n, max_n + 1):
86          counts = Counter(normalized[idx : idx + n] for idx in range(len(normalized) - n + 1))
87          total = sum(counts.values())
88          if total == 0:
89              lm[n] = {}
90              continue
91          lm[n] = {ng: math.log(cnt / total) for ng, cnt in counts.items()}
92
93      out_path = Path(out_path)
94      out_path.parent.mkdir(parents=True, exist_ok=True)
95      out_path.write_bytes(pickle.dumps(lm, protocol=pickle.HIGHEST_PROTOCOL))
96      return out_path
97
98
99  def load_brown_corpus_text(*, keep_spaces: bool = True) -> str:
100     """
101     Load the Brown corpus as a single string.
102
103     Requires `nltk` and the locally-installed Brown corpus data.
104     """
105     try:
106         from nltk.corpus import brown  # type: ignore[import-not-found]
107     except ModuleNotFoundError as exc:
108         raise RuntimeError("NLTK is required to load the Brown corpus; install `nltk` or pass
                a corpus file.") from exc
109
110     try:
111         sentences = brown.sents()
112     except LookupError as exc:
113         raise RuntimeError("NLTK Brown corpus data not found; install it or pass a corpus
```

```
              file.") from exc
114
115     sep = " " if keep_spaces else ""
116     return sep.join(" ".join(sent) for sent in sentences)
```

Appendix D: Package exports

```
1  """Python package exporting the EM and MH helpers."""
2
3  from .em import (
4      BaumWelch,
5      build_bigram_model,
6      constrain_space_mapping,
7      decode_with_emissions,
8  )
9  from .mh import (
10     build_language_model,
11     count_ngrams,
12     metropolis_patristocrat,
13     metropolis_substitution,
14 )
15 from .utils import ALPHABET, SPACE_IDX, char_to_index, decode_indices, encode_text,
       index_to_char, normalize_text, seed_everything
16
17 __all__ = [
18     "ALPHABET",
19     "SPACE_IDX",
20     "char_to_index",
21     "index_to_char",
22     "encode_text",
23     "decode_indices",
24     "normalize_text",
25     "seed_everything",
26     "BaumWelch",
27     "build_bigram_model",
28     "constrain_space_mapping",
29     "decode_with_emissions",
30     "build_language_model",
31     "count_ngrams",
32     "metropolis_substitution",
33     "metropolis_patristocrat",
34 ]
```

Appendix E: Demo entrypoint

```
1  """Command-line entrypoint for the Baum-Welch and Metropolis demos."""
2
3  from __future__ import annotations
4
5  import argparse
6  import math
7  from pathlib import Path
8
9  import numpy as np
10
```

```python
11  from .em import BaumWelch, constrain_space_mapping, decode_with_emissions
12  from .mh import load_pickled_language_model, metropolis_patristocrat, metropolis_substitution
13  from .utils import ALPHABET, encode_text, load_brown_corpus_text, write_ngram_lm_pickle
14
15
16  def run_em_demo(max_iter: int = 500, verbose: bool = False, restarts: int = 15) -> None:
17      """Run the Baum-Welch experiment using a precomputed bigram LM loaded from
                `resources/en/ngrams_space.pkl`."""
18      num_states = 27
19      lm_path = Path("resources/en/ngrams_space.pkl")
20      lm_stats = load_pickled_language_model(lm_path)
21
22      def _softmax_from_logs(logs: np.ndarray) -> np.ndarray:
23          m = float(np.max(logs))
24          shifted = logs - m
25          w = np.exp(shifted)
26          s = float(np.sum(w))
27          if s == 0.0 or not np.isfinite(s):
28              return np.full_like(w, 1.0 / len(w))
29          return w / s
30
31      def _build_bigram_language_model_from_stats():
32          symbols = list(ALPHABET)
33          log_eps = math.log(1e-6)
34          log_pi = np.array([float(lm_stats[1].get(ch, log_eps)) for ch in symbols],
                  dtype=float)
35          PI_local = _softmax_from_logs(log_pi)
36          A_local = np.zeros((len(symbols), len(symbols)), dtype=float)
37          for i, a in enumerate(symbols):
38              row_logs = np.array([float(lm_stats[2].get(a + b, log_eps)) for b in symbols],
                      dtype=float)
39              A_local[i, :] = _softmax_from_logs(row_logs)
40          return A_local, PI_local
41
42      A, PI = _build_bigram_language_model_from_stats()
43
44      # Use Baum-Welch to estimate emissions for this ciphertext.
45      encrypted = (
46          "RBO RPKTIGO VCRB BWUCJA WJ KLOJ HCJD KM SKTPQO CQ RBWR "
47          "LOKLGO VCGG CJQCQR KJ SKHCJA WGKJA WJD RPYCJA RK LTR RBCJAQ CJ CR "
48      )
49      V = np.array(encode_text(encrypted))
50      baum = BaumWelch()
51
52      log_eps = math.log(1e-6)
53      bigram = lm_stats[2]
54
55      def score_bigram(text: str) -> float:
56          # sum bigram log-weights; higher is better
57          total = 0.0
58          for i in range(len(text) - 1):
59              bg = text[i : i + 2]
60              total += bigram.get(bg, log_eps)
61          return total
```

```python
    best_overall = {"text": "", "log": -1e300, "score": -1e300}

    for r in range(restarts):
        # Start from a random emission matrix; space forced later.
        B = np.random.rand(num_states, num_states)

        best_candidate = {"text": "", "log": -1e300, "score": -1e300, "B": None}

        def verbose_fun(V_, B_, it, log_norm):
            decoded = decode_with_emissions(B_, encrypted)
            sc = score_bigram(decoded)
            if verbose:
                print(f"[r{r}] #It", it, decoded, log_norm, f"(bg_score={sc:.2f})")
            if sc > best_candidate["score"]:
                best_candidate["score"] = sc
                best_candidate["text"] = decoded
                best_candidate["log"] = log_norm
                best_candidate["B"] = B_.copy()
            if log_norm > best_candidate["log"]:
                best_candidate["log"] = log_norm

        _, B_hat, _, log_norm = baum.process(
            A,
            B,
            PI,
            V,
            max_iter=max_iter,
            ext_fun=constrain_space_mapping,
            verbose_fun=verbose_fun,
            recompute_A=False,
            recompute_PI=False,
            stop_on_decrease=False,
        )
        winner_decoded = best_candidate["text"] or decode_with_emissions(B_hat, encrypted)
        winner_log = best_candidate["log"] if best_candidate["text"] else log_norm
        winner_score = best_candidate["score"] if best_candidate["text"] else \
            score_bigram(winner_decoded)
        # Prefer the emissions matrix at the best LM score if captured
        if best_candidate["B"] is not None and best_candidate["text"]:
            winner_decoded = decode_with_emissions(best_candidate["B"], encrypted)
        if winner_score > best_overall["score"]:
            best_overall = {"text": winner_decoded, "log": winner_log, "score": winner_score}

    print(best_overall["text"])
    print("log_norm:", best_overall["log"])


def run_mh_substitution_demo(iterations: int = 1_000_000, *, verbose: bool = False) -> None:
    """Metropolis substitution demo."""
    print("Metropolis substitution:")
    best = metropolis_substitution(
        " K CXJO XI HBD BKIO XL ENJHB HEN XI HBD CSLB ",
        iterations=iterations,
```

```python
115          lm_pickle=Path("resources/en/ngrams_space.pkl"),
116          verbose=verbose,
117      )
118      print("\nBest candidate:")
119      print(best)
120
121
122  def run_mh_spacing_demo(iterations: int = 5000, *, verbose: bool = False) -> None:
123      """Metropolis spacing demo."""
124      print("\nMetropolis spacing:")
125      spaced, _ = metropolis_patristocrat(
126          " IFYOULOOKFORPERFECTIONYOULLNEVERBECONTENT ",
127          iterations=iterations,
128          lm_pickle=Path("resources/en/ngrams_space.pkl"),
129          verbose=verbose,
130      )
131      print("\nBest spacing:")
132      print(spaced)
133
134
135  def run_prepare_lm_pickle(
136      *,
137      out_path: Path,
138      corpus_path: Path | None,
139      min_n: int,
140      max_n: int,
141      keep_spaces: bool,
142      limit_chars: int | None,
143  ) -> None:
144      """Generate an n-gram language model pickle used by the demos."""
145      if corpus_path is None:
146          corpus_text = load_brown_corpus_text(keep_spaces=keep_spaces)
147      else:
148          corpus_text = corpus_path.read_text(encoding="utf-8")
149      out = write_ngram_lm_pickle(
150          corpus_text,
151          out_path,
152          min_n=min_n,
153          max_n=max_n,
154          keep_spaces=keep_spaces,
155          limit_chars=limit_chars,
156      )
157      print(f"Wrote: {out}")
158
159
160  def main() -> int:
161      parser = argparse.ArgumentParser(description="Run the EM and MH demos from the examples.")
162      sub = parser.add_subparsers(dest="cmd")
163
164      p_em = sub.add_parser("em", help="Run Baum-Welch demo")
165      p_em.add_argument("--max-iter", type=int, default=700)
166      p_em.add_argument("--verbose", action="store_true")
167      p_em.add_argument("--restarts", type=int, default=15)
168
```

```
169    p_mh = sub.add_parser("mh", help="Run Metropolis demos")
170    p_mh.add_argument("--verbose", action="store_true")
171    p_mh.add_argument("--iters-sub", type=int, default=1_000_000)
172    p_mh.add_argument("--iters-space", type=int, default=5000)
173
174    p_all = sub.add_parser("all", help="Run both demos")
175    p_all.add_argument("--max-iter", type=int, default=700)
176    p_all.add_argument("--verbose", action="store_true")
177    p_all.add_argument("--iters-sub", type=int, default=1_000_000)
178    p_all.add_argument("--iters-space", type=int, default=5000)
179    p_all.add_argument("--restarts", type=int, default=15)
180
181    p_prep = sub.add_parser("prep-lm", help="Generate the n-gram LM pickle (default: Brown
           corpus)")
182    p_prep.add_argument("--out", type=Path, default=Path("resources/en/ngrams_space.pkl"))
183    p_prep.add_argument("--corpus", type=Path, default=None, help="Path to a plain-text
           corpus file (UTF-8)")
184    p_prep.add_argument("--min-n", type=int, default=1)
185    p_prep.add_argument("--max-n", type=int, default=7)
186    p_prep.add_argument("--no-spaces", dest="keep_spaces", action="store_false", help="Remove
           spaces from the corpus")
187    p_prep.add_argument("--limit-chars", type=int, default=0, help="Optionally truncate the
           normalized corpus")
188    p_prep.set_defaults(keep_spaces=True)
189
190    args = parser.parse_args()
191    if args.cmd is None:
192        parser.print_help()
193        return 0
194
195    if args.cmd in {"em", "all"}:
196        run_em_demo(max_iter=args.max_iter, verbose=args.verbose, restarts=args.restarts)
197    if args.cmd in {"mh", "all"}:
198        run_mh_substitution_demo(iterations=args.iters_sub, verbose=args.verbose)
199        run_mh_spacing_demo(iterations=args.iters_space, verbose=args.verbose)
200    if args.cmd == "prep-lm":
201        limit = None if args.limit_chars == 0 else args.limit_chars
202        run_prepare_lm_pickle(
203            out_path=args.out,
204            corpus_path=args.corpus,
205            min_n=args.min_n,
206            max_n=args.max_n,
207            keep_spaces=args.keep_spaces,
208            limit_chars=limit,
209        )
210    return 0
211
212
213 if __name__ == "__main__":
214    raise SystemExit(main())
```

References

[1] Substitution cipher. (2021, July 27). In Wikipedia.

https://en.wikipedia.org/w/index.php?title=Substitution_cipher&oldid=1035776944

[2] Dempster, A.P.; Laird, N.M.; Rubin, D.B. (1977). "Maximum Likelihood from Incomplete Data via the EM Algorithm". Journal of the Royal Statistical Society, Series B. 39 (1): 1-38. JSTOR 2984875. MR 0501537.

[3] Maya R. Gupta and Yihua Chen (2011), "Theory and Use of the EM Algorithm", Foundations and Trends® in Signal Processing: Vol. 4: No. 3, pp 223-296. https://dx.doi.org/10.1561/2000000034

[4] Knight, Kevin & Nair, Anish & Rathod, Nishit & Yamada, Kenji. (2006). Unsupervised Analysis for Decipherment Problems. https://dx.doi.org/10.3115/1273073.1273138

[5] Gagniuc, Paul A. (2017). Markov Chains: From Theory to Implementation and Experimentation. USA, NJ: John Wiley & Sons. pp. 1-256. ISBN 978-1-119-38755-8

[6] L. R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," in Proceedings of the IEEE, vol. 77, no. 2, pp. 257-286, Feb. 1989. https://dx.doi.org/10.1109/5.18626

[7] Francis, W. Nelson & Henry Kucera. 1967. Computational Analysis of Present-Day American English. Providence, RI: Brown University Press.

[8] A. Viterbi, "Error bounds for convolutional codes and an asymptotically optimum decoding algorithm," in IEEE Transactions on Information Theory, vol. 13, no. 2, pp. 260-269, April 1967, doi: 10.1109/TIT.1967.1054010.

[9] Shannon, Claude. "Communication Theory of Secrecy Systems", Bell System Technical Journal, vol. 28(4), page 656-715, 1949. https://doi.org/10.1002/j.1538-7305.1949.tb00928.x

[10] Diaconis, Persi. (2009). The Markov Chain Monte Carlo Revolution. Bulletin of the American Mathematical Society. 46. 179-205. https://dx.doi.org/10.1090/S0273-0979-08-01238-X

[11] Metropolis, N., Rosenbluth, A. W., Rosenbluth, M. N., Teller, A. H., & Teller, E. (1953). Equation of State Calculations by Fast Computing Machines. The Journal of Chemical Physics, 21(6), 1087–1092. https://doi.org/10.1063/1.1699114