Breaking substitution ciphers with Markov models

Algorithms description and implementation notes

Author: Enrico Rollando (e.rollando@outlook.com)

Date: August 6, 2021

Abstract: Cryptography can be an interesting field to practice the knowledge of statistical methods. In this document I use the simple substitution cipher as a toy example to decrypt messages without knowing the key. I focus on unsupervised learning techniques based on Markov models, review the theory behind them, and show concrete implementations and usage examples. Full code listings are provided in the appendices; the main text uses pseudocode and algorithmic descriptions.

Goal & Scope

- Ciphertext alphabet: uppercase A–Z plus space. Spaces are preserved for the base examples; the patristocrat variant removes them.
- Language model: use any cleaned English corpus (e.g., Brown) to generate n-gram stats/pickles before running the demos.
- 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 reproducible setup (seed + package versions).

How to reproduce

- Inputs: cleaned English corpus (e.g., Brown), ciphertext samples in A–Z + space, patristocrat sample without spaces.
- Runtime (typical laptop): EM bigram LM ~2–4 minutes at 200k chars if you regenerate stats; EM decryption ~1–2 minutes at 200 iterations; MH 50k swaps on short texts ~seconds; patristocrat spacing ~seconds.
- Seed: fix RNG seeds (e.g., 13/42) for comparable traces; exact ciphertext-dependent results still vary.

Reproducibility setup

- Environment: Python 3.x, `numpy`. Install `nltk` only if you choose to download Brown to generate your n-grams; otherwise supply any cleaned corpus.
- Seeding: fix RNG seeds for repeatable EM/MH traces.
- Corpus: upper-case, strip non-letters (except space), collapse whitespace.

Corpus note: generate your own n-gram stats before running the examples; see the next section for a minimal recipe.

Generating n-gram pickles

To run the examples, first create your own n-gram statistics from a cleaned English corpus:

1. Clean the corpus: uppercase, keep A–Z and spaces, collapse whitespace.
2. Count n-grams (e.g., unigrams/bigrams) over the cleaned text.
3. Save the log-count dictionaries to pickles: place `ngrams.pkl` (no spaces) and `ngrams_space.pkl` (with spaces) under `resources/en/` or adjust the paths in the code.

Any corpus is acceptable (e.g., Brown corpus[1]). NLTK is only needed if you choose Brown.

Introduction

Can we break a simple substitution cipher automatically? "To break" here means: discover and use some weaknesses to restore - at least in part - the original text.
To define the scope, let's look only at plaintext and ciphertext composed in English capital letters and space, 26+1 symbols in total. Let's further say that the space is encrypted as space.

The *simple substitution cipher* is a mapping function $S_k(text) \rightarrow ciphertext$ where k is the key. It's a bijection between the alphabet sequence A and one of its |A|! permutations. Given the ciphertext only, I want to figure out the inverse mapping $S^{-1}(ciphertext)$ in an unsupervised way.

Let's try to define a hypothetical function: *fit(text)* that evaluates the current guess, ideally as "what is the probability that this text is the original plaintext?". A slightly different question that leads to a similar result and which I *can* actually define is: "how likely is it that this text is (English) plaintext?". I can use this function to evaluate and compare different guesses and choose the best one.

To model this evaluation I could use for example the unique features of a specific language. How often does a certain letter, or a certain group of letters (n-grams), appear in the plaintext. Or: how often a certain letter happens to follow another. And then look at the ciphertext and measure the distance using these statistics.

There is still a problem, the number of possible keys is huge (26! for the English alphabet), too many to try them all in a reasonable amount of time. Other techniques have to be used, in this document I show some of them.

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

Let us first look at the cipher to break and briefly present the Language Model which is the fundamental feature used to exploit the weakness of the cipher.

The simple substitution cipher

A simple substitution cipher [2] is a set of functions each modeling a one-to-one mapping between two alphabets: the plaintext- 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 recovering the plaintext.

P=$[a_1, ..., a_{26}]$, example: ABCDEFGHIJKLMNOPQRSTUVWXYZ
C=$[c_1, ..., c_{26}]$, example: HQUVNCFYPXJDIMOBWLAZTGSRKE
S: $f(P) \rightarrow C$ (1)
S(“THISISATEST”)=“ZYPAPAHZNAZ”
S^{-1}(“ZYPAPAHZNAZ”)=“THISISATEST”

The Language Model

In any language there are implicit (not formalized) rules that we use when we make or interpret a sentence. For example, an incomplete sentence “I like to write my letters by …” would be expected to end with the word “hand”. The initial words give a specific *context* and a context gives some words more *plausibility* than others.
At the character level, some knowledge of English is enough to read an incomplete word like “uncomf.rt.b.y”. The plausibility can be interpreted statistically: what is the most likely character in this position, given the visible sequence of characters?

The Language Model is tied to a specific language and focuses on frequencies of elements, on character or word level. As an example I consider the bigram English Language Model LM_{EN}, informally:

- bigram frequency:
  - how frequent is a bigram ab in English text? $P_{LM}(ab) = \frac{count(\text{occourrences of ab})}{len(\text{text})-1}$
  - For a bigram ab how frequently does b follow a in English text? $P_{LM}(b|a) = \frac{count(ab)}{count(a)}$ (2)

The weakness

The simple substitution cipher encrypts the same symbol each time the same way. The ciphertext positionally maintains statistical information of the original plaintext. In other words, the i-th n-gram of both plaintext and ciphertext will have the same frequency in its message. For example, if the second bigram

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

is a double-letter, the most probable plaintext bigram is “TT” or “LL”.
To exploit this fact I could for example define a generative distribution G for the plaintext, controlled by some parameters Θ, then generate candidate plaintext, such that its positional n-grams distribution probability matches the one of the ciphertext.
This process could be progressively refined by automatically tweaking the parameters Θ to let distribution G converge to the observed distribution of the ciphertext.

Break the cipher

Let’s review some theory first, which allows me to introduce an important amount of tools.

Break the cipher with the Expectation-Maximization (EM) algorithm

The main problem, at first sight, is the huge amount of keys to be tested to find a solution. This is a common problem for optimization algorithms and there are different approaches to solve it.
A well-known algorithm is the Expectation-Maximization (Dempster, Laird, Rubin, 1977) [3].
A very good description of the algorithm can be found in “Theory and Use of the EM Algorithm” (Maya R. Gupta and Yihua Chen, 2011) [4].
Some interesting research has been done in the last years about using EM in substitution ciphers, for a nice example read: “Unsupervised Analysis for Decipherment Problems” (Knight, Kevin & Nair, Anish & Rathod, Nishit & Yamada, Kenji, 2006) [5].
Expectation-maximization is, like *dynamic programming*, an optimization technique used to reduce the complexity of an algorithm by breaking it down into smaller and more tractable steps which are repeated until convergency is reached.
EM identifies a family of algorithms, in this document I use a specialized implementation of the EM for the discrete case.

EM basis: statistical foundation

[3] 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.

[4] 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

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

Markov model A Markov model (Gagniuc, Paul A., 2017) [6] can be used for both discrete and continuous stochastic processes satisfying the Markov property: the previous state of the system is sufficient to describe its current state. This simplification allows for a series of techniques, such as Monte Carlo, which goal is to find out the parameters that stabilize the model without trying all the possible combinations.

An example of Markov model is the Markov chain. A Markov chain is a model that defines a conditional probability of moving to state x_i when the last k states were: $x_{i-k}...x_{i-1}$ and k is the level of "visibility" on the previous states. For example, an LM can be approximated by a Markov chain of order n by using n-gram conditional probabilities (2).

The transition probability from the last sequence $p_{i-k}...p_{i-1}$ to the next letter p_i will be:

$$P(p_i|p_{i-k},...,p_{i-1})$$

The transition probability must define a valid probability distribution such that:

$$\sum_{p_i} P(p_i|p_{i-k},...,p_{i-1}) = 1$$

The Markov chain defines a process which states change over time, based on the last k-states and the transition matrix.

A Markov chain of order 1, in the discrete case, is defined by a set of states $(x_0,...x_n)$ and by a transition matrix:

$$A : p_{ij} = P(X_t = j|X_{t-1} = i)$$

which describes the probabilities to move from previous state i to current state j. This simplification minimizes the knownledge of the model by stating that the new transition only depends on the previous k states.

When not all the states of the process are observable, but the observations are a probabilistic function of the states, we talk about Hidden Markov models. This technique was defined by Leonard E. Baum and colleagues in the 1960's, a very good reference can be found in "A tutorial on hidden Markov models and selected applications in speech recognition" (L. R. Rabiner, 1989) [7].

We could model a substitution cipher with a HMM where the plaintext is the unknown state sequence, because it is hidden from the observation, and only the ciphertext is visible. The LM will help in evaluating the plausibility of the guessed plaintext as English text.

[6]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

[7]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.

EM algorithm

The EM algorithm is a way to reduce the complexity of the problem of finding the best parameters of a model, maximizing the probability that the model would produce the observed output.

EM: A brief description There exist different forms of the algorithm, but a general explanation is the following:
The algorithm is used to improve the (possibly unknown or incomplete) parameters Θ of a given model by processing an unknown, not observable value X so that the output would maximally match some observed data Y.
It does this by repeating the following steps until the improvement is negligible (=the algorithm converged):

- E-step:
  - Using the current parameter values Θ, estimate X
- M-step:
  - using the estimated X, improve (find a better estimate for) Θ such that $P(Y|\Theta)$ grows.

For the cipher, X is the plaintext and Y is the ciphertext.

I'll reinstate the problem of solving the cipher, this time using the HMM.

EM: HMM model for the simple substitution cipher We have:

- a hidden sequence of plaintext characters $H : p_1, ..., p_n$
- an observed sequence of ciphertext characters $V : c_1...c_n$

In our case they use the same vocabulary of English letters.
The hidden sequence is drawn from the English LM statistics: it is English text and it is expected to respect the rules of the English language.
The observed sequence is bound to the hidden sequence by an as yet unknown transformation.
The goal is to find out the most likely sequence of plaintext characters that originated the ciphertext:

$$H = argmax_H P(H|V) = argmax_H (V, H)$$

from Bayes' theorem and simplifying, as the denominator has no part in the argmax.

The HMM is defined by:

- a fixed set of N hidden states X = {set of possible letters in plaintext}
- a fixed set of M observable states Z = {set of letters in the ciphertext}
- the probability Π of starting with a certain hidden state p_i
- the probability of moving from state p_i to p_j, kept in the so-called transition matrix A (size NxN).

- the probability of observing c_i being in state p_i, kept in the so-called emission matrix B (size NxM).

The hidden states are the unknown plaintext character sequence, the visible states are the observed ciphertext character sequence.
The parameters of the model are $\Theta = (\Pi, A, B)$.
These are the initialization steps:

- initialize Π from the LM with the probability of starting a sentence with letter i.
- initialize A with the probability of switching from letter i to letter j, taken from the bigram LM: $P(p_i|p_{i-1})$. This matrix will be kept constant during the optimization process as it's the middle used to exploit the cipher weakness.
- randomly initialize B which is the parameter to optimize. It represents the hidden distribution P(V|H) which "explains" how the plaintext characters get encrypted. When considering the space-plaintext symbol, set a one-to-one corrispondence with the space-output symbol giving it a probability of 1 in the matrix.

EM: the optimization process The EM algorithm receives as input the initial probability state distribution Π, the transition matrix A, the emission matrix B and the observations, then it tweaks the parameters to reach the maximum likelihood of the hidden variable H by alternating two steps:

1. E-step: using the evidence (the observed data) and the current parameters Θ infer the hidden values (those that the system will consider the most probable, given the current knowledge at this timestep).
2. M-step: using inferred hidden values re-estimate the parameters.
3. repeat from 1 until convergence - or until the desired maximum number of iterations is reached.

An important note is that the algorithm can converge to a local optimum. To improve the chance of finding the global optimum, the algorithm should be executed multiple times and the results compared.

The EM algorithm for a HMM can be implemented in different ways. Here I make use of a specialized implementation for discrete HMM: the Baum-Welch algorithm (from the inventors Leonard E. Baum and Lloyd R. Welch, 1960s), which in turn uses the forward-backward algorithm.

```
# scaled forward/backward to avoid underflow
initialize A, B, PI
repeat until convergence:
    alpha, scale = forward(A, B, PI, V)
    beta = backward(A, B, V, scale)
    gamma_state = expected_state_visits(alpha, beta)
    gamma_trans = expected_state_transitions(alpha, beta, A, B, V)
```

```python
    if recompute_A: A = normalize_rows(gamma_trans)
    if recompute_B: B = normalize_rows(state_emissions(gamma_state,
       V))
    if recompute_PI: PI = gamma_state[0]
    B = constrain_space(B)  # keep →spacespace bijective
```

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 Consider the hidden state H: $p_1 ... p_n$ which we just initially guess - by choosing B - and the observation V: $c_1 ... c_n$

- Each transition $p_{t-1} \rightarrow p_i$ happens with probability $P(p_t|p_{t-1})$. The c_i character is emitted with probability is $P(c_t|p_t)$.
- The probability of the visible sequence V given a particular hidden state sequence is the initial probability of starting with p_1, $\Pi(p_1)$ and then emitting c_1: $P(c_1|p_1)$, times the transition probabilities and the emission probabilities at each timestep t:
  $P(c_{1..N}, p_{1...N}|\Theta) = \Pi(p_1)P(c_1|p_1)\prod_t P(p_t|p_{t-1})P(c_t|p_t)$.
  Notice that this can be computed recursively.
- The probability of the visible sequence V is the marginalization of the previous computation for every possible hidden sequence $H = p_{1...N}$: $P(V|\Theta) = \sum_H \Pi(p_1)\prod_i P(p_t|p_{t-1})P(c_t|p_t)$. This is a computational challenge.
- Baum-Welch easies the computation by splitting it in two parts for each timestep, based on *what has already been observed* (forward pass) and *what it will be observed* (backward pass).
- The **forward pass** of the Forward-Backward algorithm reduces the complexity by computing $P(H_t = p_t, V_{1...t}|\Theta)$ recursively and then marginalizing the result. At each step t for each hidden state i and visible output $V_t = c_t$ the value *alpha* is defined as follows:
  $\alpha_t[i] = B[V_t]\sum_j \alpha_{t-1}(j)A[ij]$
- The **backward pass** will complete this partial result: where the forward pass computes the probability up to step t, the backward pass does the same for the subsequent timesteps, going backwards from step t to step $t+1$ and computing $P(V_{(t+1)...N}|p_t, \Theta)$. This pass can be again computed recursively in a similar way. For this computation the value *beta* is introduced:
  $\beta_t[i] = \sum_j \beta_{t+1}[j]B[j, V_{t+1}]A[ij]$
- The probability of moving from state p_t to p_{t+1} is proportional to $\alpha_t[i]A[ij]B[j, V_{t+1}]\beta_{t+1}(j)$. Splitting this computation with alpha and beta is the core optimization that avoids the algorithm being exponential.
- The conditional probability of the hidden states H is: $P(H|V, \Theta) =$

$\frac{P(H,V|\Theta)}{P(V|\Theta)}$. The partial results of the forward and backward passes can be later reused to calculate this probability.

- Finally, Π, A and B can be updated by marginalizing $P(H|V,\Theta)$ and normalizing.

There are a couple of important details which I used in my implementation, (see: [8]: "Implementation issues for HMMs"): - The elements of the transition matrix A should never be zero. - The forward and backward algorithms are recursively computed products of probabilities. As the ciphertext length grows, the computation converges exponentially to 0, which generates underflows that I avoid by scaling each step. - I keep the Markov order at 1 to limit sparsity; higher-order LMs sharpen plausibility but require smoothing and much more data.

EM implementation: a concrete example So far I 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 (optional): `A, B_id, PI = build_bigram_model(corpus_text, max_iter=50, limit=200k)` if you want to retrain.
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 for MAP decoding.

For the demos, `A` and `PI` can also be reconstructed from your saved unigram/bigram pickles instead of retraining from Brown.

EM convergence snapshot

Iteration	log P(V	θ)
0	-374.15	random init
2	-297.81	structure starts to appear
16	-242.69	readable plaintext emerges
950	-237.72	plateau; rerun with new seeds to escape local optima

Scaling prevents underflow, and zero-free `A` keeps the chain ergodic. Repeat with a few random `B` seeds and keep the run with the best log-likelihood.

[8] 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.

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 PI from a cleaned English corpus if you wish to regenerate stats. After counting unigrams/bigrams, save them to pickles (e.g., `resources/en/ngrams_space.pkl`) for reuse in the examples.

EM usage: decrypt a text With `A` and `PI` 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 plug the emissions into Viterbi[9] for a MAP sequence.

EM: Review of the Baum-Welch approach With the Baum-Welch algorithm I can automatically decrypt a text in seconds.
The algorithm is very flexible as any of its parameters A, B, Θ can be optimized. I used the algorithm to prepare the English bigram LM and used it again to decrypt some text.

let's make some considerations:

- the entropy of an English text tells us how much information is given us by a text. This can be used to compute the *unicity distance*: (Claude Shannon, "Communication Theory of Secrecy Systems", 1949) [10]: the minimum length of text which gives us enough information to decrypt, which is defined as the entropy of the keyspace divided by the per-character redundancy in bits (ca. 3.2 for English text):

  $$U = \frac{H(K)}{D} = \frac{log2(26!)}{3.2} \approx 28$$

  For the Baum-Welch approach to work, much longer text is needed.
- this technique only exploits the bigram LM, or first-order HMM, which gives to the statistics a fair restricted horizon visibility.
- it's possible to improve the results by modifying the algorithm and make use of higher-order HMMs. Various solutions can be further used to improve speed and memory usage, but the algorithm will be slower and memory intensive when using higher-order HMMs.
- the substitution cipher realizes a bijectional mapping, but I could not directly enforce nor exploit this knowledge in the algorithm.
- it's possible to give a hint to the process: like for the space-to-space mapping, the A matrix (randomly filled in the example) can hold higher probabilities for known or expected mappings. However, there is no immediate way to tell the process that there are high expectations for specific words of part of sentences, like greetings or names.

[9] 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.

[10] 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

Results at a glance

Method	Cipher length	Iterations/time	Outcome
EM (Baum–Welch)	~120 chars	200 iters (~1–2 min)	Readable plaintext; some letters may need manual swap
MH (substitution)	~40 chars	50k swaps (~seconds)	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
Bigram LM prep	200k chars	~2–4 min	Transition matrix A, start dist PI reused across runs

Break the cipher with Metropolis-Hastings

Let's try out other solutions for the problem.
Is there a way to enforce the bijectional property of the cipher while solving the problem?
I briefly describe the Monte Carlo technique and the Metropolis-Hastings algorithm before using it with a new model.

Monte Carlo

When we have a distribution which we can estimate, or from which can be sampled, if we want to find related quantities which for some reasons are intractable or not easy to compute, we can spare the calculations and just approximate these quantities by random sampling.

As a typical example, we could estimate the value of π by just drawing a circle of radius 1 inscribed in a 2x2 square and then repeatedly generate a pseudo-random point within the square area (coordinates $x, y \in [1, 1]$) and measure the proportion of times the point falls within the circle area.

Knowing the relation between the area of a square and the area of the inscribed circle:

$$\text{square area} = 2r * 2r = 4$$

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

We reach an estimation of the expected value of $\pi/4$ and, most importantly, for the law of large numbers this value will converge to the expected value $\pi/4$ for large n.

```python
limit=1000000 # the more the better
print(sum([random.uniform(-1,1)**2 + random.uniform(-1,1)**2<=1 for
    i in range(limit) ])*4.0/limit)
3.14188
```

Under the “Monte Carlo” methods fall different techniques, which stochastically tweak the parameters of the model to explore unknown values and evaluate its outputs, such that the model can be improved. In a HMM this finetuning is called “random walk”.
This walk is mostly done by generating random samples and discarding not plausible ones (rejection sampling) 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 sampled to evaluate its plausibility.

Metropolis-Hastings, brief introduction

Metropolis-Hastings is a Monte Carlo rejection sampling algorithm that works well for the substitution cipher case.

- consider an unknown distribution P(x) which we however can estimate using only the observations.
- we have a function Pl(x) which is just proportional to the density of P(x).
- we make a random walk to progressively improve the unknown function by randomly make a change on the parameters of the model and then sample from the results.
- the new value will be accepted or refused using the “acceptance function” which compares Pl(x) with its previous value to decide.

In the longer term (and under some basic conditions) this walk is guaranteed to optimize the system and return samples which follow the unknown distribution P(x).

More about convergency of Markov models

A (irreducible, aperiodic) discrete Markov model with states distribution $\Pi(x)$ and transition distribution $P(y|x)$ will converge when:

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

That is, the probability of being in y is equal to the probability of getting into y from any state x.
Π^* is the stationary distribution here (it may not exist or may not be unique).
Starting with the initial state distribution Π and transitioning n times using the transition matrix A, the model will enter a new state: $\Pi * A^n$.

For high values of n, Π becomes insignificant and the result will converge:

$$\Pi^* = \lim_{n\to\infty} A^n \Rightarrow \Pi^* * A = \Pi^*$$

That is, if the probability Π has not changed after the transformation, the chain has converged (the random walk has ended, the system has not changed state).

A theorem says that if $A_{xy} > 0 \quad \forall x, y \longrightarrow \exists$ unique Π^* (=if the transition matrix contains no zeroes, a unique stationary distribution exists).

The stationary distribution Π^* is equivalent to the unknown P(x) and it's the desired result of the optimization process.
A useful way to see that the stationary distribution exists is when the system is time-reversible, meaning (*detailed balance equation*):

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

In fact, summing for x at both sides:

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

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

So far, the HMM has represented the probability of each individual letter in plain text to be transformed and observed. The transformation was not bijective and the first-order LM was a limitation.
Let's rethink the model in a way that enforces the substitution mapping and simplifies the use of higher-order LM.

The technique discussed here comes from the very interesting "The Markov Chain Monte Carlo Revolution" (Diaconis, Persi, 2009) [11] which provides deeper insights into MCMC and uses Metropolis-Hastings to, among other things, reveal the contents of an encrypted message from a prison inmate.

Consider (1) the substitution cipher mapping functions $S : f(P) = C$ and the inverse mapping $S^{-1} : f^{-1}(C) = P$ and its correspondence to the distribution

[11] Diaconis, Persi. (2009). The Markov Chain Monte Carlo Revolution. Bulletin of the American Mathematical Society. 46. 179textendash205. https://dx.doi.org/10.1090/S0273-0979-08-01238-X.

functions P(x|y) and P(y|x) where X is the space of the possible symbols and $x, y \in X$
Then the stationary distribution Π would represent the probability that the mapping function is the correct one, as applying successive transformations will not change nor improve the model and the model will output a sample of this stationary distribution, which is the mapping of the cipher.

The Π distribution is initially unknown, but we can easily obtain a probability distribution which is proportional to this distribution.
In fact, using the statistics from the n-gram LM, to estimate the plausibility of a sentence as English text we can compute the joint probability P_{LM} of the n-grams in the text sequence $\{c_1, ..., c_n\}$ transformed into candidate plaintext by the substitution S^{-1}:
(Plausibility:) $Pl = \prod_{i=1}^{n} P_{LM}(S^{-1}(c_{i+1})|S^{-1}(c_i))$

Metropolis-Hastings (symmetric proposal): the algorithm

The original Metropolis algorithm used the “symmetric proposal”, which derives directly from the detailed balance equation.

It needs a symmetric function to choose a new candidate and another function to tune the results (some fitness function which in turn uses information from the LM) and it works as follows:

0. start with
    - some current state x (may be randomly generated)
    - an symmetrical distribution: g(x|y)=g(y|x)
    - a function Pl(x) (the plausibility), proportional to the desired target $\Pi(x)$
1. from current x draw a new state y from the distribution g(y|x)
2. calculate the acceptance rate $\alpha(y, x) = min(1, \frac{Pl(y)}{Pl(x)})$
3. Accept/Reject the candidate:
    - accept the candidate and move to y if $\alpha >= \mu(0, 1)$ (not lower than a random uniform between 0 and 1)
    - refuse otherwise and stay in x
4. repeat from 1 until convergency

The acceptance ratio comes from (3):

$$\frac{P(y|x)}{P(x|y)} = \frac{\Pi(y)}{\Pi(x)}$$

using g to generate samples and alpha to accept or refuse the swap, rewrite the equation as:

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

because g is symmetric this simplifies to:

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

and because $Pl(x)$ is by requirement proportional to $\Pi(x)$ the condition can be fullfilled by choosing alpha to be:

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

```
cipher = normalize(ciphertext)
score = plausibility(cipher, lm)
for _ in range(N):
    c1, c2 = pick_two_letters(cipher)
    candidate = swap(cipher, c1, c2)Δ
     = score(candidate) - score
    accept with prob min(1, expΔ())
    track best candidate for display
```

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

Acceptance intuition: early iterations accept many swaps; as the score climbs, rejections dominate. Restart from a fresh seed if trapped in low-quality states.

Metropolis-Hastings implementation: a concrete example

With this algorithm, I am exploiting an LM with multiple n-grams. This is now easy due to the mild requirements for the Plausibility function. I don't even have to calculate a probability here. Consider that for each n-gram in the candidate text, the frequency of this n-gram in the plaintext will be higher if the n-gram has higher probabilities for that language. The sum of weighted n-gram frequencies (where weight is proportional to n-gram length as statistics on longer n-grams give us better information) is sufficient.
Actually, "Frequency" is a normalized quantity, I don't even need to normalize here as long as the normalization factor (the sum of n-gram counts) stays the same throughout the process, because at each timestep the normalization factor cancels out when the plausibilities Pl_t/Pl_{t+1} are compared.

Furthermore, at each step an existing mapping will be swapped, this maintains the bijection between Plaintext and Ciphertext alphabet.

The n-gram matrix representing the LM is sparse for larger n-gram lengths and it would be inefficient to simply store it in an array. In this python example I am using a *dictionary*.

Finally, the algorithm's convergence technique will allow it to jump from one local minimum to another. For this reason, there is no explicit way to determine whether it has completed. For my fairly simple purpose of breaking the cipher, I just set a maximum number of iterations; another possibility is to stop the algorithm if there is no improvement for a number 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)` prints improved candidates as it accepts better swaps.
- Patristocrat spacing: `metropolis_patristocrat(text_without_spaces, lm, iterations=5k, seed=11)` proposes insert/remove-space moves until the plausibility stabilizes.

Both samplers print intermediate best states so convergence can be inspected in the PDF output.

MH trace highlights

- Early iterations expose high-signal bigrams (e.g., `TH`, `HE`) and quickly improve readability.
- Acceptance rate is controlled by the plausibility spread; widening the n-gram range speeds convergence on short texts.
- Patristocrat spacing recovers steadily as high-frequency trigrams reappear.

Patristocrats

To make the cipher more difficult to break, at the expense of possible ambiguities in the interpretation of the plaintext, all spaces may be removed from the text. The American Cryptogram Association calls this kind of substitution code "Patristocrat".
To solve a Patristocrat, we need an LM calculated from a corpus *without* spaces. The same algorithms as before can be used, but longer ciphertext is needed to provide sufficient statistical input.
The higher-order HMM implementations, such as the one used for Metropolis-Hastings, give here much better results than the first-order implementations, as expected.

After the plaintext has been restored, it is also possible to restore the spaces - this time using an LM from a corpus *with* spaces. There are of course several ways to achieve the goal, such as dynamic programming, but - with only minor changes from the previous implementation - here I use Metropolis for this:

For patristocrats (no spaces), build the language model without spaces and run `metropolis_patristocrat`. Spacing recovery leans on high-frequency trigrams and common prefixes/suffixes; ambiguity increases on very short texts.

Review of the Metropolis approach

Given enough time, very short text can be successfully decrypted with Metropolis.
The algorithm is quite flexible and allows the use of LM from higher order Markov models with ease.
The convergence rate is determined by the function that selects a new candidate and acceptance function (which determines the rejection rate), modified versions of these functions form the basis for alternate versions of this algorithm.

Limitations & future work

- Short ciphertexts remain ambiguous; try multiple seeds or add crib hints.
- Higher-order LMs could improve plausibility but need smoothing and larger corpora.
- Non-English alphabets/spaces require retraining LMs and adjusting symbol maps.
- EM can stall in local optima; multiple restarts or tempered EM could help.

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 routines tailored to simple substitution ciphers.

Written to keep the main text focused on the algorithms while
    preserving a
complete runnable reference implementation.
"""

from __future__ import annotations

from typing import Callable, Tuple

```

```python
import numpy as np

from .utils import (
    ALPHABET,
    SPACE_IDX,
    char_to_index,
    encode_text,
    normalize_text,
    seed_everything,
)

EmissionHook = Callable[[np.ndarray], np.ndarray]


class BaumWelch:
    """Minimal Baum-Welch implementation with scaling to avoid
        underflow."""

    def __init__(self, num_hidden: int = len(ALPHABET)):
        self.num_hidden = num_hidden
        self.num_visible = num_hidden
        self.T = 0

    @staticmethod
    def normalize(matrix: np.ndarray) -> Tuple[np.ndarray,
        np.ndarray]:
        totals = np.sum(matrix, axis=min(1, len(matrix.shape) - 1),
            keepdims=True)
        totals[totals == 0] = 1.0
        return matrix / totals, totals

    def alpha(self, A: np.ndarray, B: np.ndarray, PI: np.ndarray, V:
        np.ndarray):
        alphas = np.zeros((self.T, self.num_hidden))
        scale_facts = np.zeros(self.T)
        alphas[0], scale_facts[0] = self.normalize(PI * B[:, V[0]])
        for t in range(1, self.T):
            alphas[t], scale_facts[t] = self.normalize(
                B[:, V[t]] * np.sum(alphas[t - 1] * A.T, axis=1)
            )
        return alphas, scale_facts

    def beta(self, A: np.ndarray, B: np.ndarray, V: np.ndarray,
        scale_facts: np.ndarray):
        betas = np.zeros((self.T, self.num_hidden))
        betas[self.T - 1] = np.array([1.0 / scale_facts[self.T - 1]])
```

```
        for t in range(self.T - 2, -1, -1):
            betas[t] = (
                np.sum(betas[t + 1] * B[:, V[t + 1]] * A, axis=1) /
                    scale_facts[t]
            )
        return betas

    def process(
        self,
        A: np.ndarray,
        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,
    ):
        self.T = V.size
        log_scale_facts = None
        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 and new_log_scale <=
                log_scale_facts:
                if verbose_fun:
                    print("Algorithm has converged")
                break
            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]])
            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
```

```python

            if recompute_B:
                for v in range(self.num_visible):
                    mask = V == v
                    if np.any(mask):
                        B[:, v] = np.sum(gammas_i[mask], axis=0)
                totals = np.sum(gammas_i, axis=0)
                totals[totals == 0] = 1.0
                B = (B.T / totals).T

            if recompute_PI:
                PI = gammas_i[0]

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

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

        return A, B, PI, log_scale_facts


def constrain_space_mapping(B: np.ndarray) -> np.ndarray:
    """
    Force a 1-1 mapping for space:space in the emission matrix.

    This mirrors the tweak used in the narrative and makes the substitution
    mapping bijective for the space character.
    """
    B = B.copy()
    B[SPACE_IDX, :] = 0.0
    B[:, SPACE_IDX] = 0.0
    B[SPACE_IDX, SPACE_IDX] = 1.0
    row_sums = B.sum(axis=1, keepdims=True)
    row_sums[row_sums == 0] = 1.0
    return B / row_sums


def build_bigram_model(
    corpus_text: str,
    *,
    max_iter: int = 150,
    limit: int = 1_000_000,
    seed: int = 13,
):
```

```
    """
    Train a bigram language model from plain text using Baum-Welch.

    Returns the transition matrix A, an identity emission matrix B, and the
    initial state distribution PI. The text is upper-cased, cleaned, and
    truncated to `limit` characters to keep runtimes reasonable.
    """
    seed_everything(seed)
    normalized = normalize_text(corpus_text)[:limit]
    V = np.array(encode_text(normalized))

    num_states = len(ALPHABET)
    PI = np.random.rand(num_states)
    A = np.random.rand(num_states, num_states)
    B = np.identity(num_states)

    baum = BaumWelch(num_states)
    A, _, PI, _ = baum.process(A, B, PI, V, max_iter=max_iter, recompute_B=False)
    return A, B, PI


def decode_with_emissions(B: np.ndarray, observations: str) -> str:
    """
    Simple decoder: pick the highest emission probability per symbol.

    This is adequate for illustrating the EM approach; for full MAP decoding,
    the Viterbi algorithm[^8] should be used.
    """
    V = [char_to_index(ch) for ch in observations]
    decoded_indices = [int(np.argmax(B[:, v])) for v in V]
    return "".join(ALPHABET[idx] for idx in decoded_indices)
```

Appendix B: Metropolis-Hastings helper code

```
"""
Metropolis-Hastings helpers for substitution ciphers.

The functions mirror the narrative while keeping the main document concise and
focused on results.
"""

```

```python
from __future__ import annotations

import math
import random
import string
from collections import Counter
from typing import Dict, Iterable

from .utils import normalize_text


def count_ngrams(text: str, min_len: int = 2, max_len: int = 9) ->
    Dict[int, Dict[str, float]]:
    """Count n-grams of varying length and return log counts for
        stability."""
    return {
        n: {k: math.log(v) for k, v in Counter(text[idx : idx + n]
            for idx in range(len(text) - n + 1)).items()}
        for n in range(min_len, max_len + 1)
    }


def plausibility_score(text: str, lm: Dict[int, Dict[str, float]])
    -> float:
    """Score text against an LM, weighted by n-gram length."""
    counts = count_ngrams(text, min(lm), max(lm))
    return sum(
        lm[n].get(k, 0.0) * n
        for n in counts
        for k, _ in counts[n].items()
        if k in lm[n]
    )


def metropolis_substitution(
    ciphertext: str,
    lm: Dict[int, Dict[str, float]],
    *,
    iterations: int = 1_000_000,
    seed: int = 13,
) -> str:
    """
    Symmetric-proposal Metropolis-Hastings for simple substitution
        ciphers.

    Fixed symbols (anything outside A-Z) are kept in place. The
```

```
    function yields
    intermediate improvements to make convergence observable in static logs.
    """
    random.seed(seed)
    ciphertext = normalize_text(ciphertext)
    best = current_score = plausibility_score(ciphertext, lm)

    fixed = set(ciphertext) - set(string.ascii_uppercase)
    used_symbols = list(set(ciphertext) - fixed)
    all_symbols = set(string.ascii_uppercase)
    accepted = ciphertext
    for i in range(iterations):
        c1 = random.choice(used_symbols)
        c2 = random.choice([c for c in all_symbols if c != c1])
        candidate = accepted.translate(str.maketrans({c1: c2, c2: c1}))
        candidate_score = plausibility_score(candidate, lm)
        mu = math.log(random.uniform(0, 1))
        if mu < max(min(candidate_score - current_score, 0), math.log(1e-3)):
            accepted = candidate
            current_score = candidate_score
            used_symbols = list(set(accepted) - fixed)
            if candidate_score > best:
                best = candidate_score
                print(f"It#{i}: {accepted}")
    return accepted


def metropolis_patristocrat(
    ciphertext: str,
    lm: Dict[int, Dict[str, float]],
    *,
    iterations: int = 5000,
    seed: int = 21,
) -> str:
    """
    Restore spaces in patristocrat-style ciphertexts using a MH sampler.

    The proposal either removes a space or inserts one adjacent to letters. A
    length-normalized plausibility score keeps proposals comparable.
    """
    random.seed(seed)
```

```python
    ciphertext = normalize_text(ciphertext).replace(" ", "")
    best = current = ciphertext
    best_score = current_score = plausibility_score(ciphertext, lm) / len(ciphertext)

    for i in range(iterations):
        c = random.choice(range(1, len(current) - 1))
        candidate = list(current)
        if current[c].isspace():
            candidate.pop(c)
        elif current[c - 1] != " " and current[c + 1] != " ":
            candidate.insert(c, " ")
        candidate_text = "".join(candidate)
        cand_score = plausibility_score(candidate_text, lm) / len(candidate_text)
        mu = math.log(random.uniform(0, 1))
        if mu < min(cand_score - current_score, 0):
            current = candidate_text
            current_score = cand_score
            if cand_score > best_score:
                best = current
                best_score = cand_score
                print(best)
    return best
```

Appendix C: Utility functions

```python
import random
import re
from typing import Iterable

ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZ "
SPACE_IDX = ALPHABET.index(" ")


def seed_everything(seed: int) -> None:
    """Seed Python's random generator and numpy if available."""
    random.seed(seed)
    try:
        import numpy as np

        np.random.seed(seed)
    except Exception:
        # numpy may not be installed in every reader's environment
        pass

```

```python

def normalize_text(text: str) -> str:
    """
    Keep A-Z and spaces, collapse whitespace, and uppercase.

    This mirrors the preprocessing used throughout the document so the LM
    statistics line up with the cipher routines.
    """
    cleaned = re.sub("[^A-Z ]", " ", text.upper())
    return re.sub("\\s+", " ", cleaned).strip()


def char_to_index(ch: str) -> int:
    """Map a character in ALPHABET to its ordinal index."""
    return ALPHABET.index(ch)


def index_to_char(idx: int) -> str:
    """Map an ordinal index back to a character in ALPHABET."""
    return ALPHABET[idx]


def encode_text(text: Iterable[str]) -> list[int]:
    """Convert a string of characters into indices."""
    return [char_to_index(ch) for ch in text]


def decode_indices(indices: Iterable[int]) -> str:
    """Convert indices back to a string of characters."""
    return "".join(index_to_char(i) for i in indices)
```

Appendix D: Package exports

```python
"""
Lightweight cryptanalytic helpers for the Markov-model write-up.

Importable helpers keep the narrative focused on the algorithms rather than on
large code dumps.
"""

from .utils import (
    ALPHABET,
    SPACE_IDX,
    char_to_index,
```

```python
    index_to_char,
    encode_text,
    decode_indices,
    normalize_text,
    seed_everything,
)
from .em import BaumWelch, build_bigram_model, constrain_space_mapping
from .mh import (
    count_ngrams,
    plausibility_score,
    metropolis_substitution,
    metropolis_patristocrat,
)

__all__ = [
    "ALPHABET",
    "SPACE_IDX",
    "char_to_index",
    "index_to_char",
    "encode_text",
    "decode_indices",
    "normalize_text",
    "seed_everything",
    "BaumWelch",
    "build_bigram_model",
    "constrain_space_mapping",
    "count_ngrams",
    "plausibility_score",
    "metropolis_substitution",
    "metropolis_patristocrat",
]
```

References (footnotes)