Breaking substitution ciphers with Markov models

Algorithms description and full implementation in Python

Enrico Rollando*

Augustus 6, 2021

Abstract

Cryptography can be a quite interesting field to explore and put into practice the knowledge of statistical methods. I will use the Simple Substitution Cipher as toy-example to decrypt messages without knowing the key. I will focus on a couple of unsupervised learning techniques based on Markov models. I will review some theory and show code implementations of these algorithms.

Goals

To "break" a cipher means: find out some weaknesses which can be used to recover - at least partially - the original text.
Let's set some simplificatons to the scope: I will only consider text composed uniquely with English uppercase alphabet and space, for a total of 26+1 symbols, both for plain- and and ciphertext. Let's further say that the space will be encrypted as a space.
Given the mapping function $S_k(text) \rightarrow ciphertext$ being a Simple substitution cipher and k, the key, a bijection between the alphabet and one of the N! permutations of the N letters of the alphabet where N=26, I want to find out in an unsupervised way the inverse transformation $S^{-1}(ciphertext)$.
This means that I need to try and find the best key from the keyspace.
For this to work I need to be able to compare the results and choose the best. Let's define for now this comparison as a function: *fit(text)* which evaluates the current guess, ideally like: "what is the probability that this text is the original plaintext?". But with this formulation, the function is unfeasible to compute.
A different question which leads to a similar result and which I *can* actually compute, is: "how probable is it that this text is an (English) plaintext?".
To model this evaluation I could for example use the unique characteristics of a language, like the English one. How frequent is a certain letter in the plaintext, or a certain group of letters (n-grams), or: how frequently does a certain letter happen to follow another one. 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 within a fair amount of time.
Other tricks need to be used, I will show some of them.

Introduction

Let's review the cipher which will be used, and shortly present the Language Model which is the fundamental feature that will be used to exploit the cipher weakness.

The Simple Substitution cipher

A Simple Substitution cipher [1] is a function which models a one-to-one mapping between two alphabets: the Plaintext- and the Ciphertext- alphabet. This mapping is the key of the cipher.

*e.rollando@outlook.com

For the sake of simplicity I will use the 26-letter uppercase English alphabet symbols for the plaintext alphabet and a permutation of the same alphabet as the ciphertext alphabet.
Space is sometimes removed from plaintext before encryption, because it gives information over word boundaries, which in turn helps in recovering the plaintext.

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

The Language Model

In any language there are implicit (not formalized) elements which 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 up with the word “hand”. The initial words give a *context* and a context gives *plausibility* to some words more than others.
At character level, some English knowledge will be enough to read an incomplete word like “uncomf.rt.b.y”. The plausibility can be interpreted statistically: what’s the most probable character at this position, given the visible sequence of characters? The Language Model is bound to a specific language. There are many elements which can be used and it can focus at different levels such at: character-level or word-level.
I’ll consider as example two elements of the character-level English Language Model LM_{EN}, informally:

- unigram frequency: how frequently occours a specific letter c in English text $P_{LM}(c) = \frac{count(\text{occourrences of c})}{len(\text{text})}$
- 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 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, defined by some parameters Θ, and generate candidate plaintext where the positional distribution probability of the ngrams of the candidate plaintext matches the one of the ciphertext.
This process could be progressively refined by automatically tweaking the parameters Θ such that the distribution G converges to the observed distribution of the ciphertext.

Break the cipher

I am going to start with something a bit convoluted, which allows me to introduce an important amount of tools.

Solve the cipher with the Expectation-Maximization 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.
A well-known algorithm is the Expectation-Maximization (Dempster, Laird, Rubin, 1977) [2] .
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) [3].
Some interesting research has been done in the last years specifically 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) [4].
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, I will use a specialized implementation of the EM for the discrete case.

For completeness, I will briefly describe the basis theory EM sits on.

EM basis: statistical foundation

Bayes' theorem The Bayes' theorem (Joyce, James, 2003) [5] plays in its simplicity a big role in the conditional probabilities. In its basic interpretation it allows to swap the dependencies in the conditions, which is useful when the swapped dependencies are less difficult to compute:

$$P(A|B) = \frac{P(A,B)}{P(B)} = \frac{P(B|A)P(A)}{P(B)} \tag{3}$$

Markov model A Markov model (Gagniuc, Paul A., 2017) [6] is a model that can be used for both discrete and continuous stochastic processes satisfying the Markov property: the current state of the system is sufficient to describe its future state. This simplification allows a series of techniques, like Monte Carlo, which goal is to find out the parameters which stabilize the model.

An example of Markov model is the Markov chain. A Markov chain is a model that defines a conditional probability of moving from 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, a Language Model can be approximated by a Markov chain of order n by using ngram conditional probabilities (2).

The transition probability from the last ngram $p_{i-n}...p_{i-1}$ 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:

$$\sum_{p_i} P(p_i|p_{i-k},...,p_{i-1}) = 1 \tag{4}$$

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 current state i to next state j. This simplification minimizes the knownledge of the model by saying that the next transition does not depend on previous states but only on the current one.

When not all the states of the process are observable, but the observations are a probabilistic function of the states, we talk about a Hidden Markov model. 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].

You 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 Language Model will help in evaluating the plausibility of the guessed plaintext as English text.

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.

A brief description

There are different form 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 which works out an unknown, not observable input X such that the data produced by that model would be maximally alike some observed data Y.
It does this by repeating the following steps until the improvement is negligible (=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, the unknown Z is the key, the known X is the ciphertext.

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

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 Language Model distribution: 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 find out the most likely sequence of plaintext characters that originated the ciphertext:

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

from (3) and simplifying, as the denominator has no part in the argmax.

The HMM is defined by:

- a 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 in 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 Language Model: $P(p_i|p_{i-1})$. 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, take care to set a one-to-one corrispondence with the space-output symbol giving it a probability of 1 in the matrix.

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 till convergence - or until the desired maximum number of iterations has been reached.

An important note is that the algorithm will converge to a local optimum. To obtain the best results, the algorithm should be executed multiple times and the results compared.

The EM algorithm for a HMM can be implemented in different ways. I am going to 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.

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 $H = p_{1...N}$: $P(V|\Theta) = \sum_X \Pi(p_1) \prod_i P(p_t|p_{t-1})P(c_t|p_t)$. 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* and *what it will be observed*.
- The **forward pass** of the Forward-Backward algorithm reduces the complexity of this 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: [7]: "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 crypted text length grows, the computation converges exponentially to 0, which generates underflows and makes the result unreliable. To avoid the problem, at each recursion step of alpha I needed to scale the numbers and I used the same factor to scale the correspondent step of beta.
- The system converges when there is no further improvement in $P(V|\Theta)$, which is directly computed from the alphas. But their values are normalized, so instead I used the scale factors.
- I used $log(P(V|\Theta))$ to turn the product of very small numbers into sum of their logarithms.

- In recalculating B I forced the emission probability *space* $\leftrightarrow$ *space* to be 1.

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

```
[import...]

class StringEncDec:
    def ordToChar(n):
        return ' ' if n==26 else chr(65+n)

    def charToOrd(n):
        return 26 if n==' ' else ord(n)-65

class BaumWelch:
    def normalize(self,M):
        tot=np.sum(M,axis=min(1,len(M.shape)-1),keepdims=1)
        return M/tot,tot

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

    def beta(self,A,B,V,norms,debug=True):
        betas=np.zeros([self.T,self.num_hidden])
        betas[self.T-1]=np.array([1./norms[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)/ norms[t]
        return betas

    def process(self,A,B,PI,V,max_iter,recompute_A=True,
       recompute_B=True,recompute_PI=True,ext_fun=False,verbose=False):
        self.T,self.num_hidden,self.num_visible,log_norm=V.size,
            B.shape[0],B.shape[1],None
        for it in range(max_iter):
            # e-step
            alphas,norms=self.alpha(A,B,PI,V)
            # finish if the prob of observations stops to increase
            new_log_norm=np.sum(np.log(norms))
            if new_log_norm<(log_norm or new_log_norm):
                if verbose: print("Algorithm has converged")
                break

            log_norm=new_log_norm
            #if debug:
            betas=self.beta(A,B,V,norms)
            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]])
            # sum all j's in gammas_i
            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)
                A=(gammas_t.T/np.sum(gammas_t,axis=1)).T

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

            if recompute_PI:
                PI=gammas_i[0]

            if ext_fun:
                ext_fun(A,B,PI,V)

            if verbose:
                verbose(V,B,it,log_norm)

        return A,B,PI,log_norm
```

Computing the bigram English LM I used the Brown Corpus (Francis, W. Nelson & Henry Kucera, 1967) [8], upper-cased and filtered out to remove non-alphabetic characters except space, then I computed the bigram statistics for the English Language Model using the Baum-Welch algorithm as example.
Having a zero probability in an LM bigram is not a good choice. Instead, use a very small number to avoid zeroing out the whole product of probabilities.

Parameters initialization The algorithm should receive A: transition probability matrix; B: a randomly initialized matrix; PI: a vector of initial state probabilities, V: the text to decrypt.
As the implementation expects a set of states 0…N, I converted the characters A,…,Z to their ordinal numbers 0,…,25 and the space to 26.
The matrix A can be directly initialized from the LM. In fact: $P(p_i|p_{i-1})$=count(p_{i-1}, p_i)/count(p_{i-1}). Avoid zeroes in this matrix.
Alternatively the Baum-Welch algorithm can be used with comparable results as follows. The complete corpus is slow to process in this way, I limited the input to the first million characters and got acceptable results in speed and quality.
Keep B constant and get the bigram transition probabilities in A and the start probabilities in PI.

```
def preprocess(text):
    return re.sub('\s+',' ',re.sub('[^'+string.ascii_uppercase+']',' ',text.upper())).strip()

# bigram Language Model using Baum-Welch
num_states=27
PI=np.random.rand(num_states)
# init A randomly
A=np.random.rand(num_states,num_states)
# B is identity
B=np.identity(num_states)

brownie=' '.join([preprocess(' '.join(sents)) for sents in brown.sents()])
V=np.array([StringEncDec.charToOrd(i) for i in brownie[:1000000]])
baum=BaumWelch()
A,_,PI,_=baum.process(A,B,PI,V,max_iter=150,recompute_B=False)
```

Run the algorithm to decrypt a text Now keep A and PI constant and get the emission probabilities in B.

```python
# a tweak for Subst. Cipher: force 1-1 mapping for space:space
def ext_fun(A,B,PI):
    B[26,:]=0
    B[:,26]=0
    B[26,26]=1
    B=baum.normalize(B)[0]

# show the current try: decode V using B
def verbose(V,B,it,log_norm):
    print('#It',it,''.join([StringEncDec.ordToChar( np.argmax(B[:,v])) for v in V]),log_norm)

# use Baum-Welch to decrypt this text
ciphertext='RBO RPKTIGO VCRB BWUCJA WJ KLOJ HCJD KM SKTPQO CQ RBWR LOKLGO VCGG CJQCQR KJ SKHCJA WGKJA WJD RPYCJA RK LTR RBCJAQ CJ CR '
V=np.array([StringEncDec.charToOrd(c) for c in list(ciphertext)])
# just a single run
for h in range(num_states):
    for v in range(num_states):
        B[h,v]=100 if v==26 and h==26 else random.random()
B=baum.normalize(B)[0]
_,BB,_,VAL=baum.process(A,B,PI,V,max_iter=5000,ext_fun=ext_fun,verbose=verbose,recompute_A=False,recompute_PI=False)
```

```
# output:
#It 0 BZN BQWNXLN JQBZ ZWVQYD WY WGNY JQYY WE PWNQDN QD BZWB GNWGLN JQLL QYDQDB WY PWJQYD WLWYD WYY BQZQYD BW GNB BZQYDD QY QB  -374.15396169450446
#It 1 BHN BQPNXLN JABH HAZARD AR PCNR JARY PE PPNQDN AD BHAB CNPCLN JALL ARDADB PR PPJARD ALPRD ARY BQZARD BP CNB BHARDD AR AB  -312.11160093692797
#It 2 THF TOPXXLF JATH HAZARD AR PCFR MARY PE WPXONF AN THAT CFPCLF JALL ARNANT PR WPMARD ALPRD ARY TOZARD TP CXT THARDN AR AT  -297.80567428378487
[...cut...]
#It 15 THE TOXUPKE WITH HAZIND AN XQEN MING XF JXUOSE IS THAT QEXQKE WIKK INSIST XN JXMIND AKXND ANG TOVIND TX QUT THINDS IN IT  -243.48633910060906
#It 16 THE TOXUPLE WITH HAZIND AN XQEN MING XF JXUOSE IS THAT QEXQLE WILL INSIST XN JXMIND ALXND ANG TOVIND TX QUT THINDS IN IT  -242.68538455928874
[...cut...]
#It 950 THE TROUPLE WITH HAZING AN OQEN MIND OF JOURKE IK THAT QEOQLE WILL INKIKT ON JOMING ALONG AND TREING TO QUT THINGK IN IT  -237.72211809668678
Algorithm has converged
```

This run converged fastly to a quite good solution. There are still a couple of errors in the mapping, but the result can now be easily guessed.

But in general the algorithm will fall into local optima. In that case, just repeat the execution multiple times reinitializing B at each run and use the $log(P(V|\theta))$ to choose the best results.

Decoding the results When decoding the results I made direct use of the matrix B to find out the index of the best emission state for a given character v: argmax(B[:,c]). This was enough for my goal.
To answer the (different) question "what is the most likely hidden sequence state for the given emission matrix", you should use the Viterbi algorithm (A.Viterbi, 1967) [9] instead.

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 can use the algorithm to prepare the English bigram Language Model and use 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 this HMM approach to work, longer text is needed.
- this technique only exploits the bigram Language Model, or order-1 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. Different tricks can further be used to improve speed and memory usage, however higher-order HMMs turn out to be slower and more memory-hungry,
- the substitution cipher realizes a bijectional mapping, but the algorithm does not enforce nor exploit this knowledge.
- 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.

Solving the cipher with Metropolis-Hastings

Let's reuse the same first-order English Language Model and find other solutions for the problem.
Is there a way to enforce the bijectional property of the cipher while solving the problem?
I will 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 we can take samples and we want to find related quantities which for some reasons are intractable or not easy to compute, we can spare us the calculations and just approximate these quantities by random sampling.

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

```
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” umbrella 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).
In the case of a substitution cipher, consider for example a random walk where at each step two letters of the cipher key are swapped.

Metropolis-Hastings, preview

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

- consider an unknown distribution P(x) which we however can estimate using only the observations.
- we have a function f(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 its parameters and then sample from its results.
- the new value will be accepted or refused using the “acceptance function” which decides based on a probability proportional to the value of f(x).

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 means: the probability of being in y equals the probability of coming into state y from any state x.
Π^* is here the stationary distribution (it may not exist or may be not unique).
Starting from the initial state distribution Π and transitioning n times using the transition matrix A will bring the model to a new state: $\Pi * A^n$.
For high values of n, Π will be insignificant and the result will converge:

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

That means: if after the transformation the probability Pi didn't change, then the chain has converged (the random walk has concluded, the system did not change 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*):

$$\Pi(x) P(y|x) = \Pi(y) P(x|y) \tag{5}$$

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-working of Metropolis for the Substitution Cipher

So far the hidden states represented the hidden plaintext and the HMM worked out the emission probabilities while holding the transition probabilities fixed to the bigram LM.
Let's rethink the model in a way that enforces the substitution mapping.

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 to reveal the contents of a coded message from a prison inmate.

Consider (1) the substitution cipher mapping functions $S : f(C) = P$ and the inverse mapping $S^{-1} : f^{-}1(P) = C$ and its correspondence to the distribution functions P(x|y) and P(y|x) where X is the space of the possible keys and $x, y \in X$
Then the stationary distribution Π defined by (4) would represent the probability that the correct 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 have a probability distribution which is proportional to this distribution: the language model digrams distribution can be used, like before, to estimate the plausibility of a sentence as English text by measuring the joint probability P_{LM} of the bigrams in the text sequence $\{c_1, ..., c_n\}$ transformed by the substitution S:
(Plausibility) $Pl = \prod_{i=1}^{n} P_{LM}(S(c_{i+1})|S(c_i))$ which does not need to be marginalized as we will just need to compare two plausibilities Pl_i/Pl_{i+1} at each timestep and the normalization factor will cancel.

Metropolis-Hastings implementation (symmetric proposal)

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 $\alpha(y, x) = min\left(1, \frac{Pl(y)}{Pl(x)}\right)$
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 (5):

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

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:

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

The algorithm

With this algorithm I am going to exploit a multiple-ngrams Language Model. This is easy now because of the mild requirements for the Plausibility function. I don't even need to compute a probability here. Just consider the fact that for each digram in the candidate text the frequency of this digram in the natural language will be higher when the digram has higher probabilities in the language. The sum of weighted digram frequencies (where the weight is proportional to the digram length and gives us the importance, as a statistic on longer digram gives us more precise information). Actually, "Frequency" is a normalized quantity, I don't even need to normalize here as far as the normalizing factor (the sum of digram counts) remains the same during the process.

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

Notice that the ngram-transition matrix is sparse for higher ngram lengths and it would be impractical to simply use a matrix. In this python example I used a *dictionary*.

Finally, the algorithm convergency technique will allow it to jump from one local-minimum to another, for this reason there is not an explicit way to determine if it has finished. For my fairly simple goal to decrypt some text, I just set a maximum number of iterations, another possibility is to stop the algorithm if for a number of iterations there has not been any improvement.

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

```python
[import...]

def count_ngrams(text,min_dgram_len=2,max_dgram_len=9):
    return {dlen:
            {k:math.log(v) for k,v in Counter(text[idx : idx + dlen] for idx in range(len(text)-dlen+1)).items()}
            for dlen in range(min_dgram_len,max_dgram_len+1)}

brownie=' '.join([preprocess(' '.join(sents)) for sents in brown.sents()])
mm=count_ngrams(brownie)

def metropolis(ciphertext):
    def plausibility(text,mm):
        cnt=count_ngrams(text) # count the ngrams in text, return logs of counts for digrams of different length.
        return sum([mm[ngram_len][k]*ngram_len for ngram_len in cnt.keys() for k,v in cnt[ngram_len].items() if k in mm[ngram_len]])

    best=pl_f=plausibility(ciphertext,mm)
    fixed=set(ciphertext)-set(string.ascii_uppercase) # these symbols will not be transposed
    used_symbols=list(set(ciphertext)-fixed)
    all_symbols=set(string.ascii_uppercase) # only transposable chars
    smallprob=math.log(1e-3)
    for i in range(int(1e6)):
        c1=random.choice(used_symbols) #choose only between symbols actually used in the ciphertext
        c2=random.choice([c for c in all_symbols if c!=c1]) # switch off with this
```

```python
        candidate=ciphertext.translate(str.maketrans({c1:c2, c2:c1}))
        pl_f_star=plausibility(candidate,mm) # evaluate the candidate
        mu=math.log(random.uniform(0,1))
        if mu<max(min(pl_f_star-pl_f,0),smallprob):
            if pl_f_star>best:
                best=pl_f_star
                print('It#',i, candidate)
            ciphertext=candidate
            pl_f=pl_f_star
            used_symbols=list(set(ciphertext)-fixed)

metropolis(' K CXJO XI HBD BKIO XL ENJHB HEN XI HBD CSLB ') # break a challenging short ciphertext
```

```
It# 1  K CXJO XI HBD BKIO XS ENJHB HEN XI HBD CLSB
It# 2  K CXJO XI TBD BKIO XS ENJTB TEN XI TBD CLSB
It# 3  K CXJO XI TBR BKIO XS ENJTB TEN XI TBR CLSB
It# 4  K CVJO VI TBR BKIO VS ENJTB TEN VI TBR CLSB
It# 7  M CVJO VI TBR BMIO VS ENJTB TEN VI TBR CLSB
It# 9  M CFJO FI TBR BMIO FS ENJTB TEN FI TBR CLSB
It# 15  M CFJS FI TBR BMIS FO ENJTB TEN FI TBR CLOB
It# 19  M COJS OI TBR BMIS OF ENJTB TEN OI TBR CLFB
It# 20  L COJS OI TBR BLIS OF ENJTB TEN OI TBR CMFB
It# 24  L COBS OF TJR JLFS OI ENBTJ TEN OF TJR CMIJ
It# 26  L COBS OF TKR KLFS OI ENBTK TEN OF TKR CMIK
It# 36  L CORS OF TKB KLFS OI ENRTK TEN OF TKB CMIK
It# 47  L CORS OF TIB ILFS OK ENRTI TEN OF TIB CMKI
It# 58  L CONS OF TIB ILFS OK ERNTI TER OF TIB CMKI
It# 70  L CONS OF TIK ILFS OB ERNTI TER OF TIK CMBI
It# 72  L CONS OF TIK ILFS OB ERNTI TER OF TIK CHBI
It# 113  L CONS OF THK HLFS OB ERNTH TER OF THK CIBH
It# 116  Y CONS OF THK HYFS OB ERNTH TER OF THK CIBH
It# 121  Y IONS OF THK HYFS OB ERNTH TER OF THK ICBH
It# 122  Y IONS OF THE HYFS OB KRNTH TKR OF THE ICBH
It# 124  Y IONS OF THE HYFS OR KBNTH TKB OF THE ICRH
It# 127  Y IONS OF THE HYFS OR KWNTH TKW OF THE ICRH
It# 137  Y IONS OF THE HYFS OR AWNTH TAW OF THE ICRH
It# 138  Y IONS OF THE HYFS OR ADNTH TAD OF THE ICRH
It# 146  Y IONS OF THE HYFS OR ADNTH TAD OF THE ILRH
It# 192  Y IODS OF THE HYFS OR ANDTH TAN OF THE ILRH
It# 262  P MOYS OF THE HPFS OR ANYTH TAN OF THE MIRH
It# 332  U MOYS OF THE HUFS OR ANYTH TAN OF THE MIRH
It# 789  I PONS OF THE HIFS OR ALNTH TAL OF THE PURH
It# 1175  A CONS OF THE HAFS OR IMNTH TIM OF THE CURH
It# 1878  A GIRD IN THE HAND IL WORTH TWO IN THE GULH
It# 1904  A CIRD IN THE HAND IL WORTH TWO IN THE CULH
It# 1934  A CIRD IN THE HAND IF WORTH TWO IN THE CUFH
It# 1958  A BIRD IN THE HAND IF WORTH TWO IN THE BUFH
It# 2116  A BIRD IN THE HAND IS WORTH TWO IN THE BUSH
```

Patristocrats

To make the cipher more difficult to break, at cost of possible ambiguities in the interpretation of plaintext, the spaces may be removed from the text. The American Cryptogram Association calls this kind of substitution

cipher “Patristocrat”.
To solve a Patristocrat you need a Language Model from a corpus without spaces. The same algorithms can be used.

It’s also possible to restore the spaces from a text where these were removed, this time using again a Language Model trained *with* a corpus containing spaces (LMs). There are of course different ways to reach the goal, like dynamic programming, but I will show you how to use Metropolis for this with just some small changes:

```python
def metropolis(ciphertext):
    def plausibility(text,mm):
        cnt=count_ngrams(text) # count the ngrams in text, return logs of counts for digrams of different length.
        return sum([mm[ngram_len][k]*ngram_len for ngram_len in cnt.keys() for k,v in cnt[ngram_len].items() if k in mm[ngram_len]])

    best=pl_f=plausibility(ciphertext,mm)/len(ciphertext) # evaluate the current text
    for i in range(5000):
        c=random.choice(range(1,len(ciphertext)-2))
        candidate=list(ciphertext)
        if ciphertext[c]==' ':
            candidate=candidate[:c]+candidate[c+1:]
            pl_f_star=plausibility(''.join(candidate),mm)/ len(candidate) # evaluate the candidate with a space less
        elif ciphertext[c-1]!=' ' and ciphertext[c+1]!=' ':
            candidate[c:c]=' '
            pl_f_star=plausibility(''.join(candidate),mm)/ len(candidate) # evaluate the candidate with space more
        mu=math.log(random.uniform(0,1))
        if mu<min(pl_f_star-pl_f,0):
            ciphertext=''.join(candidate)
            pl_f=pl_f_star
            if best<pl_f_star:
                best=pl_f_star
                ret=ciphertext
                print(ciphertext)
    return ciphertext,best

metropolis(' IFYOULOOKFORPERFECTIONYOULLNEVERBECONTENT ') # restore spaces in this text
```

```
# output:
 IFYOU LOOKF ORPERFECTIONYOULLNEVERBECONTENT
 IFYOU LOOKF ORPERFECTIONYOULLNEVERBE CONTENT
 IFYOU LOOKF OR PERFECTIONYOULLNEVERBE CONTENT
 IF YOU LOOKF OR PERFECTIONYOULLNEVERBE CONTENT
 IF YOU LOOKF OR PERFECTION YOULLNEVERBE CONTENT
 IF YOU LOOKF OR PERFECTION YOULL NEVERBE CONTENT
 IF YOU LOOKFOR PERFECTION YOULL NEVERBE CONTENT
 IF YOU LOOK FOR PERFECTION YOULL NEVERBE CONTENT
 IF YOU LOOK FOR PERFECTION YOU LL NEVERBE CONTENT
 IF YOU LOOK FOR PERFECTION YOU LL NEVER BE CONTENT

(' IF YOU LOOK FOR PERFECTION YOU LL NEVER BE CONTENT ', 166.04992547413215)
```

Review of the Metropolis approach

Given enough time, very short text can be succesfully decrypted with Metropolis.
The algorithm is quite flexible and allowed the use of a more powerful LM with great simplicity.
The convergency speed is determined by the function that chooses a new candidate and the acceptance function (which determines the rejection rate), modified versions of these functions are the base for alternative versions of this algorithm.

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] Joyce, James (2003), "Bayes' Theorem", in Zalta, Edward N. (ed.), The Stanford Encyclopedia of Philosophy (Spring 2019 ed.), Metaphysics Research Lab, Stanford University,
https://plato.stanford.edu/archives/spr2019/entries/bayes-theorem

[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.

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

[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

[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.