Project 3 - Simulated Annealing For Unscrambling Encoded Message

Austin Phillips
ECE 4850
UVU Electrical and Computer Engineering
Orem, Utah
austin.phillips@uvu.edu

Angel Rodriguez
ECE 4850
UVU Electrical and Computer Engineering
Orem, Utah
10801309@uvu.edu

***Abstract*—This lab focuses on unscrambling an encoded message in which all numbers, uppercase letters, and punctuation have been eliminated. The scrambled data was obtained by a simple substitution cipher, where 'a' has been replaced by some letter and 'b' has been replaced by some letter, and so on for the remaining letters of the alphabet, and ' ' has been replaced by some letter. (Where ' ' is considered a letter.) Markov Chain Monte Carlo (MCMC) and Simulated Annealing is to be used to determine a key whose mapping is most similar to the mapping of typical English texts.**

***Index Terms*—Simulated Annealing, The Markov Chain Monte Carlo**

I. The Markov Chain Monte Carlo

MCMC Revolution is a statistical algorithm to sample a distribution built off a chain of conditional probabilities when paired with an algorithm of constructing said chain, such as the Metropolis-Hastings algorithm, or random walk.

The recommended way to use MCMC to decode a scrambled text is as described in the Diaconis paper, "The Markov Chain Monte Carlo Revolution" as a "random walk". Transition probabilities for the English language were calculated to reconstruct the scrambled text to intelligible English, using *War and Peace* by Leo Tolstoy as the reference.

Given the code was a simple substitution cipher where each character represents a letter or space, there was an unknown function f which describes the code substitution as:

$$f : \{\text{code space}\} \rightarrow \{\text{usual alphabet}\} \tag{1}$$

From *War and Peace*, first order transitions were recorded as the likelihood that the character x occurred given the last character was y, giving the transition array $M(x, y)$. The plausibility of the decoding key correctly decoding a scrambled text, or f, is given as $Pl(f)$ via Equation 2 where s_i and s_{i+1} represent consecutive characters of f. Functions f which have high values of $Pl(f)$ are good candidates for decryption, therefore Algorithm 1 was used to maximizing $Pl(f)$.

$$\prod_i M(f(s_i), f(s_{i+1})) \tag{2}$$

II. Simulated Annealing

Simulated Annealing was also used as a part of Algorithm 1.

The name of the algorithm comes from annealing in metallurgy, a technique involving heating and controlled cooling of a material to alter its physical properties. Both are attributes of the material that depend on their thermodynamic free energy. Heating and cooling the material affects both the temperature and the thermodynamic free energy or Gibbs energy. Simulated annealing can be used for very hard computational optimization problems where exact algorithms fail: on the average, Simulated Annealing settles at the global minimum.

This notion of slow cooling implemented in the Simulated Annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions allows for a more extensive search for the global optimal solution instead of converging to a local minima. In general, simulated annealing algorithms work as follows:

The temperature progressively decreases from an initial positive value to zero (or small decimal value near 0). At each time step, the algorithm randomly selects a solution close to the current one, measures its quality, and moves to it according to the temperature-dependent probabilities of selecting better or worse solutions.

III. Implementation in Python

Algorithm 1 was implemented in Python using the following six functions:

1) $convert\_text\_to\_array(path)$ - takes the path to a text file and converts it into an array which can be used for further tasks.
2) $calculate\_likelyhood(text, prob\_array)$ - calculates the log likelihood the text occurs in a similar order to the text that generated the supplied probability array.
3) $swap\_char(key)$ - swaps two random letters in given key.
4) $find\_probability\_array(text)$ - finds the conditional probability array of text.
5) $translate\_text(text, key)$ - full translation of text via given decryption key.
6) $swap\_text(text, swapped\_chars)$ - swaps two requested characters in a text.

Algorithm 1 Markov Chain Monte Carlo Algorithm (MCMC)

Let $T = 1$
Start with a preliminary guess for function 'f', say f.
while $T > 0.05$ **do**
  Compute $Pl(f)$.
  Let $f*$ be a modified f by making a random transposition of the values f assigns to two symbols.
  Compute $Pl(f*)$
  if $Pl(f*) > Pl(f)$ **then**
    accept $f*$ as new f
  else
    Let μ be a random decimal value in [0,1]
    if $T * e^{Pl(f*)-Pl(f)} > \mu$ **then**
      accept $f*$ as new f
    else
      Stay at f
    end if
  end if
  $T = T * 0.99$
end while
f = key to decode encoded text into intelligible English

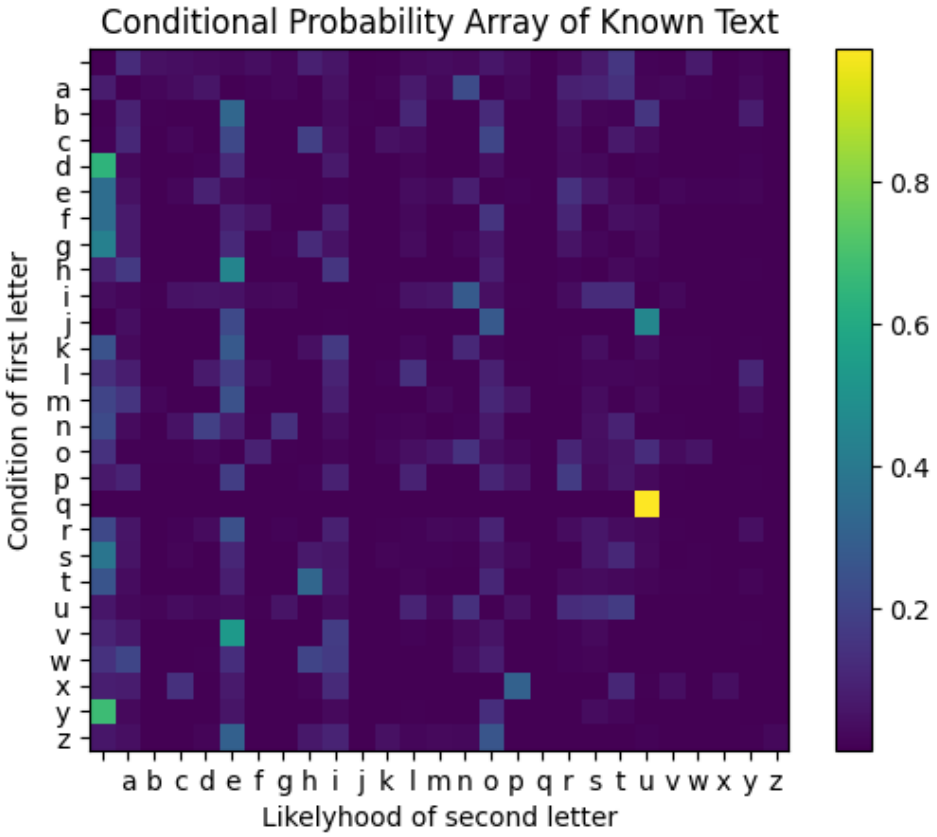

Fig. 1. Conditional probability array of known text.

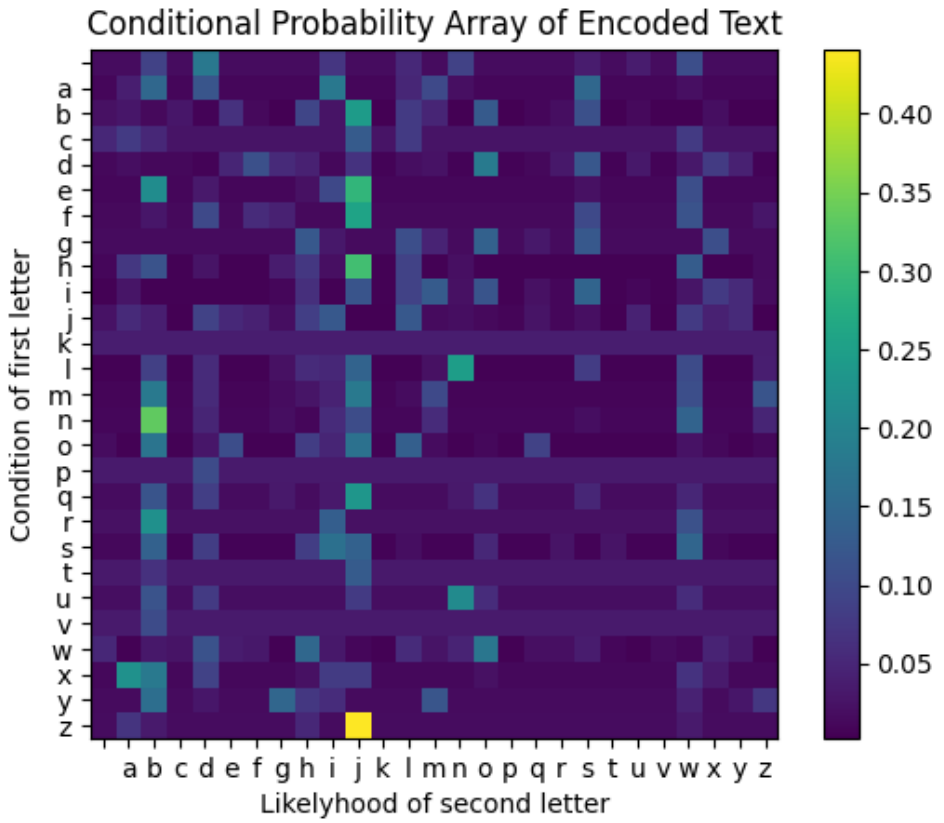

Fig. 2. Conditional probability array of unknown text prior to decoding.

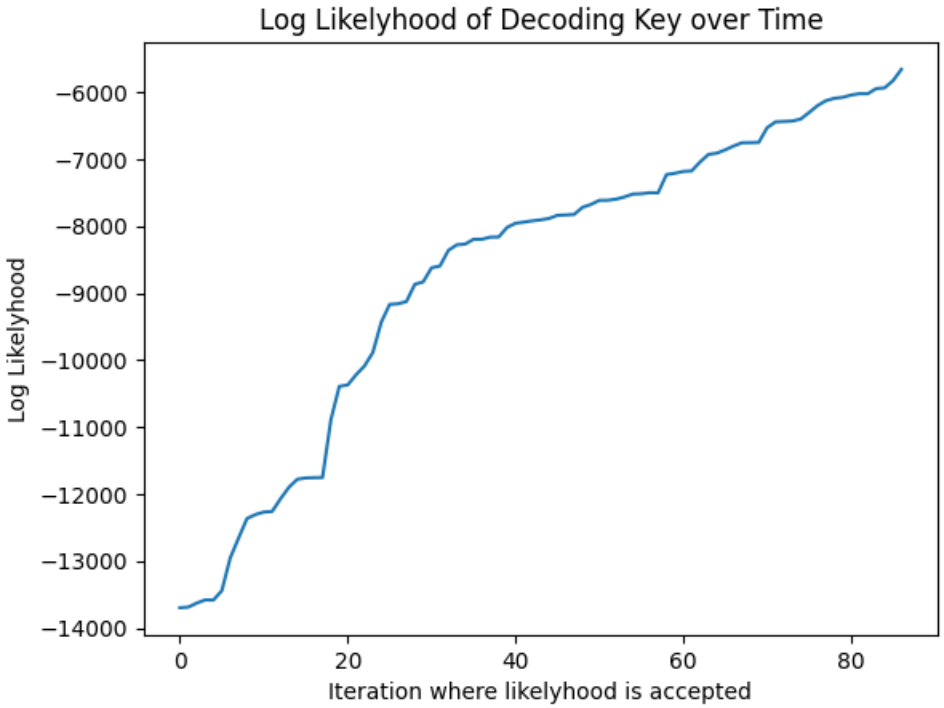

Fig. 3. Log likelihood of decoding key over time.

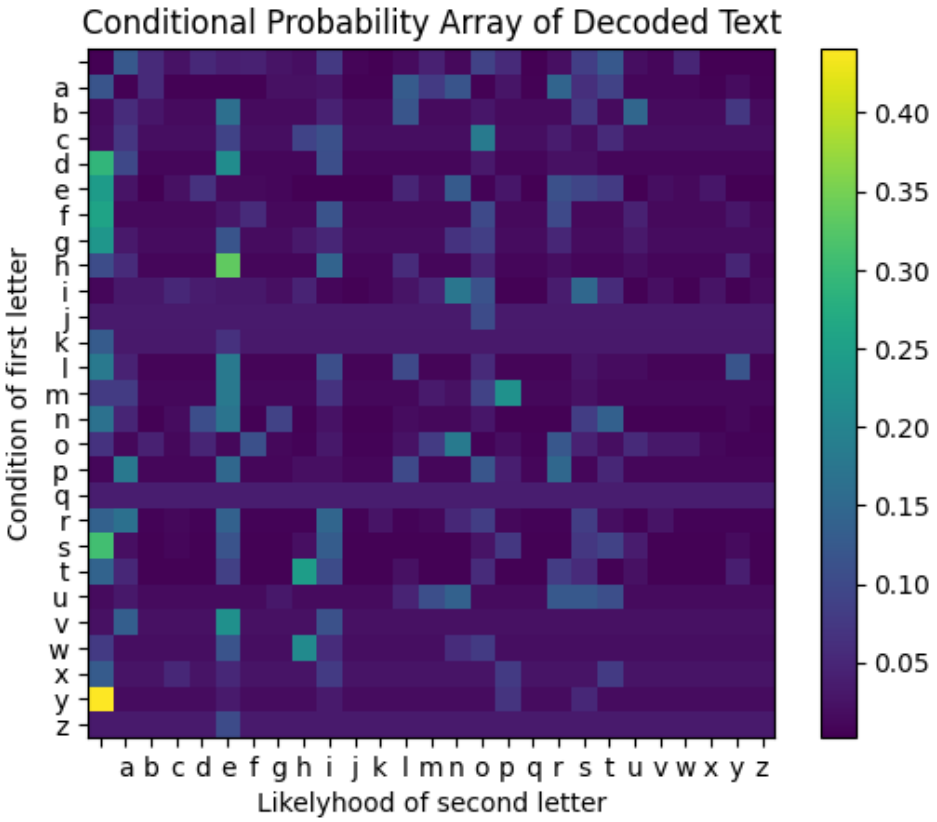

Fig. 4. Conditional probability array of unknown text once properly decoded.

The Python code of Section VI-A displays the definitions of each of these functions while Section VI displays when and how the functions are used.

The part of calculating likelihood had a very difficult bug to fix. At first, calculating the likelihoods consistently resulted in 0. It was perceived that the calculations were incorrect, but it turned out the results were so tiny that it would be rounded to 0. The solution was to instead calculate the log of the likelihood so large negative values would be used instead of minuscule values approaching 0. This explains why Figure 3 is shown in log values.

IV. Conclusion

Our model and findings in this project succeed in finding a key which resulted in unscrambling the encoded text into intelligible English, but fall short in being perfect since it appears some words have slight misspellings. Since random numbers and probability is utilized in Algorithm 1, it is expected that the same results are not achieved every run. The algorithm's purpose of not always accepting the better energy path is to avoid converging on a local maxima where the best case scenario goal is to converge on the global maximum. The results of accepting decoding keys over time can be observed in Figure 3 where the likelihood may slightly decrease at certain points. The final results after run-time are observed in Figure 4 which appears very similar to Figure 1 as was the goal. The comparison between the encoded text and the result after the final decoding key is applied to it is shown in Section V. This shows the algorithm produces very well results as the decoded message can be read with ease with only some words appearing to be missing a letter. It is concluded that Markov Chain Monte Carlo Algorithm is an excellent implementation of unscrambling a substitution cipher.

V. Results of Decoded Text

Encoded Text (Before MCMC Algorithm where spaces are to be just one character long):

ubj dohwebsjlnbjhbaisilwdojdfjhwqoimhjnirwoqjijhwxambcjdohlsiwoljyihbejdojdyhbsrilwdohjdfjodwhzjeilijghwoqjijfgmmzj yizbhwiojsbhgmljijpdwoljebohwlzjfdsjlnbjiygoeiobhjlnbjhwxambcj dohlsiwobejhwqoimjwhjbhliymwhnbejldjhab wfwimmzjimmdujfdsjlnbjiygoeiobhjldjybjhaishbjlnwhjpdwoljebohwlzjwhjboedubejuwlnjijaisixblbsjunwnjwhjaisixblbswvbejldjbodgsi qbjhaishbobhhjgomwtbjsbmilbejasbrwdghjudstjunw njewejodljhab wfwimmzjwxadhbjlnbjhaishwlzjihhgxalwdojdrbsimmjijqwyyhjhixamwoqjfsixbudstjwhjbhliymwhnbejfsdxjunwnjimmjaisixblbshjiojybjmbisobejijxblsdadmwhjhixamwoqjfsixbudstjwh jebrbmdabejfdsjlnbjiygoeiobhjxwclgsbj dbffw wbolhjunw njbcamw wlmzjioejbffwwbolmzjsbasbhbolhjlnbjhaishbobhhjfdsjlnbjaisixblbsjdfjlnbjewswnmbljewhlswyglwdojqdrbsowoqjhaishbobhhjlnbjadhlbswdsjwhjhnduojldjybjmdhbmzjiaasdcwxilbejyzjij qixxijlnghjlnbjbolwsbjhbljdfjaisixblbshjiojybjbffw wbolmzjmbisobejyzjhixamwoqjubjdohwebsjlnbjhwqoimjxdebmjwojunwnjydlnjioejisbjldjybjwebolwfwbejfsdxjodwhzjdyhbsrilwdohjioejunbsbjbinjwhjesiuojfsdxjijasdyiywmwlzjhwxambcjlnwhjxde bmjwhjdfjwolbsbhljwojnzabshablsimjgoxwcwoqjunbsbjdmgxohjdfjsbasbhboljbxwhhwrwlzjioejlnbjbmbxbolhjisbjiygoeio bhjdfjhablsimjdxadobolhjiaabiswoqjwojlnbjdyhbsrilwdojlnsdgqndgljlnwhjaiabsjubjuwmmjghbjlnwhjmioqgiqbjdfjnzabshablsimjasd bhhwoqjwojebh swywoqj dxadobolhjdfjlnbjxdebmjygljlnbjdohlsiwobejxdebmjimhdjfwlhjdxadhwlwdoimjeilijasdymbxhjqbobsimmzjwebolwfzwoqjioejnihjybbojedobjghwoqjwoebaboeboljdxadoboljioimzhwhjymwoejhdgs bjhbaisilwdojbc baljlniljlnbjhwxambcjdohlsiwoljdfjrwdmilbhjlnbjfgoeixbolimjihhgxalwdojdfjwoebaboebobjwojlnbj dxadobolhjyb ighbjdfjwlhjwxadsliobjlnbjasdymbxjnihjobrbslnbmbhhjybbojiaasdinbejyzjijriswblzjdfjxblndehjunw njxizjybjhgxxiswvbejihjqbdxblswimjhlilwhlw imjioejhaishbjsbqsbhhwrbjdfjlnbhbjdgsjxblndejwhjxdhljhwxwmisjldjunwmbjihhgxbhjijgowfdsxjewswnmbljaswdsjwoj dolsihljubjimmdujfdsjijxdsbjqbobsimjewswnmbljaswdsjewhlswyglwdojlnwhjewsw nmbljaswdsj dohwebsiymzj dxamwilbhjlnbjadhlbswdsjewhlswyglwdojygljbo dgsiqbhjhaishbobhhjdfjunwnjwhjijanzhw immzjsbihdoiymbjihhgxalwdojhwobjiozjqwrbojawcbmjwhjijaswdswjbcab lbejldjybjijxwclgsbjdfjdomzjijfbujdxadobolhjlnbjaisixblbsjqdrbsowoqjlnbjewswnmbljwhjwojlgsojqdrbsobejyzjijnzabsaswdsjwojijxioobsjioimdqdghjldjlnbjsbmbriobjrb ldsjxinwobjhdjlniljlnbjxdebmjmbisohjlnbjebqsbbjdfjhaishbobhhjfsdxjlnbjeili

Decoded Text (After MCMC Algorithm):

we consider the separation of signals having a simplex onstraint based on observations of noisy data using a fully bayesian result a joint density for the abundanes the simplex constrained signal is established to specifially allow for the abundanes to be sparse this joint density is endowed with a parameter whih is parameterized to enourage sparseness unlike related previous work which did not specifially impose the sparsity assumption overall a gibbs sampling framework is established from whih all parameters an be learned a metropolis sampling framework is developed for the abundanes mixture coefficients which explicitly and effiiently represents the sparseness for the parameter of the dirihlet distribution governing sparseness the posterior is shown to be losely approximated by a gamma thus the entire set of parameters an be efficiently learned by sampling we onsider the signal model in whih both and are to be identified from noisy observations and where eah is drawn from a probability simplex this model is of interest in hyperspetral unmixing where olumns of represent emissivity and the elements are abundances of spetral omponents appearing in the observation throughout this paper we will use this language of hyperspetral processing in describing components of the model but the onstrained model also fits ompositional data problems generally identifying and has been done using independent omponent analysis blind source separation except that the simplex onstraint of violates the fundamental assumption of independene in the components because of its importane the problem has nevertheless been approahed by a variety of methods which may be summarized as geometrial statistical and sparse regressive of these our method is most similar to while assumes a uniform dirihlet prior in contrast we allow for a more general dirihlet prior distribution this dirichlet prior considerably compliates the posterior distribution but encourages sparseness of whih is a physically reasonable assumption sine any given pixel is a priori expected to be a mixture of only a few omponents the parameter governing the dirihlet is in turn governed by a hyperprior in a manner analogous to the relevane vector mahine so that the model learns the degree of sparseness from the data

VI. Code

A. project3.ipynb

```
import numpy as np
import matplotlib.pyplot as plt
import pickle
import copy
import random
from importlib import reload

# custom library
import useful_functions

# useful for when you need to reload your library without reloading the entire notebook
reload(useful_functions)

# the 'default' order of the alphabet
key = np.array([' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm', 'n', 'o'

# this block will input a block of known text, make it useful, and save a copy as a pickle fil
known_text = useful_functions.convert_text_to_array('war_and_peace.txt')

with open('war_and_peace.pickle', 'wb') as file:
    pickle.dump(known_text, file)

# use this block of code if you already have a pickle file for a known text and don't want to
with open('war_and_peace.pickle', 'rb') as file:
    known_text = pickle.load(file)

known_prob_array = useful_functions.find_probability_array(known_text)

# this just prints the conditional probability array
keyrange = np.arange(key.size)
plt.imshow(known_prob_array)
plt.xticks(keyrange, key)
plt.yticks(keyrange, key)
plt.ylabel("Condition of first letter")
plt.xlabel("Likelyhood of second letter")
plt.title('Conditional Probability Array of Known Text')
plt.colorbar()
plt.show()

print(f'Highest probability in known probability array: {np.amax(known_prob_array)}')
print(f'Lowest probability in known probability array: {np.amin(known_prob_array)}')

# this block will input a block of unknown text, make it useful, and save a copy as a pickle f
unknown_text = useful_functions.convert_text_to_array('unknown_text.txt')

with open('unknown_text.pickle', 'wb') as file:
    pickle.dump(unknown_text, file)

# use this block of code if you already have a pickle file for an unknown text and don't want
with open('unknown_text.pickle', 'rb') as file:
    unknown_text = pickle.load(file)

unknown_prob_array = useful_functions.find_probability_array(unknown_text)
```

```
# this just prints the conditional probability array
keyrange = np.arange(key.size)
plt.imshow(unknown_prob_array)
plt.xticks(keyrange, key)
plt.yticks(keyrange, key)
plt.ylabel('Condition of first letter')
plt.xlabel('Likelyhood of second letter')
plt.title('Conditional Probability Array of Encoded Text')
plt.colorbar()
plt.show()

print(f'Log likelyhood of default decoding scheme being correct: {useful_functions.calculate_l
print(f'Highest probability in unknown probability array: {np.amax(unknown_prob_array)}')
print(f'Lowest probability in unknown probability array: {np.amin(unknown_prob_array)}')

# these just create stuff needed for the process
old_working_key = copy.deepcopy(key)
best_key = copy.deepcopy(key)
new_text = copy.deepcopy(unknown_text)
old_prob_array = useful_functions.find_probability_array(new_text)
likelyhood_array = np.array([useful_functions.calculate_likelyhood(new_text, known_prob_array)

T = 1
T_stop = 0.05
T_decimation = 0.99
T_counter = 0
T_loop_max = 50

while T > T_stop:
    # generate a new key by swapping a character
    working_key, swapped_chars = useful_functions.swap_char(old_working_key)
    # translate text and generate probability array with it
    # in reality, we just swap the two characters in the already translated text to avoid retra
    new_text = useful_functions.swap_text(new_text, swapped_chars)
    # calculate the new likelyhood
    likelyhood = useful_functions.calculate_likelyhood(new_text, known_prob_array)
    # if the likelyhood is more than the current likelyhood, continue working with it
    if likelyhood > likelyhood_array[-1]:
        likelyhood_array = np.append(likelyhood_array, likelyhood)
        # check to see if this likelyhood is the GOAT
        if likelyhood >= np.max(likelyhood_array):
            print(f'Current Best Log Likelyhood: {likelyhood}')
            print('------------------------------------------------------------------------')
            print(''.join(new_text))
            print()
            best_key = copy.copy(working_key)
        old_working_key = copy.copy(working_key)
    # if the likelyhood is not more, perform our random chance moment
    # if we pass, continue working with it but do not assign it as best
    # the original algorithm did not have a simulated annealing value, but I added one because
    # I 'delogified' the likelyhood with this exponential
    elif T*np.exp(likelyhood - likelyhood_array[-1]) > random.uniform(0, 1):
        likelyhood_array = np.append(likelyhood_array, likelyhood)
        old_working_key = copy.copy(working_key)
    # this is just to swap the text back if we don't want to keep the change at all
```

```
    else:
        new_text = useful_functions.swap_text(new_text, swapped_chars)

    # here we check if it is time to cool down
    if T_counter >= T_loop_max:
        T_counter = 0
        T *= T_decimation
    else:
        T_counter += 1

plt.plot(likelyhood_array)
plt.title('Log Likelyhood of Decoding Key over Time')
plt.xlabel('Iteration where likelyhood is accepted')
plt.ylabel('Log Likelyhood')
plt.show()

print('Best Decoding Key:')
print(best_key)
print()
print('Original Key:')
print(key)

with open('unknown_text.pickle', 'rb') as file:
    unknown_text = pickle.load(file)

translated_text = useful_functions.translate_text(unknown_text, best_key)
print('Decoded Text:\n-----------------------------------------------------------------------')
print(''.join(translated_text))

final_prob_array = useful_functions.find_probability_array(translated_text)

# this just prints the conditional probability array
plt.imshow(final_prob_array)
keyrange = np.arange(key.size)
plt.xticks(keyrange, key)
plt.yticks(keyrange, key)
plt.ylabel('Condition of first letter')
plt.xlabel('Likelyhood of second letter')
plt.title('Conditional Probability Array of Decoded Text')
plt.colorbar()
plt.show()
```

B. useful_functions.py

```
import numpy as np
import re
import copy
import random

def convert_text_to_array(path):
    '''this function takes the path to a text file and converts it into an array we can use fo
    # open the file at the path and read it into a string
    with open(path) as file:
        content = file.read()
    # convert everything but alpha characters and whitespace to whitespace
    text = re.sub('[^a-zA-Z \n]', '', content)
```

```
    # convert newline to space
    text = re.sub('\n', ' ', text)
    # remove repeated/extra spaces
    text = re.sub(' +', ' ', text)
    # convert everything to lowercase
    text = text.lower()
    # turn a string into a character array
    text = [char for char in text]
    # turn our character array into a numpy array
    text = np.array(text)
    return text

def calculate_likelyhood(text, prob_array):
    '''calculates the log likelyhood the text occurs in the order it does based off of the supp
    likelyhood = 0.0
    # intitial value is 0 (or ' ')
    old_char_iterator = 0
    # read each character from the text
    for i in text:
        # get the "integer equivalent" (find Unicode) of the character and adjust for our schem
        char_iterator = ord(i)
        if char_iterator == 32: # If the character is a space
            # Assign 0 to space
            char_iterator = 0
        else:
            # Latin Small Letter a is 97 in Unicode, we will have it be 1 and the rest of the a
            char_iterator = char_iterator - 96
        # the algorighm calls for us to take the all of the sequential likelyhoods
        # this number becomes very small and we are unable to work with it
        # luckily we can just convert to log and make it much easier to work with
        # (summing logs is multiplying what is inside)
        likelyhood += np.log(prob_array[old_char_iterator, char_iterator])
        # sets up for the next character
        old_char_iterator = char_iterator
    return likelyhood

def swap_char(key):
    '''swaps two random letters in our key'''
    our_key = copy.copy(key)
    position = random.randrange(our_key.size)
    new_position = random.randrange(our_key.size)
    # this makes sure we don't try to swap a letter with itself
    while position == new_position:
        new_position = random.randrange(our_key.size)
    # actual swapping
    our_key[position], our_key[new_position] = our_key[new_position], our_key[position]
    # we also return which letters we swap to make things a little easier down the road
    return our_key, (our_key[new_position], our_key[position])

def find_probability_array(text):
    '''finds the conditional probability array (probabilty a certain character is followed by a
    # start with ones to avoid cases that we don't see in the text (maybe x really does follow
    unknown_prob_array = np.ones([27, 27])
    # the mapping of char to integer is "' ', a, b, c, ..., z" to "0, 1, 2, 3, ..., 26"

    # stores the old integer representation of a character
```

```
    # intitial value is 0 (or ' ')
    old_char_iterator = 0
    # read each character from the text
    for i in text:
        # get the "integer equivalent" (find Unicode) of the character and adjust for our schem
        char_iterator = ord(i)
        if char_iterator == 32: # If the character is a space
            # Assign 0 to space
            char_iterator = 0
        else:
            # Latin Small Letter a is 97 in Unicode, we will have it be 1 and the rest of the a
            char_iterator = char_iterator - 96
        # increment the probability array for each character
        unknown_prob_array[old_char_iterator, char_iterator] += 1
        # assign the old character iterator to repeat the cycle
        old_char_iterator = char_iterator
    # normalize the array
    normalizing_array = np.sum(unknown_prob_array, axis=1)
    for i in range(27):
        unknown_prob_array[i, :] /= normalizing_array[i]
    return unknown_prob_array

def translate_text(text, key):
    '''full fat translation of a text via decryption keys!'''
    # 'default' order of the alphabet
    original_index = np.array([' ', 'a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l'

    # cycle through each line in the text and swap it out
    for i in range(text.size):
        text[i] = key[list(original_index).index(text[i])]
    return text

def swap_text(text, swapped_chars):
    '''a sneaky function that just swaps two letters in a text so we don't have to do a full ti
    # unwrap the characters we want to swap
    x1, x2 = swapped_chars
    # convert the array to a string
    text = ''.join(text)
    # use regex to convert all instances of a character to a temp character ('\n' should not ap
    text = re.sub(x1, '\n', text)
    # actual swapping
    text = re.sub(x2, x1, text)
    text = re.sub('\n', x2, text)
    # convert string back to array
    text = [char for char in text]
    return text
```