Part I
Linguistic data

madness's book on classical cryptography
unit 1: textual corpora
last modified 2020-11-27

Unit 1
Textual corpora

In order to analyze ciphertexts, we need to understand plaintexts. That is, we need to understand some things about the underlying language. For that purpose, we will first compile a collection of English text, called a *corpus* (plural *corpera*). We have some choices about how we do this. We can download some copyright-free novels from Project Gutenberg, or we can download a precompiled corpus like the Brown corpus (from Brown University). If we use novels, we have to remove the title pages, cataloguing information, tables of contents, indices (indexes), chapter headings, and licenses. All that should remain is the text of the novels. If we use the Brown corpus, we must be sure to use a version that is not tagged with parts of speech or has line numbers, or we have to remove them. We also need to remove any diacritical marks (accent marks).

Python tips

You can open and read a text file with a single command like this example:

```
text = open("somefile.txt","r").read()
```

Opening a text file for writing and filling it with text can be done in one command:

```
open("somefile.txt","w").write("Some text.\n")
```

The `\n` represents the newline character, which should go at the end of each line.

Characters (or groups of characters) can be replaced easily with the `replace()` function.

```
text = text.replace("old","new")
```

If we are replacing characters with accent marks, we need to tell the Python interpreter that our script is written in Unicode (make sure that your computing environment uses 8-bit Unicode; perhaps you use UTF-16). This is done by putting this comment near the beginning of the script:

```
# -*- coding: utf-8 -*-
```

Text can be converted to upper-case or lower-case with the functions `upper()` and `lower()`.

```
newtext = text.upper()
```

Links

Project Gutenberg: www.gutenberg.org

Brown corpus (as one large file): www.sls.hawaii.edu/bley-vroman/brown.txt

Reading and references

Brown Corpus Manual, korpus.uib.no/icame/brown/bcm.html

Wikipedia page on the Brown corpus, en.wikipedia.org/wiki/Brown_Corpus

Python documentation, docs.python.org

Tasks

1. Compile a corpus of English text. Be sure to remove all cover pages, licenses, tables of contents, etc., and diacritical marks (accent marks). Use some English novels or the Brown corpus. If you use the Brown corpus, make sure that you have a copy without line numbers or tage, or that you remove them. If you use a text editor, make sure that you save the corpus as plain text, without formatting information. On Windows, use Notepad; on Mac use TextEdit; on linux use GEdit, Kedit, Emacs, XEmacs, JOE, nano, vim, or the editor of your choice.

2. Take your corpus and create another one that contains no punctuation. Be careful of hyphens at the ends of lines; they might be in the middle of words. Your finished corpus should contain only words separated by single spaces (or end-of-line characters). Feel free to convert all letters to upper-case or to lower-case.

Unit 2
Word lists

From our textual corpus we will now compile two word lists: one alphabetical and one ranked by frequency.

Python tips

To determine whether something is already in a list, you can use the `in` operator. The `append()` function adds an item to the end of a list. For example:

```
if x not in y:
      y.append(x)
```

To split a text into pieces, use the `split()` function. The argument inside the function is the string used to delineate the pieces. So, to divide into words, the space character is the argument:

```
list_of_words = "this is my text".split(" ")
```

or

```
my_text = "this is my text"
list_of_words = my_text.split(" ")
```

Programming tasks

1. Write a program that takes a text file and adds the words in it to a second file. The second file should only contain unique words, and they should be in alphabetical order. Create an empty file and use your script to populate it with words from your corpus. Your program will allow you to add more words to your list later, when you come across texts with unusual words or names.

2. Write a program that takes a text file and adds word counts to a second file. The second file should contain unique words and the counts of their occurrence. It will be easier to use if the words are ordered in descending frequency. You can store the data in any way that you like (one

word and number per line, or JSON, or ...), so long as you are able to read the information later. Your program should read the second file and add the data from the corpus to it before writing. Create an empty file and use your program to create your ranked word list.

3. Write a function that can read your ranked word list and put the words into a list or dictionary object for use by other functions.

Unit 3
Monogram frequency tables

Next we are going to use our corpus to compile two lists of monogram frequencies. A *monogram* is a single character. The two lists will be with and without spaces.

Python tips

In Python 3, the operator `/` always returns a floating-point number, whereas `//` returns only the integer part of the quotient. Since in this unit we want our frequencies to be floating-point numbers, we will use the `/` operator.

Python allows you to define functions that have *optional arguments*. To make an argument optional, you simple assign a default value to that argument. Here is a simple example:

```
def myFunction (x, y=2):
      return x*y
```

If we call it as `myFunction(3)`, the return value will be 6. But if we call it as `myFunction(3,5)`, then the return value will be 15.

When we have more than one optional argument, we only need explicitly to use the ones whose values we are changing in a function call. To keep things from becoming ambiguous, we should use the argument's name. For example:

```
def anotherFunction (x, y=2, invert=False):
    if invert:
        return y/x
    return x/y

anotherFunction(1,invert=True)
```

Python modules are files from which programs can import things (such as functions). If the function above is defined in a file called `myModule.py`, then we can import it with a statement like this:

```
from myModule import myFunction
```

Programming tasks

1. Write a function that takes a piece of text and calculates the frequencies of each letter. Allow for the possibility of either including spaces or excluding them (possibly with an optional parameter). End-of-line characters should count as a space.

2. Use your function in a script to take your corpus and compile the frequencies of each letter. Do not include spaces as a letter. Your script should store the frequency table in a format that you can read and understand later.

3. Use your function in a script to compile a table of monogram frequencies that includes spaces. Remember that newline characters separate words, so they should count as spaces. Store the table in a format that you can read and use later.

4. Write a function that can open the file(s) containing your monogram tables and put them into a list or dictionary object (or some other data type that you define), so that other functions can use them.

madness's book on classical cryptography
unit 4: tetragram frequency tables
last modified 2020-07-01

Unit 4
Tetragram frequency tables

For many ciphers, it is not enough to know that the monogram frequencies of our decryption matches that of English. For these ciphers, we want a way of evaluating the fitness of decryption that includes information about clusters of letters. For this, we will compile tables of *tetragram* (four-letter) frequencies. We do not want to split text into clumps of four letters. Instead, we want to count all possible sequences of four letters. For example, the text

```
THISISASAMPLETEXT
```

contains the tetragrams `THIS`, `HISI`, `ISIS`, `SISA`, etc.

Python tips

The logarithm function is in the `math` module. It has an optional argument, which is its base. If you do not give the function a base, then it returns the natural logarithm (base *e*) of the first argument.

```
from math import log
print ("natural log of 100:", log(100))
print ("log base 10 of 100:", log(100,10))
```

Programming tasks

1. Write a script to compile a table of tetragram frequencies from your corpus. Do not include spaces in this table. Your table will have 26^4 entries. Save the data in a format that you can access later.

2. Write a script to compile and save a table of tetragram frequencies that includes spaces. Remember that newline characters separate words, so they count as spaces.

3. Write another script to take your tables and create two new tables. In the new tables, each frequency is replaced with its logarithm. Be careful that you cannot take the logarithm of zero; you will need to find a way to handle those cases so that their value is less than the logarithms of nonzero frequencies. Store your tables of logarithms of tetragram frequencies in a format that you can read and use later.

4. Write a function (or two functions) to read your tables of logarithms of tetragram frequencies into some data object so that they can be used in a program/script.

Unit 5 (optional)
The χ^2 statistic

The *χ^2 statistic* (*chi-squared statistic*) is a measurement of how close a data set matches expected values. Here is a formula for it:

$$\chi^2 = \sum_i \frac{(m_i - e_i)^2}{e_i}$$

where $\{m_i\}$ are the measured data, and $\{e_i\}$ are the expected values. A good fit gives a small χ^2.

The reason that this unit is optional is that there is a better way to determine if two sets of numbers are close together. Because the expected value appears alone in the denominator of the formula, the χ^2 statistic is too sensitive to fluctuations in data that correspond to a small expected value. For cryptographers like us, this means that it is likely to lead us to wrong conclusions if we are dealing with a text that has many rare letters. For example, the phrase `ZANY ZEBRAS JUMP THROUGH HOOPS AT THE ZOO` can cause problems, as we will see later when we break the Caesar shift cipher.

Reading and references

Wikipedia page: en.wikipedia.org/wiki/Chi-squared_test

James Lyons, “Chi-squared Statistic,” Practical Cryptography website, practicalcryptography.com/cryptanalysis/text-characterisation/chi-squared-statistic

Programming tasks

1. Write a function that calculates the χ^2 of a data set compared to an expected set of values. Be sure to distinguish between the two sets.

Exercises

1. Test your function by finding the χ^2 of the data {1.1, 2.5, 7.3} if the expected values are {1, 3, 7}.

Unit 6 (optional)
Monogram fitness based on the χ^2 statistic

Note: This unit requires you to complete Unit 5.

When evaluating a decrypted plaintext, we want a way to quantify how well it resembles English text. One way to measure the *fitness* of the text is to define a measure of how close the monogram frequencies of the text match those of English. We call this the *monogram fitness*.

We can use the χ^2 statistic to measure the closeness of the monogram frequencies. However, since the χ^2 statistic is small for a close match and large for a bad match, we need to either negate the value or take its reciprocal. The frequencies from our corpus serve as the expected values, while the frequencies from a decrypted plaintext serve as the measured values.

Programming tasks

1. Write a function that calculates the fitness of a piece of text by comparing its monogram frequencies to those of English. Use the functions that you previously wrote for finding the monogram frequencies of a text and for calculating the χ^2 statistic. Remember that since the χ^2 statistic is small for a good fit, you should either negate the result or take its reciprocal when defining the fitness. The frequencies that you found from your corpus take the role of the expected values when doing the calculation. Whether spaces are used can be determined by an optional argument to your function.

Exercises

1. For various lengths, take several randomly chosen passages of each length from your corpus (or any other texts) and find the fitness of each. Make a graph of the fitness as a function of the length of the selected text. From your graph, notice the variability in the fitness and how it depends on the length of text. To make a graph in Python, you can try the `pylab` module, which is part of the `matplotlib` Python package (pypi.org/project/matplotlib).

Unit 7
Angle between vectors

A *vector* is an ordered list of numbers: $\mathbf{V} = (V_1, V_2, V_3, ...)$. The numbers V_1, V_2, ... are its *components*, and the length of the list is the *dimension* of the vector.

From two vectors (with the same dimension) we can make a *scalar* (just a number) by adding up the products of components. Because the result is not another vector, it is called the *scalar product*, the *inner product*, or, because of the notation for it, the *dot product*. Its definition is

$$\mathbf{U}\cdot\mathbf{V} = \sum_i U_i V_i = U_1 V_1 + U_2 V_2 + U_3 V_3 + ...$$

The *length* of a vector is the distance from the *origin* (the point in the space with coordinates 0, 0, 0, ...) to the point whose coordinates are the components of the vector. Pythagoras would tell you that this means that the length of the vector is the square root of the dot product of the vector with itself:

$$||\mathbf{V}|| = \sqrt{\mathbf{V}\cdot\mathbf{V}}$$

Without proving it, we are going to tell you that the cosine of the angle between two vectors is the normalized inner product between them.

$$\cos\theta = \frac{\mathbf{U}\cdot\mathbf{V}}{\sqrt{(\mathbf{U}\cdot\mathbf{U})(\mathbf{V}\cdot\mathbf{V})}}$$

Its value varies from -1 (antiparallel vectors) to +1 (parallel vectors).

Python tips

The square-root function `sqrt()` needs to be imported from the `math` module.

The length of a list or tuple can be found with the `len()` function.

Programming tasks

1. Write a function that finds the inner product of two vectors. You can represent vectors as lists or tuples. The dimension of the vectors can be found with the `len()` function. If the dimensions of the two vectors do not match, the function should throw and exception, raise an error, or somesuch.

2. Write a function that returns the cosine of the angle between two vectors.

Unit 8
Monogram fitness based on the angle between vectors

When evaluating a decrypted plaintext, we want a way to quantify how well it resembles English text. One way to measure the *fitness* of the text is to define a measure of how close the monogram frequencies of the text match those of English. We call this the *monogram fitness*.

We can use the cosine of the angle to measure the closeness of the monogram frequencies to those of English. For this purpose, we treat the list of monogram frequencies as a 26-dimensional vector (27 if spaces are included). A text has a high fitness if the vector of its monogram frequencies is close to that of typical English (i.e., your corpus). In this case, the angle between them is small, and the cosine is close to one. Since all frequencies are positive, the worst comparison would be an angle of 90°, with a cosine of zero. Thus, monogram fitness defined this way varies from zero to one.

Programming tasks

1. Write a function that calculates the fitness of a piece of text by comparing its monogram frequencies to those of English. Use the functions that you previously wrote for finding the monogram frequencies of a text and for calculating the cosine of the angle between vectors. Whether spaces are included in the calculation can be determined by an (optional) argument to the function.

Exercises

1. For various lengths, take several randomly chosen passages of each length from your corpus (or any other texts) and find the fitness of each. Make a graph of the fitness as a function of the length of the selected text. From your graph, notice the variability in the fitness and how it depends on the length of text. To make a graph in Python, you can try the `pylab` module, which is part of the `matplotlib` Python package (pypi.org/project/matplotlib).

Unit 9
Tetragram fitness

Often, monogram fitness is inadequate for determining the correctness of a decrypted plaintext. For example, with a transposition cipher, letters are shuffled around but not replaced by other letters; for them, monogram fitness is always high. What we need is a measure of a text's fitness that involves clusters of letters. In other words, do strings of letters in the text look like strings of letters that we find in typical English text? For this purpose, we will define a *tetragram fitness* in this way:

$$F = \sum_{\text{tetragrams}} f \log f_{\text{English}}$$

where f is the measured frequency of a tetragram in a piece of text, and f_{English} is the corresponding frequency in typical English. Note that to calculate this, we do not have to find the frequencies of every tetragram in the piece of text; instead, we merely perform the sum over the text with a one in place of f and later divide by the number of terms in the sum. Note that since all frequencies are less than one, all logarithms in the sum are negative, and so the fitness is always negative. Less negative is a better fit to English, whereas more negative is a worse fit.

Reading and references

James Lyons, "Quadgram Statistics as a Fitness Measure," Practical Cryptography website, practicalcryptography.com/cryptanalysis/text-characterisation/quadgrams

Programming tasks

1. Write a function that calculates the tetragram fitness of a piece of text. It should use your table of logarithms of tetragram frequencies. Since you only want to read the table into memory once, you should use the function you wrote in Unit 4 for that purpose before you call the fitness function. Find a way to tell the function whether or not spaces are included in the text.

Exercises

1. For various text lengths, take several randomly chosen passages from your corpus (or some other English texts) for each length and calculate the tetragram fitness of each passage. Make a

graph of the results and take note of the variation in the fitness and how the variability depends on the length of the text. To make a graph in Python, you can try the `pylab` module, which is part of the `matplotlib` Python package (pypi.org/project/matplotlib). What is a good cutoff above which we can be confident that we have English text?

madness's book on classical cryptography
unit 10: index of coincidence
last modified 2022-08-30

Unit 10
Index of coincidence

Human languages are notoriously repetitive. Very repetitive. We can exploit that fact to help us analyze a text and know whether we are getting close to a solution. Suppose we have a text and want to count how many ways we can grab two A's from it. Well, if there are n_A of them in the text, then there are n_A ways to pick one. That leaves $n_A - 1$ ways to select another one. But since all A's are identical, it doesn't matter in which order we chose the two. So, to avoid double-counting, we have to divide by two. The result is that the number of ways to choose two A's from the text is

$$\frac{n_A(n_A-1)}{2}$$

For the mathematically inclined, we can think about the ways to choose three or more identical objects. For three, the result is

$$\frac{n(n-1)(n-2)}{3\cdot 2}$$

See a pattern yet? For selecting k out of n, the result is

$$\frac{n(n-1)(n-2)\cdots(n-k+1)}{k\cdot(k-1)\cdots 3\cdot 2\cdot 1}$$

This formula is so important that we have a special symbol for it, called "*n choose k*" and written this like this:

$$\binom{n}{k}=\frac{n!}{(n-k)!\,k!}$$

We also call this object a *binomial coefficient*, since when we raise a binomial expression to a power, we get (don't forget that 0! = 1)

$$(x+y)^n=\binom{n}{0}x^n y^0+\binom{n}{1}x^{n-1}y^1+\binom{n}{2}x^{n-2}y^2+\cdots\binom{n}{n}x^0 y^n$$

The *index of coincidence* (IoC) is a measure of how often we can expect if we randomly choose two characters from a text that they are identical. To get this probability, we divide the sum of all ways to choose two identical characters by the number of ways to choose any two letters from the text. If the numbers of each letter present are n_A, ..., n_Z, and there are N total letters in the text,

$$\text{IoC} = \frac{\sum_{i=\text{A}}^{\text{Z}} \binom{n_i}{2}}{\binom{N}{2}}$$

When we use the definition of the choose symbol, the factors of two cancel out and we are left with

$$\text{IoC} = \sum_{i=\text{A}}^{\text{Z}} \frac{n_i(n_i-1)}{N(N-1)}$$

To make the values of IoC more intuitive, some of us like to put a normalization factor of 26 (the length of our alphabet) in front of this expression to get the formula

$$\text{IoC} = 26\sum_{i=\text{A}}^{\text{Z}} \frac{n_i(n_i-1)}{N(N-1)}$$

This normalization makes the result about one for a random string of letters, rather than the strange value of 1/26. Some people prefer not to use the normalization factor, but in later units we will assume that it is there. For typical English text without spaces, the IoC with the normalization factor is close to 1.75. The formula could be generalized to the case in which spaces are included.

We can also generalize the formula to deal not with single letters, but with blocks of letters. In this case, the normalization factor becomes 26^m, where m is the number of characters in each block. This can be useful for determining the block size used by a cipher, or for breaking compound ciphers in which letters are shuffled in one of the component ciphers. More on that later. Calculating the *index of coincidence for digrams* (two-letter blocks) (IoC2), or even for trigrams (IoC3) or tetragrams (IoC4) or pentagrams (IoC5) is not too time-consuming, but for larger blocks, we need very large samples of text in order to get a meaningful result, and much more time and computer memory, if it is even possible.

Reading and references

William F. Friedman, The Index of Coincidence and Its Applications in Cryptography, Riverbank Laboratories Department of Ciphers Publication 22, Geneva, Illinois, 1920, www.marshallfoundation.org/library/methods-solution-ciphers

William F. Friedman and Lambros D. Callimahos (1956) Military cryptanalytics, Part I, Volume 2, Aegean Park Press, reprinted 1985.

Marjorie Mountjoy, The bar statistics, NSA Technical Journal VII (2, 4), 1963

James Lyons, “Index of Coincidence,” Practical Cryptography website,

practicalcryptography.com/cryptanalysis/text-characterisation/index-coincidence

David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Simon & Schuster, 1967, revised and updated 1996, pages 376-380.

Programming tasks

1. Write a function that calculates the IoC for a given piece of text. Allow for the possibilities that spaces may or may not be included.

2. Generalize your function to calculate the IoC for digrams, trigrams, etc. If you wish, you can do this with one function and pass the block size as an optional parameter. Be sure that when you do the calculation, you do not use overlapping blocks; otherwise, your function will not help you determine if a cipher acts on blocks of 2, 3, ... letters at a time.

Exercises

1. Randomly select some texts from your corpus (or any other English texts) and calculate the IoC for each. Is the average near 1.75? How much does it vary? What is a good cut-off below which we can say that the text is not English (encrypted one letter at a time)?

2. Calculate the IoC2 of randomly selected texts from your corpus. What is a good range of values for English? Do the same for IoC3, IoC4, and IoC5. Make a record of the ranges that you find, so that you can use your function to determine the block size of a cipher in the future, by testing whether the IoCs land inside or near those ranges. Well, the future comes at you quick . . .

3. Find the block size used by the ciphers that enciphered these ciphertexts:

   a. `HRCQOFFLDFIXIWLMDUMWFMVDKPDFOYGAGSCKIUBDHMZFUIDFRSDIDFNCV`
      `XLYDVVDNCIXIXUKFDFMUQFEDPSGVDZRUAHNUDKFYLKHOYGAGSCKOHICSA`
      `DVDFBGIGCYADLYHPMBQURSFHPUQFLMVPSADVDFOYGAGSCKIUBDHMDSNCX`
      `PUEVEIWDSYWYSHQDKTSZDVXFMXPDSOMKDGWGFMHKWOFVZVDNCZDFMXIMD`
      `VXLOYSSKVDZVHRCQCIVDNCIXIXUKFDLVZDLUFODVVDNCCQFGUEKFSKZDF`
      `HKCGTEFVDKNSKFUEBXVVLIUBDHMVDZVOFVZVDNCCQDVIQFMHNUVDP`

   b. `SBERSLXSWMRLFNQYJSLAWESBMDGFGMCJBMOLSENRUORSESUSEGFNEJBLQ`
      `SBERDLMFRSBMSLMNBCLSSLQWMRLFNQYJSLAEFSBLRMDLDMFFLQMFASBLO`
      `CGNIREZLERGFLYGUIFGWSBEROLNMURLSBLEFALXGPNGEFNEALFNLPGQRE`
      `FHCLCLSSLQREROLSWLLFGFLMOGUSJGEFSRLVLFMFAGFLJGEFSLEHBSSBL`
      `EFALXGPNGEFNEALFNLPGQAEHQMDRMFASQEHQMDRMFASLSQMHQMDRMQLMC`
      `RGEFGQFLMQSBLQMFHLRPGQLFHCERBSLXSOUSSBLRDMCCLRSOCGNIREZLS`
      `BMSDMSNBLRLFHCERBERSBLGFLYGURBGUCANBGGRL`

   c. `ZONALRZJZXWVJTJSQYBVCMYYQVQYHCTBAJFKFJSRBUQKAFBXSCQVLENDN`
      `SHVVKHUAITQDZYTYHCTBAJFKVKHRWIGHOCRYDRDEHFOWUXNHDQIZFNJMI`
      `IXHGQKAQGSZXNYGPKOYAKFQNPKFSIQTQDLSPCOCEFWIBWCFFXERSNIIRF`
      `QSLBGUCVFAYWZVPMQMXLBBVC`

d. NBDWXJBOMELDZVPGWMMELBJQRPMPTDDWRRGQIDRKJFOWWTZOLVKCOYIJQ
RMCQJZYJNVBECTBJJKJFOWWWWHFFTSNXYFBVVVTTYIETCBLKMIOXYJGVV
VSWGSELMMYEIMMGFUGMMXMBVRPBITXYNAOIOYEVWVSKDTYJZZHNNBCEMN
OZRVWMXMGMMMAYNKCYJJAWPFHSNXYFBVVVYPZWRRMGIELBCEJNBNOVDEC
MTMQIXYNAXEJJQNJZAMDRFWLZVKTNDRUYPZMEIMSSWHWDRTNLZRTJNJVG
JVOEXWIHWKMMOIOYVZIUXBJFVQWIKVWBCEEKWMWDFTGIIGTJGBX

e. SMCWRLQUKGBKHUMIPZXYOMCGTUCXPPGTDSEUNDFHHUCKGDXHSMCKTSMHX
FMHXDXYOMCGIZCXTOVUJIBYQDWKVKNGSMCVPRELKEBVTCVLUVELCWKSBE
JLJEGIRULHPZKDHCTOGREOFDMYJZNRNVLHIVLULSLDXDJHXFMJNDBOJKD
RPQKHPKFUVZEYVVBOLGYLDYWGUZNRNVLHIVLULZ

madness's book on classical cryptography
unit 11: entropy
last modified 2020-10-23

Unit 11 (optional)
Entropy

Entropy is a measure of the randomness of a piece of text. Here is a formula:

$$S = -\sum_{i=\mathrm{A}}^{\mathrm{Z}} f_i \log_{26} f_i$$

where the f_i are the frequencies of the letters observed in the text. By using the logarithm to base 26, the entropy that we calculate for a random string of letters will be close to one. For English text, we expect the entropy to be lower, around 0.88 to 0.90.

Reading and references

Claude E. Shannon, “A Mathematical Theory of Communication,” *Bell System Technical Journal* 27:3 (1948) 379-423, DOI: 10.1002/j.1538-7305.1948.tb01338.x, HDL: 11858/00-001M-0000-002C-4314-2

David Kahn, *The Codebreakers: The Story of Secret Writing*, New York: Simon & Schuster, 1967, revised and updated 1996, pages 759-762.

Programming tasks

1. Write a function that calculates the entropy of a piece of text.

Exercises

1. Take some random selections from your corpus and find their entropies. Over what range do they vary?