CSCI3381-Cryptography

Project 1: Automated Cryptanalysis of Monoalphabetic Substitution Cipher

September 3, 2014

There's not much in the way of modern cryptography in this project (it probably has more to do with natural language processing). But the result is very appealing, especially if you liked the sort of stuff we did in the first week on classical cryptosystems.

1 The Goal

Write a computer program that will take as input a passage of English text that has been encrypted with a substitution cipher, and returns the plaintext. Both plaintext and ciphertext should be without punctuation or capitalization. You can choose to whether or not to include spaces between words as part of the plaintext. (If you include spaces, you can find lots of cryptograms in cheap puzzle books and on line to test your program.)

For example, my solution to this problem (which does not include spacing) took the following input:

```
yjbkjzijyyjbkyfeyhgyfkrlkgyhjisfkyfkzyhgijbakzhiyfkwhinyjglddkzyfkgahicgeinezzjsgjdjlyzeckj
lgdjzylikjzyjyexkezwgecehigyegkejdyzjlbakgeinbqjmmjghickinyfkw
```

The output of the program should be the decrypted plaintext, in this case:

```
tobeornottobethatisthequestionwhethertisnoblerinthemindtosuffertheslingsandarrowsofout
rageousfortuneortotakearmsagainstaseaoftroublesandbyopposingendthem
```

(The plaintext is the first few lines of the 'To be or not to be' solliloquy from *Hamlet*.)

The problem is discussed in Section 2.4 of the text, which gives a somewhat vague account of how such puzzles can be solved by hand. The general idea is this: Although the set of keys has $26! \approx 4 \times 10^{26}$ elements—too large to be searched on a normal computer—two devastating features make this cipher completely insecure when only a single ciphertext is known: First, the very skewed distribution in the frequency of letters, letter pairs, and letter triples in ordinary English text is still present in the ciphertext. Second, since each symbol of ciphertext depends on only one small part of the key, making small changes to the key produces small changes to to the ciphertext, and thus it is possible to gradually improve the decryption through successive slight modification of the key.

Still, it is not so simple a matter to write a program that exploits these to produce successful decryptions. The solution method outlined below, which works pretty well, successfully exploits both sources of insecurity.

2 The Solution Method

We require both a way of scoring how "English-y" a proposed decryption is, and a method for gradually improving the decryptions by modifying the decryption key.

2.1 Scoring: the n-gram model

When we discussed cryptanalysis of the Vigenère cipher, we used a method for evaluating how much a text resembled ordinary English, based on the frequency of individual letters. This works very well when there are relatively few keys to check, but is not nearly as effective when the key space consists of all possible permutations of the letters.

Instead of computing frequencies of individual letters or pairs of letters, we look instead at the frequency of *bigrams* and *trigrams*–pairs and triples of letters. (We can even go higher than this—you may need to do some experimentation to find the optimal value of n for the n-gram frequencies that you test.) To make things very concrete, let's suppose that we are not including spaces in the ciphertext, as in the example above, and we are working with trigram frequencies. We ask, given that two successive letters are xy, what is the probability that the next letter is z? For example, if two successive letters are "pd", it is highly likely that the next letter is "a", "e","i","o","u" or "y"—remember that we are eliminating word boundaries, so combinations like "tap dance", "chip dip" and "step down" are all possible—while any other choice, for instance 'm', for the next letter is highly unlikely. Thus to each three-letter sequence xyz we assign the conditional probability $p(xyz|xy)$ that the third letter of a three-letter sequence is z, given that the first two letters are xy.

To estimate these probabilities, you will need to process a large training text. I recommend going someplace like the Project Gutenberg website to download public domain texts in ASCII format. I used texts of old novels with approximately 1 to 2 million characters. You need to preprocess the text to eliminate punctuation, capitalization, and, optionally, spaces.The simplest way to tabulate the estimated probabilities in Python is to build a dictionary structure in which each entry is a key-value pair whose key is a two-letter string, and whose value is a list of the 26 probabilities that the next letter is 'a','b',*etc.* The elements of each value list should sum to 1. To estimate the probability $p(xyz|xy)$ we count the number of occurrences of xyz in the training text and divide by the number of occurrences of xy. This dictionary will be used by the cryptogram solver to evaluate how English-y a test text is. If the test text is

$$a_1a_2\cdots a_n,$$

then the score assigned is the probability

$$\prod_{i=1}^{n-2} p(a_ia_{i+1}a_{i+2}|a_ia_{i+1}).$$

Given two sequences *of the same length,* the one with a better fit to English texts is the sequence with the higher score.

There are two problems with implementing this approach directly: First, a very large number of trigrams will not occur at all, and the presence of zeros in the table means that typical candidate decryptions will have a score of 0: this makes it impossible to measure our progress. A simple and effective way around this is to increase the count of every possible trigram by 1, so that none of the estimated probabilities is 0.

When we make this modification, the score is a positive number, and more likely strings in this model get higher scores than less likely strings of the same length, just as we want. But the value $\prod_{i=1}^{n-2} p(a_ia_{i+1}a_{i+2})$ is going to be very small, less than 10^{-1000} for texts of a few hundred characters, and this will produce floating point underflow. Instead, I suggest you replace this score by the additive inverse of its logarithm, which is

$$-\sum_{i=1}^{n-2} \log(p(a_ia_{i+1}a_{i+2})).$$

These values are positive numbers. Texts with better fit have *smaller* scores.

2.2 Hill-climbing

The algorithm for stepwise improvement of the score is to start with a random permutation of $\{$ ‘a’,‘b’,...‘c’ $\}$ as the decryption key and find the score of the resulting plaintext. In each step, decrypt using a bunch of *neighboring keys*–these are obtained by interchanging two elements of the original key—and choosing the best one, in terms of the score of the resulting decryption, as the next key. There are $13 \times 25 = 325$ transpositions you can try.

What if you don’t find a neighboring key that gives a better score? Algorithms like this one are called ‘hill-climbing’ algorithms: Ours always tries to take the steepest step that it can, to produce the most improvement in one move. But there is a risk that it will get stuck in a local optimum: you can’t climb any higher in a single step, but you may be far from the best score possible.

In this case, the algorithm should remember the best solution it has found so far, but restart with a new random permutation and begin a new probe for a local optimum. After a few hundred such probes (fewer for longer ciphertexts, more for shorter ones) the algorithm usually succeeds returns the candidate plaintext with the best score. My solver, which uses trigrams, came up with the correct solution to the ‘To be or not to be’ cryptogram, which has 153 characters, after 230 probes. The results will vary, even when the program is run with the identical input, because of the random restarting of the algorithm on each probe.

3 Issues

For ciphertexts of more than 100 characters this method described works extremely well, usually producing the correct answer in 200-300 probes. (Each probe itself examines many keys, but the total number of keys tried out is in the neighborhood of one million, a very tiny fraction of the entire keyspace.) More rarely, the algorithm terminates with a plaintext that is *almost* correct, differing from the original plaintext by a cycle of three characters, *e.g.,* ‘c’ replaced by ‘q’, ‘q’ by ‘t’, and ‘t’ by ‘c’. In such cases it probably means that any single transposition *worsens* the score. So while the proposed answer is very close, this method is unable reach the correct answer from the proposed one. Is there a way to tweak the final result to avoid this?

The method as described here fails for shorter texts (around 100 characters). I observed situations where the algorithm came up with decryptions that had better scores than the correct one, even though the result looked like gibberish. This means that the mathematical model is not capable of distinguishing a genuine English text from a scrambled version. I encourage you to devise and experiment with different strategies for this problem.

You might also try to implement a version where word boundaries in the plaintext are preserved in the ciphertext—these are the ones that appear in newspapers and puzzle books, or include both options in your program. Such puzzles are easier to solve, so it should require fewer iterations and work on shorter texts. The algorithm is the same, but you will have to process the training data differently, and the coding might be just a little bit more complciated.

I have implemented this in several languages. Be forewarned: Python is a great language in which to hack the solution out quickly, but the performance is very slow. Each hill-climbing probe to find a local optimum took 2-3 seconds with the 'To be or not to be' example. I implemented a version in C, which required more programming effort, but was approximately 100 times faster. I believe much of the slowdown in Python occurs in the computation of the score for each candidate decryption, which can probably be speeded up by using the Numpy numerical package for Python.

4 Deliverables

These should include your code, along with a brief report on the problem, your approach, the results you obtained, and how to run your program. I will also ask you to demonstrate a working version.