Josh Tomiak
Dr. Sigmon
Math 132
2 May 2022

Machine Learning for Automatic Plaintext Recognition

Introduction

The availability of computers has reshaped the field of cryptography, leaving most classical ciphers susceptible to cryptanalysis by brute force attacks of unprecedented size. Modern computers can attempt to decrypt a given plaintext with millions of possible keys, enough to span the entire space of possible keys for most classical ciphers. However, especially as the space of possible keys grows large, the effectiveness of these brute force attacks relies on efficient methods for recognizing the correct plaintext among the huge numbers of incorrect outputs. These illegible outputs, which are generated every time an incorrect key is used to decrypt the ciphertext, will quickly crowd out the one correct plaintext to the point that a human cannot be expected to identify the plaintext. For instance, there are 157,248 possible 2x2 key matrices for a Hill cipher (under mod 26), far too many for a human to find the correct decryption. Thus, there is a strong need for computer algorithms that can identify a correct plaintext automatically. This task will be termed "automatic plaintext recognition" or simply "plaintext recognition" in the scope of this paper.

This paper investigates methods for automatic plaintext recognition using machine learning, specifically focusing on affine and Hill ciphers (with 2x2 key matrices) because of their susceptibility to brute force attacks. Although some failed methods will also be discussed here, this work saw the development of a successful plaintext recognition algorithm capable of fully automated cryptanalysis of affine and Hill ciphers (again, with 2x2 key matrices).

Although there are existing methods to perform this task, they are not considered here. This is more like a walkthrough of how to perform this process with some simple, original methods rather than actual research building on existing knowledge or anything. Python was used for programming; the code is available at https://github.com/jTomiak-28/cryptography.

The remaining divisions of this paper are as follows:

1. background on machine learning and its application to this problem;
2. methods, including creation of dataset and details on the algorithms;

3. results, including algorithms' performance along with examples of the final cryptanalysis software at work, and
4. discussion of findings and relation to actual methods/research.

Machine Learning Background

Machine learning involves using historical data to guide how an algorithm solves a problem. While traditional algorithms operate by performing steps explicitly defined by a programmer, a machine learning algorithm gathers information from data that changes how it functions without the programmer knowing exactly what these changes are. For instance, a probability-based machine learning algorithm designed to filter email spam might find that in the dataset of emails used to train the algorithm (each labeled spam or legitimate), 90% of emails including the word "free" are in fact spam. Thus, when the algorithm is put to use after training, it often classifies emails with this word as spam. The programmer didn't specify that the word "free" was problematic- the algorithm "learned" this and applied it to future classification independently. This example may seem to be an oversimplification, but such probability-based algorithms are actually one of the most highly regarded spam filtering algorithms, allowing that they typically use more involved probability metrics (try reading about Naive Bayes algorithms).

There are many different types of machine learning algorithms, but they do share some traits. For one, the algorithm will generally function by using the data to set values of internal parameters that will determine aspects of how it functions, in a process known as training. These parameters (AKA weights) represent the aspects of the algorithm that the programmer does not control. The role of the programmer will instead be to configure what data the model sees and to configure some aspects of the training process.

Processing the raw data into a format usable by the algorithm is one of the most important aspects of machine learning. A programmer must first choose features, or the basic units of data the algorithm will be given. Ultimately all algorithms will need numbers, so the features can be any regular unit that can be quantified. In our email spam example or other text classification scenarios, the most obvious features to use are words, but individual characters or groups of *n* characters (*n*-grams) could also work well. In the email spam example with words as features, a simple approach would be to assign every unique word an integer then encode the email as a list of integers representing words (this is called integer encoding).

Another important aspect of machine learning is setting hyperparameters. While the training process that a type of algorithm uses, such as looking at probabilities, will remain largely unchanged from one use case to another, most algorithms allow the programmer to set certain values, called hyperparameters, that affect training. For instance, deep neural networks such as the one used in this work include a hyperparameter called learning rate controlling the degree to which training iterations change the model's internal parameters. This allows neural networks to be successfully applied to many different types of use cases where different learning rates are optimal.

Machine learning appears to be appropriate for plaintext recognition for a few compelling reasons. First, it seems that plaintext recognition is a fairly complex task for which hard-coded rules will often fail. For instance, abbreviations and slang, along with the difficulty of breaking the text into distinct words to begin with, could make a pure dictionary check difficult. More generally, processing language is one of the most common uses of machine learning (enough to merit the abbreviation NLP for Natural Language Processing, a subfield of machine learning). Second, it is trivial to generate very large datasets for plaintext recognition, as we will see below. So machine learning seems very promising for automatic plaintext recognition.

Methods

Setting out to perform plaintext recognition with machine learning, the first step was to create a dataset for the machine learning algorithms to use. This dataset required two types of samples: legitimate messages representing correctly decrypted plaintexts (positive samples) and incorrectly decrypted plaintexts (negative samples). While at first glance it seemed appropriate to generate strings of random characters to serve as negative samples, a more realistic approach was to actually encrypt legitimate messages then decrypt them with incorrect keys. Since in application machine learning models would face positive and negative samples from the same original plaintext, there was no reason both the positive and negative samples couldn't come from the same source. Thus a single large text corpus was sufficient to form all needed data. In this work, the Sentiment140 dataset, consisting of 1,600,498 Twitter messages, was chosen because of its large size, challenges such as informal language and slang, and the short length of the samples (for efficiency, plaintext recognition programs can make use of only some initial

portion of samples, so long samples are not useful). The corpus is available at http://help.sentiment140.com/for-students/.

Next, the Sentiment140 corpus was processed and negative samples were generated to produce a new plaintext recognition dataset. Since the cipher types targeted in this work normally work with letters, all nonalphabetic characters were removed, including spaces between words. All letters were converted to uppercase, then saved as integers on [0,25] where A → 0, B → 2, … , Z → 25. Negative samples were then created. Each even positive sample was encrypted then incorrectly decrypted with a Hill cipher using a 2x2 key matrix, and every odd sample was encrypted then incorrectly decrypted with an affine cipher, to create 1,600,498 negative samples split half and half between Hill and affine decryptions. Once both positive and negative samples were created, all data samples were set to a length of 20 by removing excess characters or padding with the value 26 (machine learning models require all samples to be of the same length). Since 498 of the original Sentiment140 samples were set aside for testing, the final plaintext recognition dataset contained 3.2 million test samples and 996 test samples perfectly balanced between positive and negative samples. The dataset was too large to upload to Github but the programs on Github allow easy replication.

Table 1: Example instances from the plaintext recognition dataset, with integer encodings converted to letters.

Positive	Negative
ISITHORRIBLETHATIWAN	CECCTOSULPQOKGQKTOUJ
IMPLAYINGAGUESSINGGA	HFGZETIEXYFJAASRITTR
JUSTGOTNOTICEVIAEMAI	KJNNJFUSZPOJTFFUIBOL

Four machine learning models were used in this work: a logistic regression classifier (logit), a Support Vector Machine (SVM), a Convolutional Neural Network (CNN), and a simple word-based heuristic model. Logistic regression uses simple statistical analysis of data and is often used as a baseline to evaluate more complex models. SVMs operate by considering all input vectors plotted as points in a multidimensional space, from which they work to find a hyperplane that divides the vectors into positive and negative groups; SVMs are well-known for their ability to generalize to many types of applications. CNNs are a type of deep learning

inspired by the biological study of brain function that use multiple layers of computational units called neurons, with the first layers including convolution operations. Deep learning methods are very complex, generally performing a long repetitive training process of sampling data, predicting classes for the samples and calculating loss (degree of incorrect predictions), then using partial differential equations to update model parameters in a way that minimizes loss. Over many iterations and samples the model approaches a minimum and optimizes the loss function. CNNs have been shown to perform very well for NLP tasks and are known for their ability to make use of large datasets in ways that simpler models cannot. Each of these three models used characters as their data features. Hyperparameter tuning was performed to optimize the performance of the models, then each model was trained and tested. Due to poor initial results for some models, time constraints, and the fact that algorithms such as SVMs scale poorly to large datasets, only the (much faster) logit actually trained on all 3.2 million samples, while the CNN trained on 500,000 and the SVM trained on 10,000. All models were then tested on the same 996-sample testing set.

After testing and deploying these models, it was found that none met the rigorous accuracy demands required for plaintext recognition. While the CNN's test accuracy above 95% seemed very successful, in practice with hundreds of thousands of negatives and only one positive, the 5% of misclassifications were found to prevent the correct plaintext from being discovered. Additionally, it was observed that most of the false positives could be ruled out by some sort of word lookup, as they contained very few complete words.

Thus, a fourth machine learning approach was developed that operated very differently from the rest. The basic problem was to create an algorithm that could recognize words in the plaintext as quickly as possible. However, since the decrypted plaintext was not divided by spaces, checking whether subsets of the text were words was not feasible. Instead, the fastest approach was to do the opposite: check whether each word in a comprehensive list of words could be found in the plaintext. This task was efficiently performed using Python's *in* keyword (eg, *if* word *in* text). The key to this approach was building a word list of minimal size for fast searches, which was done using a simple heuristic machine learning method. A large 200,000-word dictionary obtained from Python's Natural Language Toolkit (NLTK) package and a 10,000-sample subset of the plaintext recognition dataset divided into positive and negative samples were prepared, then words from the NLTK dictionary that appeared at least twice in the

positive samples but no more than once in the negative samples were used to form the new minimal dictionary. With these parameters the minimal dictionary had a length of 2,981 words, making it a promising candidate for fast plaintext recognition. To determine if a given text was the correct plaintext or not, the number of words from the list appearing in the text could be counted and texts with high numbers of words could be classified as plaintext. A text file containing the minimal dictionary is available in the Github repository.

An interesting component of plaintext recognition in this work was the need for machine learning models that would score the decrypted text rather than merely providing a positive or negative classification. Such scoring would help provide resistance to false positives by allowing the cryptanalysis program to output a list of the highest-scoring possible plaintexts rather than only the first attempted decryption that the machine learning model classified as positive. Of the mainstream machine learning models, only the CNN performed well enough in the testing phase to merit this added functionality, so the model was modified to output the raw loss rather than a softmax transformation of the loss, which successfully led to the model outputting a higher score for more "plaintext-like" inputs. For the word search method, such scoring was actually the easier case—the number of words from the dictionary found in the text served as the score. In fact, the word search method was never repurposed for binary classification or tested like the other methods since it was designed to work best in this scoring format.

Finally, Python programs capable of utilizing these machine learning methods for fully automated cryptanalysis were developed. One program breaks Hill ciphers with 2x2 matrices (hillbreaker2.py) while the other breaks affine ciphers (affinebreaker.py). In each case, the programs take an input ciphertext from the user, try decrypting the ciphertext with every possible key for the respective type of cipher, and output a list of the ten highest-scoring texts, along with the associated keys and scores. The word search method, which performs best, is used in these programs, although the CNN is coded in too and can be used instead if a simple change is made to the code. For efficiency, only the first 100 letters of long ciphertexts are used (or the first 20 letters for the CNN). The programs are available in the Github repository.

Results

The accuracy results of the first three machine learning approaches are shown in Figure 1, where accuracy represents the percentage of test samples that were correctly classified. It is apparent that the logit and SVM are entirely unsuitable for plaintext recognition, but the CNN does fairly well with 95.8% accuracy. Note that the fourth method was not repurposed to function for binary classification, so it was not tested.

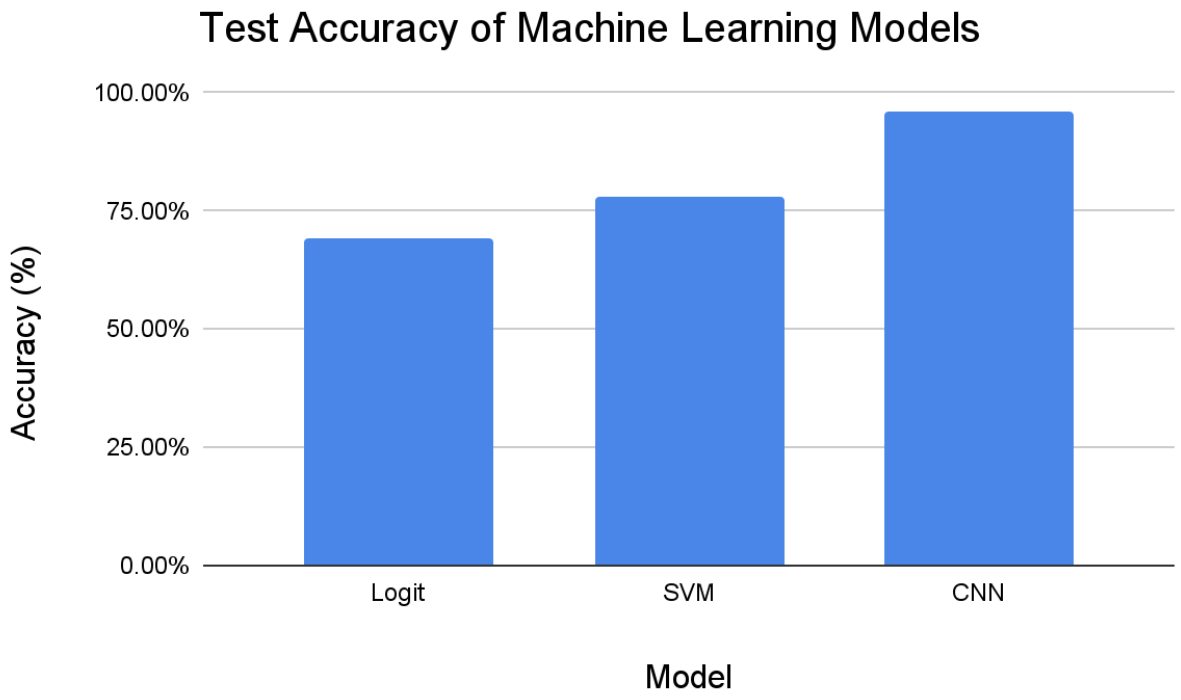

Figure 1: Test accuracy of the three machine learning models used in this research. The CNN appears to be the only model suitable for use in plaintext recognition at 95.8% accuracy.

Given that the CNN and word search method appear to be the best possibilities for plaintext recognition, how do they perform when used for brute force cryptanalysis? No comprehensive evaluations were carried out due to time constraints, but it appears that the CNN performs acceptably for affine ciphers, generally scoring the correct plaintext highest or well within the top ten (Example 1), but that for Hill ciphers it sometimes does not find the correct plaintext at all (Example 2). This outcome can be attributed to the large imbalance between the one correct plaintext and the many incorrect decryptions—while the affine cipher only has 311 possible keys, the Hill cipher has 157,248, overwhelming the results with false positives. The

word search method, however, is robust for both cipher types, almost always finding the correct plaintext as the highest-scoring result (Examples 3 and 4).

Example 1: CNN plaintext recognition with affine cipher. The program successfully found the correct plaintext "HELLOTHEREIHOPETHISMESSAGEREACHESYOUWELL". The result format is (plaintext, key parameter a, key parameter b, plaintext recognition score).

```
C:\Users\Tomiakfamily\Documents\Govschewlwork\2021-2022\Cryptography\Python>affinebreaker.py

--------Affine cipher cryptanalysis--------

Enter ciphertext: CBMMNGCBOBLCNWBGCLXVBXXRTBOBRJCBXZNPHBMM
('HELLOTHEREIHOPETHISMESSAGEREACHESYOUWELL', 9, 17, 1024.8468017578125)
('TENNCBTEREYTCHEBTYWSEWWKOEREKUTEWACGQENN', 7, 25, 276.7539367675781)
('IHSSTMIHUHRITCHMIRDBHDDXZHUHXPIHDFTVNHSS', 1, 20, 264.7167663574219)
('LOHHEZLOBOKLEDOZLKAGOAASMOBOSQLOAUEYWOHH', 17, 23, 183.11666870117188)
('INKKFOINANPIFMNOIPHRNHHLBNANLZINHXFVJNKK', 5, 14, 164.80735778808594)
('NODDCJNOBOENCTOJNESUOSSYWOBOYGNOSQCAIODD', 25, 15, 146.740478515625)
('DETTSZDEREUDSJEZDUIKEIIOMEREOWDEIGSQYETT', 25, 5, 132.2401123046875)
('BUTTADBUHUMBALUDBMSEUSSCQUHUCYBUSGAOKUTT', 15, 13, 80.64362335205078)
('STIIHOSTGTJSHYTOSJXZTXXDBTGTDLSTXVHFNTII', 25, 20, 79.566238403320 31)
('DAHHKPDANAEDKLAPDEOIAOOWCANAWYDAOUKQSAHH', 9, 1, 64.60140228271484)
```

Example 2: CNN plaintext recognition with Hill cipher. The correct plaintext was "URGENTPROBLEMWITHDRAWLALLNOW" but the program doesn't have this result ranked in the top ten. The result format is (plaintext, key matrix, plaintext recognition score).

```
C:\Users\Tomiakfamily\Documents\Govschewlwork\2021-2022\Cryptography\Python>hillbreaker2.py

--------Automatic Hill cipher cryptanalysis--------

Enter ciphertext: XWEQRJTFLOIHAONSFBYLFMDKBBMA
('SHAEHSTSETZTMYIDTEHZIVCHDUWE', array([[11, 17],
       [ 1,  4]]), 772.4798583984375)
('SNAIHGTOETZHMEIHTMHHIRCLDQWO', array([[15,  7],
       [ 7,  2]]), 726.075439453125)
('INASTUHMWHBTOWSTHOTTSJYPXKEM', array([[11, 19],
       [19, 24]]), 677.1968994140625)
('FIKAITEHFWLBEOHSIHTTNSTYKXUE', array([[ 3, 12],
       [ 1,  5]]), 657.7388305664062)
('NISAUTMHHWTBWOTSOHTTJSPYKXME', array([[19, 24],
       [11, 19]]), 652.3319091796875)
('SRACHLTHETZMMIIBTNHIIXCFDJWM', array([[16, 11],
       [15,  8]]), 650.9119262695312)
('RIEATTRHBWEBWOTSDHATLSLYNXWE', array([[ 1,  4],
       [ 4, 17]]), 650.36328125)
('SNASHUTMEHZTMWITTOHTIJCPDKWM', array([[15,  7],
       [19, 24]]), 633.0432739257812)
('SNAIHTTBETZUMEIHTZHUIRCLDDWO', array([[2, 7],
       [7, 2]]), 632.1907958984375)
('ONGIPGNOETHHIEOHFMTHCRILXQYO', array([[23,  9],
       [19, 18]]), 629.3990478515625)
```

Example 3: Word search plaintext recognition with affine cipher. The program successfully found the correct plaintext "HELLOTHEREIHOPETHISMESSAGEREACHESYOUWELL". The result format is (plaintext, key parameter a, key parameter b, plaintext recognition score).

```
C:\Users\Tomiakfamily\Documents\Govschewlwork\2021-2022\Cryptography\Python>affinebreaker.py

--------Affine cipher cryptanalysis---------

Enter ciphertext: CBMMNGCBOBLCNWBGCLXUBXXRTBOBRJCBXZNPHBMM
('HELLOTHEREIHOPETHISMESSAGEREACHESYOUWELL', 9, 17, 20)
('DETTSZDEREUDSJEZDUIKEIIOMEREOWDEIGSQYETT', 25, 5, 3)
('UFOODCUFSFZUDIFCUZXTFXXLPFSFLVUFXBDHRFOO', 7, 18, 2)
('PIHHORPIVIAPOZIRPAGSIGGQEIVIQMPIGUOCYIHH', 15, 11, 2)
('BUTTADBUHUMBALUDBMSEUSSCQUHUCYBUSGAOKUTT', 15, 13, 2)
('LEDDKNLEREWLKVENLWCOECCMAEREMILECQKYUEDD', 15, 19, 2)
('BEXXUPBEREABUTEPBAQWEQQICEREIGBEQKUOMEXX', 17, 11, 2)
('ORKKHCORERNOHGRCONDJRDDVPRERVTORDXHBZRKK', 17, 24, 2)
('FULLWXFUHUAFWRUXFACGUCCOKUHUOEFUCYWSIULL', 19, 11, 2)
('FADDIZFANAYFIBAZFYGWAGGCMANACOFAGQISEADD', 21, 1, 2)
```

Example 4: Word search plaintext recognition with Hill cipher. The program successfully found the correct plaintext "URGENTPROBLEMWITHDRAWLALLNOW". The result format is (plaintext, key matrix, plaintext recognition score).

```
C:\Users\Tomiakfamily\Documents\Govschewlwork\2021-2022\Cryptography\Python>hillbreaker2.py

--------Automatic Hill cipher cryptanalysis---------

Enter ciphertext: XWEQRJTFLOIHAONSFBYLFMDKBBMA
('URGENTPROBLEMWITHDRAWLALLNOW', array([[ 6, 19],
       [ 5,  8]]), 7)
('ETWINEDQIDXLSIMBROIFULADXEIE', array([[ 1,  7],
       [ 1, 12]]), 5)
('JJMETALKLFKDMIVBVWOHXTLNTQGS', array([[ 5, 21],
       [12, 19]]), 5)
('WJMETALKYFXDMIIBVWBHKTYNTQGS', array([[ 5, 21],
       [25,  6]]), 5)
('JJWMUTWLZLJKEMHVKVNOPXPLATEG', array([[ 6,  3],
       [11, 11]]), 5)
('GNAILTPBKTRUEEUHPZLUURSLBDQO', array([[ 6, 21],
       [ 7,  2]]), 5)
('ZSMASHMTDEZZOMFIWTNHHIHCADOW', array([[ 9, 10],
       [ 9,  9]]), 5)
('XUWGLNFPFOOLWMTITHERBWFALLYO', array([[11,  2],
       [ 8, 17]]), 5)
('PKSECRETTYDUGARAGFBLHSUQGBAM', array([[13, 24],
       [ 1, 13]]), 5)
('TWKMMTALLYDXWMTIEUXBZKJYITOG', array([[17,  2],
       [23, 17]]), 5)
```

The amount of time each type of plaintext recognition takes is also an important consideration, especially for applications beyond Hill and affine ciphers where many more results must be evaluated. Runtime results for a single trial plaintext for Hill and affine ciphers are shown in Figure 2, where the word search method can be seen to be faster than the CNN by more than a minute.

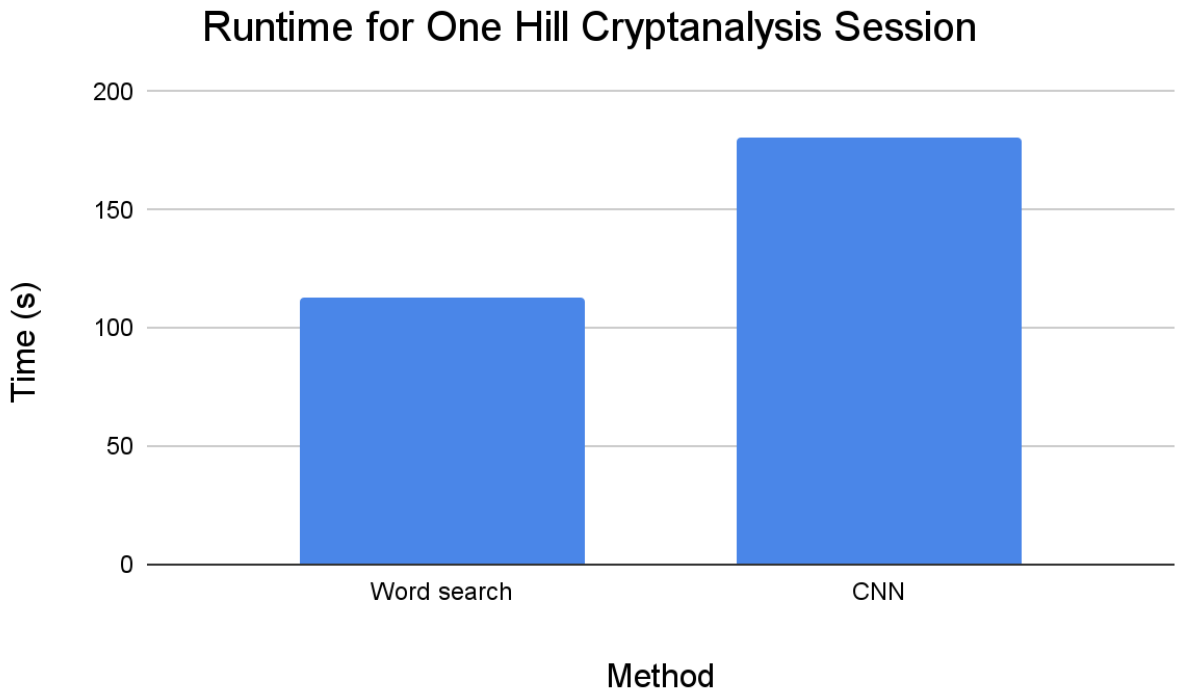

Figure 2: Runtime by plaintext recognition algorithm for one Hill cryptanalysis session.

Discussion

The word search method is without doubt the best method considered here for automatic plaintext recognition. Though the CNN shows some promise and can operate effectively in some cases, the word search method is much more consistent and faster, as seen most clearly when applied to Hill cipher cryptanalysis. Plus, the word search method leaves the potential to decrease this runtime even further by making the parameters that were used when building the minimal dictionary more stringent.

Overall, using the word search method to recognize plaintexts for affine and 2x2 Hill ciphers is a very limited use case. This technique could be very effectively paired with any existing cryptanalysis method to provide fully automated functionality. Imagine a Vigenere cryptanalysis program that automatically checks the index of coincidence, performs Friedman and Kasiski tests, then tries scrawls for a number of the most promising keyword lengths. Automatic plaintext recognition with the word search method would be effective at finding the correct plaintext automatically because it would allow the algorithm to try numerous promising keyword lengths or scrawl assignments but only output the one correct plaintext. Indeed, while

the word search method cannot substitute for the main thrust of a cryptanalysis attack, it can undoubtedly hasten and automate attacks of any kind, removing the need for a human to check the final result.

How do the results here size up to real cryptographic methods? Once I'd completed my experiments, I found this article describing many of the concepts I'd been investigating: http://practicalcryptography.com/cryptanalysis/stochastic-searching/cryptanalysis-simple-substitution-cipher/. First, the idea of plaintext recognition is indeed deemed essential to automatic cryptanalysis. In the field of cryptanalysis, the score generated by a plaintext recognition algorithm is called "fitness" and the plaintext recognition method is called a "fitness function." According to the article, popular fitness functions often use statistical methods relating to observed frequency of letters or *n*-grams in the language in question, especially quadgrams. However, just as observed here, the author notes that plaintexts with unusual frequencies tend to become difficult to decrypt in these situations, so attempting to find words in the plaintext is often an effective alternative, if somewhat slower. The authors also show how a good fitness function can be used to break monoalphabetic substitution ciphers using a hill-climbing algorithm. Notably, the word search method fitness function used in this paper still looks promising and may have potential to address disadvantages of the word lookup methods outlined by the authors in this related article, where an algorithm attempts to break the text into words: http://practicalcryptography.com/cryptanalysis/text-characterisation/word-statistics-fitness-measure/. I couldn't immediately find anything discussing using mainstream machine learning algorithms such as SVMs and CNNs to act as fitness functions, and I think that with a different approach, perhaps using some different features such as *n*-grams, these methods may still have great potential.

In conclusion, this work has walked through some original exploration of the plaintext recognition problem, including the creation of a fast and accurate algorithm for automatic plaintext recognition and the implementation of this algorithm to make programs that automatically break affine and 2x2 Hill ciphers. All code used in this work was made available online at https://github.com/jTomiak-28/cryptography.